DENIAL-OF-SERVICE (DOS) ATTACKS AND COMMERCE INFRASTRUCTURE IN PEER-TO-PEER NETWORKS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Neil Daswani
November 2004

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dan Boneh

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Armando Fox

Approved for the University Committee on Graduate Studies.

Acknowledgement

This dissertation is the culmination of years of research that lies at the intersection of distributed systems, security, and databases. I thank God for giving me the strength, persistence, and determination to complete the dissertation. I also have many people to thank, as I have learned so much from so many.

My parents deserve more credit than I could ever express for fully supporting me. They have been a constant source of dependable advice, and have also kept me "on track" in many ways. My mother, Renu Daswani, has always encouraged me to have the most positive outlook that I could possibly have, and my father, Murli Daswani, has worked amazingly hard over the years to financially support the goals of our entire family. If I am able to provide the same level of emotional and financial support for my children someday, I will consider myself a successful parent. I thank my brother, Susheel Daswani, for always playing devil's advocate as I considered various career options, teaching me the value of having good friends, and how to be firm in one's thinking.

I would like to thank Bharti for teaching me about balance in life. Before meeting Bharti, my life was spent at the extremes– working too hard most of the time, and playing too hard too little of the time. Bharti has taught me how to have a little bit of fun each day. I also thank Bharti's parents, Jagdish and Vanita Mankaney, as well as Bharti's sister, Kamini Mankaney, for their support over the years.

I would like to thank my advisor Hector Garcia-Molina, who I first met while I was working at Bellcore. He has taught me many lessons. While I have learned much technical knowledge from him, the most important things that I have learned from him have been non-technical. He has been extremely patient with me, as over

the years I flirted with many ideas, some of which eventually went nowhere! He has always provided me with guidance along the way, and taught me about "good taste" in deciding which ideas to pursue, and which to drop.

There are many other professors that have been instrumental in helping me acheive my academic accomplishments. Al Aho, whom I had the pleasure of serving as a teaching assistant for at Columbia, opened my eyes to the possibility of pursuing a doctoral degree. Shamim Naqvi placed immense confidence in me at Bellcore, and set me on track to become a good manager and public speaker.

Dan Boneh instilled the principles and tenets of computer security in me when I arrived at Stanford. Rajeev Motwani, as head of the PhD Program Committee, watched me grow in many roles– as a PhD student, as an engineer, and as an entrepreneur; I thank him for keeping an eye on me. Andreas Paepcke provided me with great advice and professional support over the years. I also thank Armando Fox for serving on my reading and defense committees. Kathi DiTommaso also deserves a hand for giving me the opportunity to serve the Computer Science Department by facilitating extracurricular activities, and I am also grateful for her help through all of the administrative issues that I had to deal with over the years.

I also have many peers at Stanford to thank. Arturo Crespo helped me decide to come to Stanford, and also encouraged me to choose Hector as an advisor. Qi Sun took some of my research in interesting directions that I did not anticipate, and I always enjoyed having philosophical chats with him. Mayank Bawa, Brian Cooper, Bevery Yang, Sergio Marti, TJ Giuli, Prasanna Ganesan, Philippe Golle, and Nagendra Modadugu have been a pleasure to work with. I have learned a little something from each of them.

In the midst of my PhD, I took a leave to help start Yodlee, a start-up company that I hope will continue to change the way financial institutions see the web. Venkat Rangan hired me early on in the company's history, and put me on a path to build good management skills. I thank Vivin Ramamurthy and Melanie Flanigan for beleiving in me, my group, and the products we built. I also thank Shailesh Rao for teaching me about confidence, and much of what I know about business.

Last but not least, I have many friends that have been instrumental in helping me

shape my personal and professional goals: Vipul Khamar, Saeed Butt, Aniket Patel, Ketan Dave, Suja Raju, Jyoti (Moni) and Neeraj Sharma, and Ami and Nimesh Shah.

Finally, I hope that all the people that have helped me achieve the milestone of completing my PhD will continue to walk with me through life. Hopefully, with my degree in hand, and all that I have learned, I hope to make contributions to the world that my family, friends, and colleagues can be proud of.

Abstract

This dissertation studies denial-of-service (DoS) attacks in peer-to-peer (P2P) networks, and electronic commerce infrastructure for such networks.

In the first part of this dissertation, we propose attack containment techniques that make search and resource discovery protocols in P2P networks more resilient to DoS attacks. We describe the importance of attack containment as a complement to prevention, detection, and recovery techniques. We describe a simple but effective traffic model that can be used to understand the effects of application-layer, query-flood DoS attacks in P2P networks. We develop a threat model that describes how good and malicious nodes are captured by the traffic model, and we describe the results of simulations based on the model. Simulations are run on both synthetic and real Gnutella network topologies, and on the Chord distributed hash table network. We analyze how different query acceptance policies can contain the effects of query-flooding DoS attacks. We also describe a DoS attack against the resource discovery mechanism of a P2P protocol called GUESS. Nodes in a GUESS network use a data structure called a pong-cache to keep track of other nodes in the network, and malicious nodes can "poison" pong-caches with malicious node ids. We describe how to contain such attacks using an ID smearing algorithm and a dynamic network partitioning scheme.

In the second part of this dissertation, we develop an architecture that can be used to support electronic commerce in a P2P network. We report on a prototype implementation of the architecture that allows mobile and wireless devices to make purchases using a digital cash scheme called PDA-Payword.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The existing client-server based Internet is vulnerable to various forms of attacks against servers. If a few servers are disabled due to a malicious attack, many clients may be unable to continue to function or execute transactions (such as making electronic commerce purchases). For instance, in February 2000, the eBay, E-Trade, Amazon, CNN, and Yahoo web sites were victims of denial-of-service (DoS) attacks, and some were disabled for almost an entire business day [31, 82]. This meant lost revenues and interruption of service for legitimate users.

The attacks that took place in February 2000 were "network-layer" denial-of-service attacks, in which the perpetrators commandeered a number of client hosts, and instructed those clients to send large numbers of network-layer data packets to the victim servers. Because there were relatively few such servers, and they were overwhelmed with the large number of packets, the servers were unable to respond to packets they received from legitimate users. These attacks were, in part, possible due to the current client-server based architecture of the Internet in which relatively few servers are used to provide service to a much larger number of clients.

In contrast, in a P2P (or peer-to-peer) system, nodes in the network can function as both clients and servers, and server functionality is distributed across many nodes in the network. In such a system, even if a fraction of those nodes is unavailable (due to, say, a network-layer denial-of-service attack), clients in the network can still execute transactions due to the larger number of nodes that can function as

servers. However, although P2P networks are not as susceptible to network-layer denial-of-service (DoS) attacks as traditional client-server systems are, P2P networks are susceptible to "application-layer" DoS attacks.

Most techniques that have been developed to date to deal with denial-of-service focus on network-layer attacks [13, 136, 191, 192, 157, 158, 190, 63, 181, 70, 27, 146, 86, 145, 4, 210, 147]. Yet, P2P systems are very vulnerable to application-layer attacks, even if they are immune to network-layer attacks. In an application-layer attack, large numbers of application requests or messages (e.g., deliver email, open account, find page) can deny service to clients. These attacks can be more damaging, relative to the effort that the malicious party needs to expend, than network-layer attacks since each small message can cause a server to waste significant resources. For example, a short query message to a web search engine, asking for many terms that occur in many pages may cause it to retrieve and process many long inverted lists. Fortunately, since requests require the server to expend a significant amount of CPU and/or disk I/O resources, the server can afford to spend some time deciding which requests to honor. In contrast, routers processing network packets cannot afford to spend as much time making such decisions, else the overhead will be too high.

In a P2P system, nodes are typically expected to do work for each other, and nodes rely on other nodes to execute application-layer requests to carry out system functions such as searching for documents. For example, a node searching for a document can ask other nodes to look for the document in their local file systems or databases. Those nodes can, in turn, ask additional nodes to look for the document. In such a fashion, a single node can indirectly ask many, many other nodes in the network to do work (i.e., query a database) on its behalf. If that node is malicious, it may be able to ask many, many nodes to do a lot of work on its behalf, thereby making them unavailable to do work on behalf of good, legitimate nodes in the system. That is, a malicious node may be able to deny service to good nodes simply by asking nodes to do a lot of work on their behalf. Clearly, some way of fairly dividing up the processing capacity of nodes in the network is needed. The development and evaluation of techniques to divide up processing capacity to mitigate the effect of application-layer denial-of-service attacks is a central focus of this dissertation.

In this dissertation, we build an understanding of, and develop techniques to mitigate application-layer attacks in P2P networks. Once a P2P network is insulated against DoS (and other types of attacks), the network may be able to serve as a secure substrate for electronic commerce transactions. Thus, once we study how to mitigate application-layer DoS attacks in P2P networks, we then study how electronic commerce might take place in such networks.

In this introduction, we provide an overview of P2P systems, discuss some of the major security challenges that need to be addressed in the context of P2P systems, discuss how existing contributions in the field of computer security are related to our work, and provide a brief overview of related work in the area of electronic commerce. We then explicitly state our thesis, and outline how the chapters of this dissertation support our thesis.

1.1 P2P Systems

A P2P network consists of a number of peers (or nodes). Each peer maintains one or more application-layer connections to a few other peers. Peers may join the network at any time by connecting to some other peer that is already part of the network, and may leave the network at any time. The peers in the network usually execute a protocol that helps them acheive some higher-level goal (i.e., searching for a file in a distributed system). In addition, peers may take on the roles of both traditional clients and servers.

For example, a peer in a network P_1 may have a collection of documents that it stores. P_1 may be interested in searching for a particular document D_4 that it may not have in its local collection. P_1 may forward its query to another peer P_2. In this case, P_1 is playing the role of a client, and P_2 is playing the role of a server. If P_2 does not have document D_4 in its local collections, it may take on the role of a client as well, and forward the query to an additional peer (server) that it is aware of that may have the document. Upon receiving a response from another peer peer, P_2 may resume its role as a server, and return any responses / results to P_1.

P2P systems have recently become a very active research area, due to the popularity and widespread use of P2P systems today, and their potential uses in future applications. P2P systems have emerged as a popular way to share huge amounts of data (e.g., [74, 142, 137]). In the future, the advent of large-scale ubiquitous computing makes P2P a natural model for interaction between devices (e.g., via the web services [201] framework).

P2P systems are popular because of the many potential benefits they offer: adaptation, self-organization, load-balancing, fault-tolerance, availability through massive replication, and the ability to pool together and harness large amounts of resources. For example, file-sharing P2P systems distribute the main cost of sharing data – bandwidth and storage – across all the peers in the network, thereby allowing them to scale without the need for powerful, expensive servers.

Despite their many strengths, however, P2P systems also present several challenges that are currently obstacles to their widespread acceptance and usage – e.g., security, susceptibility to DoS attacks, efficiency, reliability, and performance guarantees like atomicity and transactional semantics. The P2P environment is particularly challenging to work in because of the scale of the network and unreliable nature of peers characterizing most P2P systems today. Many techniques previously developed for distributed systems of tens or hundreds of servers may no longer apply; new techniques are needed to meet these challenges in P2P systems. In this dissertation, we will focus on developing techniques to reduce the susceptibility of P2P networks to application-layer DoS attacks. In Section 1.3 of this introduction, we provide references to other projects that address other security issues in P2P networks, and discuss the security challenges in P2P systems in more depth. Much work is also taking place on search efficiency and reliability in P2P systems, and the reader is referred to [42] for a discussion of open problems and related work on efficiency and reliability in P2P networks.

1.2 Types of P2P Systems

P2P systems can be roughly categorized as being of one of three major types: structured, unstructured, or non-forwarding. Not all P2P systems will cleanly fit into one of these categories, but we will mention some systems that are prototypical of each category. Also, note that in the following descriptions, we only capture the basic proposals in each category, and we do not account for or discuss the many optimizations that exist. Indeed, some of the proposed optimizations and improvements in the literature blur the lines between the different types of P2P systems. We describe the three different categories of P2P systems here as our treatment of application-layer DoS attacks in this dissertation takes all three of them into consideration.

1.2.1 Structured

In a structured P2P network, sometimes also called a distributed hash table (or DHT), each node stores $(key, value)$ pairs. The set of keys is partitioned across all of the nodes in the network, such that given a key, there is exactly one node that stores the corresponding values. Given a key, we would like to be able to determine which node in the network stores the corresponding values. The main challenge in such a network is: given a "query" that specifies a key and originates at some node in the network, how do we route that query to a node that stores the corresponding values?

To address this main challenge, each node in a structured network maintains a routing table that contains addresses of other nodes. Given a key, a routing table specifies the next node to which the query should be routed. In most DHTs, there are constraints that entries in a routing table must satisfy in order for routing to work. To illustrate how basic search and routing takes place in a DHT, we provide an example using the Chord [193] DHT.

In Chord, routing tables are used to structure the nodes of a P2P network into a "ring" in which there are exist "long-hop" connections to provide for efficient routing. Nodes in a Chord network choose ids in the range in a n-bit address space. Keys also come from the same n-bit address space.

Consider a small Chord network of three nodes as shown in Figure 1.1 in which

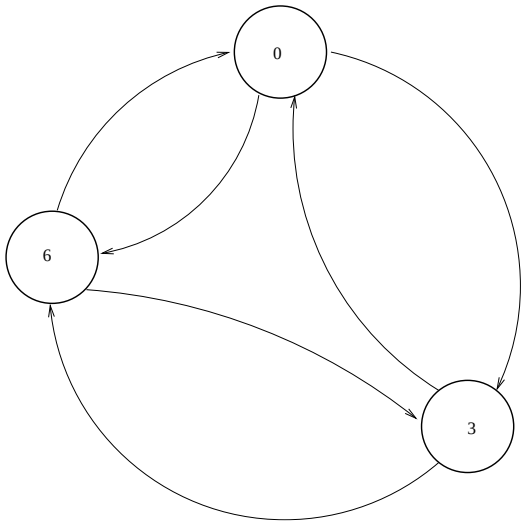

Figure 1.1: An example of a small 3-node Chord network.

the nodes have ids 0, 3, and 6. In our small network, node ids and keys are chosen from a 3-bit address space. The nodes are responsible for storing $(key, value)$ pairs where $key \in [0, 8)$. The node with id 3 is responsible for storing pairs for which $key \in (0, 3]$; the node with id 6 is responsible for storing pairs for which $key \in (3, 6]$; and, the node with id 0 is responsible for storing pairs for which $key \in (6, 0]$. (The maximum number of nodes that we choose to support in this small example is 8, and arithmetic on keys is done modulo 8.) In general, the node with id i is responsible for storing pairs for which $key \in (predecessor(i), i]$, where $predecessor(i)$ is the largest id of the node in the network that is smaller than i.

Each node with id i in Chord maintains a pointer to its successor, denoted by $successor(i)$, which is the node that has the smallest id that is greater than i. These pointers together compose a ring that can be used to route queries. If a node i has a query for the pair with key j, and $j \notin (predecessor(i), i]$, then node i can forward the query to its successor. If $successor(i)$ is not responsible for j, then the query is forwarded to $successor(successor(i))$, and so on until the appropriate node responsible for storing the pair with key j is found.

Since doing a search for a key in this fashion can take linear time with respect to

the number of nodes in the network, Chord maintains additional pointers to speed up search, and these additional pointers make up a node's routing table. A node's routing table stores $\log N$ pointers, where N is the number of nodes in the network. The kth pointer in the routing table for a node with id i must satisfy the constraint that it point to $successor(i + 2^{k-1})$. Each such pointer is a "chord" across the ring of nodes.

When a node has a query for a key, it sends the query along the chord that gets the query as close to the destination without overshooting it. In our small example, each node maintains a routing table with $\log_2 N = \log_2 8 = 3$ pointers. The first pointer is to a node's successor. The second pointer is to a node whose id is at least two greater than its own, and the third pointer is to a node whose id is at least four greater than its own. The node with id 0, for instance, can have its first, second, and third pointers point to the nodes with ids 3, 3, and 6, respectively. If the node with id 0 has a query for key 7, it will send that query directly to the node with id 6.

We have not explicitly depicted the routing tables in Figure 1.1. Also, although each node has three distinct entries in their routing tables, we have not depicted redundant pointers in the case that some of the entries point to the same node. Finally, while having every node point to every other node in this small 3-node graph might seem like overkill, it is easy to imagine that in a much larger network with a much larger address space, the pointers in each of the routing tables become essential for efficiency.

There are many proposals other than Chord for how the routing table should be structured, and how nodes should initialize and update their routing tables when nodes join and leave the network [193, 164, 176, 212, 38]. Due to the structure of routing tables and the constraints placed on the entries in the tables, most of these proposals are able to route queries to their destinations in $O(\log N)$ hops.

While search can be efficient in DHTs, they do have some disadvantages. In particular, nodes may not have control over what set of keys they are responsible for storing. Nodes may therefore be responsible for maintaining values that could correspond to content that might be contraband. For instance, most users of a P2P network would not want to have their machines be responsible for storing values for

the key corresponding to "child pornography."

In addition, when nodes join a DHT, they may become responsible for part of the key space, and for storing some values that are already stored by other nodes in the network. Such values may have to be transferred to the node that joins. A node joining the network must also populate its routing table which might take, for instance, as long as $O(\log^2 N)$ time (assuming that searching for an id takes $O(\log N)$ and a routing table has $O(\log N)$ entries).

Another disadvantage of DHTs is that while they can efficiently search for specific keys, they do not as easily support more general queries. For example, a DHT can easily and efficiently search for a file whose metadata matches a particular fixed keyword, such as "gum." The term "gum" can be hashed to a search key, and a lookup is done for that search key. However, DHTs cannot as easily search for files whose metadata matches a regular expression such as "gum*", which might match "gumshoe", "gumball", or "gumdrop."

There exist various improvements to basic DHT schemes to help address some of these problems, but we have only discussed the basic trade-offs here.

1.2.2 Unstructured

In an unstructured P2P network, by contrast, each node decides what set of content it would like to store. Nodes in an unstructured network are not relied upon to store particular keys or sets of documents. Nodes do not usually maintain a routing table, but instead just maintain a set of arbitrary "neighbors" to which they send and from which they receive queries.

To illustrate how an unstructured P2P network works, consider a Gnutella [74] network. Nodes in a Gnutella "flood" the network with their queries. That is, when a node is interested in searching for a document in Gnutella, that node broadcasts its query to all of its neighboring nodes. If any of the neighboring nodes have a document that matches the query, they send back a response to the node that issued the query. In addition, the neighboring nodes broadcast the query to all of their neighboring nodes. To prevent queries from traversing nodes in the network ad infinitum, nodes

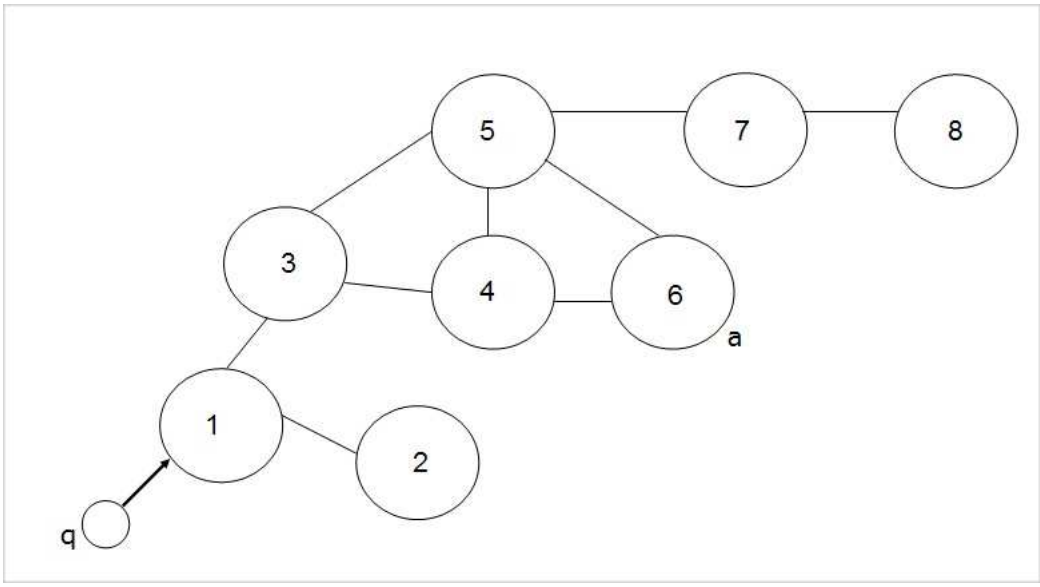

Figure 1.2: An example of a small Gnutella network.

stamp queries with a TTL (time-to-live). Each time that a query is forwarded from one node to another, the TTL is decremented, and when the TTL is equal to zero, the query is no longer forwarded to additional nodes. Finally, to prevent queries from traversing cycles, each query is assigned a unique descriptor id [1], and each node keeps track of all the descriptor ids it has processed recently. If a node determines that it has received a query that it has already processed, it does not process it again, and does not forward that query onto neighboring nodes.

To provide a concrete illustration of a Gnutella network in action, consider the Gnutella network in Figure 1.2. The figure shows a Gnutella network consisting of eight nodes, in which a query (denoted by "q") is being submitted to node 1 and node 6 contains the answer to the query (denoted by "a"). When node 1 receives the query, it stamps it with a TTL of, say, 3, and looks for an answer in its local repository of files (to which it finds none). It then broadcasts (forwards) copies of the query to each of the nodes to which it is connected. When nodes 2 and 3 receive the query, they decrement the TTL of the query to 2. Nodes 2 and 3 look for an answer

[1] The descriptor ids are unique with high probability, and are usually the output of a hash function.

in their local repository but do not find an answer. Node 2 has no other neighbors to which the query, and does not forward the query back to node 1 since that is the node from which the query arrived. Node 3 forwards the query to nodes 4 and 5. Nodes 4 and 5 decrement the TTL of the query to 1, and are unsuccessful at finding matches to the query in their local repositories. Node 4 then forwards the query to nodes 5 and 6. When node 5 receives the query a second time, it does not process it again because it has already seen the query's unique descriptor.

When node 6 receives the query, it decrements the query's TTL to 0, and then determines that it has an answer to the query. Node 6 generates a "query hit" message and sends it back to node 4 [2].

When nodes recieve queries, they keep track of the node from which they received the query, and this information is used to route the query hit back to the node that originally issued the query. Once node 1 receives the query hit, it makes a direct TCP connection to node 6 to download the answer to the query. Note that the document is downloaded out-of-band, and is not transferred through the topology of the Gnutella network.

To complete our example, we note that after node 5 processed the query, it forwarded the query to node 7 in addition to node 6. Node 7 decrements the TTL of the query to 0, processes the query, and then does not forward the query onto node 8 since its TTL expired.

While nodes in an unstructured network forward queries from one node to another, there is no "hard" guarantee that a query will make progress towards finding a node that has a corresponding answer. Finally, nodes in an unstructured network can join or leave the network at any time without transferring data to or from other nodes.

As a result of the (lack of) organization in unstructured P2P networks, they do not offer very many guarantees, and could have scalability issues. For instance, a query may have to be sent to every node in the network before a result can be found,

[2]Node 6 can actually receive the query concurrently from nodes 4 and 5. As per the "already seen" traffic management policy, it only processes the query once and responds by sending a query hit message to either node 4 or node 5, but not both. To keep our example simple, we will assume that node 6 receives the query from node 4 first, and sends it the query hit. Regardless of from where node 6 receives the query first, it does not forward it onto any other nodes after processing it because its TTL is 0.

whereas in a DHT, a result can be found in $O(\log n)$ hops (if it exists) in the network.

However, in practice, unstructured networks have a number of advantages over structured networks. Nodes in an unstructured network enjoy a significantly higher level of autonomy, the network is more resilient to node failures and the loss of messages, and unstructured networks are often easier to implement since complex routing tables do not need to be maintained. Scalability is achieved by having only the more "powerful" nodes in the network (e.g., those that have lots of bandwidth) participate directly in the unstructured network, and having less powerful nodes connect to more powerful ones to have their queries serviced. Also, since most queries are for "popular" documents that are relatively well-replicated, such documents can be found successfully using an unstructured network in a reasonable amount of time. Indeed, out of the ten or so major file-sharing P2P protocols most widely used on the Internet (FastTrack, eDonkey, Gnutella [74], etc.), all except one (OverNet) are unstructured.

1.2.3 Non-Forwarding

The key characteristic that distinguishes non-forwarding P2P networks from structured and unstructured ones is that queries are not forwarded from one node to another. Instead, nodes keep lists of "friends" to which they may send their queries. Their friends will process the queries and respond, but will not forward the queries to other nodes. Nodes may issue a query to their friends one at a time, and may stop issuing the query to additional friends once they have received enough search results. The friends that nodes have in non-forwarding networks are similar to neighbors in unstructured networks with the exception that a node does not maintain ongoing transport-layer (i.e., TCP) connections with all of its friends, and it has many more friends than nodes in an unstructured network have neighbors.

The GUESS [44] network is an example of a non-forwarding P2P network. To give the reader a flavor of how a non-forwarding P2P network works, we provide a brief illustration of how a GUESS network functions. Consider the GUESS network in Figure 1.3. Each of the circles depicts a node in the network, and the rectangles

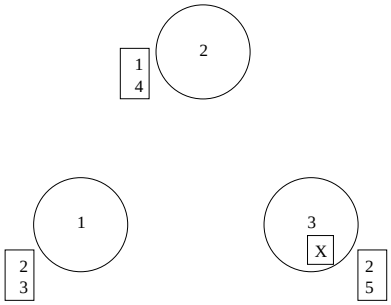

Figure 1.3: An example of a small 3-node GUESS network.

adjacent to each of the nodes contains that node's list of friends. For instance, node 1's friends are nodes 2 and 3, and node 2's friends are nodes 1 and 4. Node 3 has nodes 1 and 5 as friends, and has a document X in its local repository. Note that nodes 4 and 5 are not shown in the diagram.

If node 1 has a query for document X, it might first send that query to node 2 since node 2 appears in its friends list. When node 2 receives the query for document X, it will send a response saying that it has no matches. Node 1 may then send the query to the next friend in its list, node 3. When it sends the query to node 3, node 3 will determine that it has a match, and it will respond with the document X.

Non-forwarding networks have many of the same advantages that unstructured networks do, as compared to structured networks. Namely, nodes in a non-forwarding network enjoy high autonomy, resilience to node failure and dropped messages, and do not need to implement complex routing tables. In addition, non-forwarding networks have an advantage over both unstructured and structured networks in that nodes do not rely on other, potentially untrusted nodes, to route their queries at all. However, the number of search results that can be obtained in a non-forwarding network might be limited by the number of friends that a node has. In contrast, a node that has even just a few neighbors in an unstructured network may be able to obtain many search results because of the query broadcast that occurs at each node. Also, comparing non-forwarding networks to structured networks with respect to search results, there is no guarantee that a query will find its way to a node that may have answers, and the probability that results are found depends upon the number of friends that a node

has.

To increase the number of friends that nodes have, the lists of friends are often exchanged between nodes to increase the breadth of the lists used for search. In our example GUESS network, for instance, node 1 might want to find additional documents matching its query, and would need to find out about additional nodes to query. Node 1 might send a message to node 2 asking for its list of friends, and upon receiving a response can discover node 4's id (IP address). Node 1 can then send its query to node 4 to find additional matches to its query. Of course, there may be no reason to transitively trust that friends of friends will respond at all or respond with legitimate documents, which makes non-forwarding networks vulnerable to a class of attacks that we study in Chapter 5.

Non-forwarding P2P networks are usually used as a component of a larger system, instead of being used as a complete, end-to-end file-sharing system. For instance, GUESS [44] is a non-forwarding P2P system that has been proposed to help nodes become aware of (or discover) other nodes in Gnutella networks. Also, BitTorrent [198] is a non-forwarding P2P system that can be used as part of a larger P2P (or non-P2P) system to make file transfers more efficient by sharing bandwidth between peers. In the case of BitTorrent, "friends" happen to be sets of nodes that are interested in downloading the same document at the same time. A "marker" node sends each of these friends different parts of the document, and the friends upload parts of the document that they do have to other friends while concurrently downloading parts of the document that they do not have yet.

1.3 P2P Security

In this section, we survey research problems associated with security in P2P systems. To ensure proper, continued operation of the system, security measures must be in place to protect against availability attacks, unauthentic data, and illegal access. We describe some of these issues here to give the reader a general familiarity with the area of secure P2P systems. However, in the remainder of this thesis we will focus on addressing particular kinds of attacks against the availability of a P2P system.

Securing P2P applications is challenging due to their open and autonomous nature. Compared to a client-server system in which servers can be relied upon or trusted to always follow protocols, peers in a P2P system may provide no such guarantee. The environment in which a peer must function is a hostile one in which any peer is welcome to join the network; these peers cannot necessarily be trusted to route queries or responses correctly, store documents when asked to, or serve documents when requested. In this part of the paper, we outline a number of security issues that are characteristic to P2P systems, discuss a few examples of research that has taken place to address some of these issues, and suggest a number of open research problems.

We organize the security requirements of P2P systems into four general areas: availability, file authenticity, anonymity, and access control. Today's P2P systems rarely address all of the necessary requirements in any one of these areas, and developing systems that have the flexibility to support requirements in all of these areas is expected to be a research challenge for quite some time.

For each of these areas, it will be important to develop techniques that *prevent*, *detect*, *contain*, and *recover* from attacks. For example, since it may be difficult to prevent a denial-of-service attack against a system's availability, it will be important to develop techniques that are able to 1) detect when a denial-of service attack is taking place (as opposed to there just being a high load), 2) contain an attack that is "in-progress" such that the system can continue to provide some (possibly reduced) level of service to clients, and 3) recover from the attack by disconnecting the malicious nodes.

1.3.1 Availability

There are a number of different node and resource availability requirements that are important to P2P file sharing systems. In particular, each node in a P2P system should be able to accept messages from other nodes, and communicate with them to offer access to the resources that it contributes to the network.

Denial-of-Service (DoS)

A denial-of-service (DoS) attack attempts to make a node and its resources unavailable by overloading the resources at the node. The most obvious DoS attack is targeted at using up all of a node's bandwidth. This type of attack is similar to traditional network-layer DoS attacks (e.g. [70]). If a node's available bandwidth is used up transferring useless messages that are directly or indirectly created by a malicious node, all of the other resources that the node has to offer (including CPU and storage) will also be unavailable to the P2P network.

A specific example of a DoS attack against node availability is a chosen-victim attack in Gnutella (an unstructured P2P network) that an adversary constructs as follows: a malicious node maneuvers its way into a "central" position in the network and then responds to every query that passes thru it claiming that the victim node has a file that satisfies the query (even though it does not). Every node that receives one of these responses then attempts to connect to the victim to obtain the file that they were looking for, and the number of these requests overloads the bandwidth of the victim such that any other node seeking a file that the victim does have is unable to communicate the victim.

The key aspect to note here is that in our example the attacker exploited a vulnerability of the Gnutella P2P protocol (namely, that any node can respond to any query claiming that any file could be anywhere). In the future, P2P protocols need to be designed to make it hard for adversaries to construct DoS attacks by taking advantage of loosely constrained protocol features.

Attackers that construct DoS attacks typically can find and take advantage of an "amplification mechanism" in the network to cause significantly more damage than they could with only their own resources. For example, an attacker could send a packet to a router with a broadcast destination address. The router, upon receiving the packet, then creates many copies of the attack packet and broadcasts it to every host to which it is connected. In this example, the attacker takes advantage of router functionality to "amplify" its attack, whereas otherwise it might only be able to send attack packets to a single host at a time.

Attackers can also have fine-grained control over how their attack is carried out,

and may find or create a back-door communication channel to communicate with "zombie" hosts that they infiltrate using manual or automatic means. Then, the attacker can issue a commands at-will to the zombie hosts to have them participate in the attack. It is important to design future P2P protocols such that they do not open up new opportunities for attackers to use as amplifiers and back-door communication channels.

Induced Node Failure

Aside from DoS attacks, node availability can also be attacked by malicious users that infiltrate victim nodes and induce their failure. These types of failures can be modeled as fail-stop or byzantine failures, which could potentially be dealt with using many techniques that have already been developed (e.g. [116]). However, these techniques have typically not been popular due to their inefficiency, unusually high message overhead, and complexity. In addition, these techniques often assume complete and secure pairwise connectivity between nodes, which is not the case in most P2P networks. Further research will be necessary to make these or similar techniques acceptable from a performance and security standpoint in a P2P context.

In addition, there are many proposals to provide significant levels of fault-tolerance in the face of node failure including CAN [164], Chord [193], Pastry [177], and Viceroy [38]. Security analyses of these types of proposals can be found in [188] and [26]. The IRIS [10] project seeks to continue the investigation of these types of approaches.

Resource Availability

A malicious node can also directly attack the availability of any of the particular resources at a node. The CPU availability at a node can be attacked by sending a modest number of complex queries to bog down the CPU of a node without consuming all of its bandwidth. The available storage could be attacked by malicious nodes who are allowed to submit bogus documents for storage. One approach to deal with this is to allocate storage to nodes in a manner proportional to the resources that a node

contributes to the network as proposed in [33].

We might like to ensure that all files stored in the system are always available regardless of which nodes in the network are currently online. File availability ensures that files can be perpetually preserved, regardless of factors such as the popularity of the files. Systems such as Gnutella and Freenet provide no guarantees about the preservation of files, and unpopular files tend to disappear.

Quality-of-Service

Even if files can be assured to physically exist and are accessible, a DoS attack can still be made against the quality-of-service with which they are available. In this type of a DoS attack, a malicious node makes a file available, but when a request to download the file is received, it serves the file so slowly that the requester will most likely lose patience and cancel the download before it completes. The malicious node could also claim that it is serving the file requested but send some other file instead. As such, techniques such as hash trees [20] could to be used by the client to incrementally ensure that the server is sending the correct data, and that data is sent at a reasonable rate.

1.3.2 File Authenticity

File authenticity is a second key security requirement that remains largely unaddressed in P2P systems. The question that a file authenticity mechanism answers is: given a query and a set of documents that are responses to the query, which of the responses are "authentic" responses to the query? For example, if a peer issues a search for "Origin of Species" and receives three responses to the query, which of these responses are "authentic?" One of the responses may be the exact contents of the book authored by Charles Darwin. Another response may be the content of the book by Charles Darwin with several key passages altered. A third response might be a different document that advocates creationism as the theory by which species originated.

Note that the problem of file authenticity is different than the problem of file

(or data) integrity. The goal of file integrity is to ensure that documents do not get inadvertently corrupted due to communication failures. Solutions to the file integrity problem usually involve adding some type of redundancy to messages in the form of a "signature." After a file is sent from node A to node B, a signature of the file is also sent. There are many fewer bits in the signature than in the file itself, and every bit of the signature is dependent on every bit of the file. If the file arrived at node B corrupted, the signature would not match. Techniques such as CRCs (cyclic redundancy checks), hashing, MACs (message authentication codes), or digital signatures (using symmetric or asymmetric encryption) are well-understood solutions to the file integrity problem.

The problem of file authenticity, however, can be viewed as: given a query, what is (or are) the "authentic" signature(s) for the document(s) that satisfy the query? Once some file authenticity algorithm is used to determine what is (or are) the authentic signatures, a peer can inspect responses to the query by checking that each response has an authentic signature.

In our discussion until this point, we have not defined what it means for a file to be authentic. There are a number of potential options: we will outline four reasonable ones.

Oldest Document. The first definition of authenticity considers the oldest document that was submitted with a particular set of metadata to be the authentic copy of that document. For example, if Charles Darwin was the first author to ever submit a document with the title "Origin of Species," then his document would be considered to be an authentic match for a query looking for "Origin of Species" as the title. Any documents that were submitted with the title "Origin of Species" after Charles Darwin's submission would be considered unauthentic matches to the query. Timestamping systems (e.g. [119]) can be helpful in constructing file authenticity systems based on this approach.

Expert-Based. In this approach, a document would be deemed authentic by an "expert" or authoritative node. For example, node G may be an expert that keeps track of signatures for all files ever authored by any user of G. If a user searching for documents authored by any of G's users is ever concerned about the potential

authenticity of a file received as a response to a query, node G can be consulted. Of course, if node G is unavailable at any particular time due to a transient or permanent failure, is infiltrated by an attacker, or is itself malicious, it may be difficult to properly verify the authenticity of files that G's users authored. Offline digital signature schemes (i.e., RSA) can be used to verify file authenticity in the face of node failures, but are limited by the lifetime and security of public/private keys.

Voting-Based. To deal with the possible failure of G or a compromised key in our last approach, our third definition of authenticity takes into account the "votes" of many experts. The expert nodes may be nodes that are run by human experts qualified to study and assess the authenticity of particular types of files, and the majority opinion of the human experts can be used to assess the authenticity of a file. Alternatively, the expert nodes may simply be "regular" nodes that store files, and will vote that a particular file is authentic if they store a copy of it. In this scheme, users are expected to delete files that they do not believe are authentic, and a file's authenticity is determined by the number of copies of the file that are distributed throughout the system. The key technical issues with this approach are how to prevent spoofing of votes, of nodes, and of files.

Reputation-Based. Some experts might be more trustworthy than others (as determined by past performance), and we might weight the votes of more trustworthy expert nodes more heavily. The node weights in this approach are a simple example of "reputations" that may be maintained by a reputation system. A reputation system is responsible for maintaining, updating, and propagating such weights and other reputation information [169]. Reputation systems may or may not choose to use voting in making their assessments. There has been some study of reputation systems in the context of P2P networks, but no such system has been commercially successful (e.g. [111, 167]).

1.3.3 Anonymity

There is much work that has taken place on anonymity in the context of the Internet both at the network-layer (e.g. [66]) as well as at the application-layer (e.g. [166]).

Type of Anonymity	*Difficult for Adversary to Determine:*
Author	Which users created which documents?
Server	Which nodes store a given document?
Reader	Which users access which documents?
Document	Which documents are stored at a given node?

Table 1.1: Types of Anonymity

In this section we specifically focus on application-layer anonymity in P2P systems.

While some would suggest that many users are interested in anonymity because it allows them to illegally trade copyrighted data files in an untraceable fashion, there are many legitimate reasons for supporting anonymity in a P2P system. Anonymity can enable censorship resistance, freedom of speech without the fear of persecution, and privacy protection. Malicious parties can be prevented from deterring the creation, publication, and distribution of documents. For example, a P2P system that supports anonymity may have allowed an Iraqi nuclear scientist to publish a document about the true state of Iraq's nuclear weapons program to the world without the fear that Saddam Hussein's regime could trace the document back to him or her. Users that access documents could also have their privacy protected in such a system. An FBI agent could access a company's public information resources (i.e., web pages, databases, etc.) anonymously so as not to arouse suspicion that the company may be under investigation.

There are a number of different types of anonymity that can be provided in a P2P system. Different types of anonymity make it difficult for an adversary to determine the answers to different questions. For example, "author anonymity" makes it hard for an adversary to determine which users created which documents. On the other hand, "server anonymity" makes it difficult for an adversary to determine which nodes store which documents. Table 1.1 summarizes a few types of anonymity discussed in [160].

We would ideally like to provide anonymity while maintaining other desirable search and security features such as efficiency, decentralization, and peer discovery. Unfortunately, providing various types of anonymity often conflicts with these design

goals for a P2P system.

To illustrate one of these conflicting goals, consider the natural trade-off between server anonymity and efficient search. If we are to provide server anonymity, it should be impossible to determine which nodes are responsible for storing a document. On the other hand, if we would like to be able to efficiently search for a document, we should be able to tell exactly which nodes are responsible for storing a document. A P2P system such as Free Haven that strives to provide server anonymity resorts to broadcast search, while others such as Freenet [30] provide for efficient search but do not provide for server anonymity. Freenet does, however, provide author anonymity. Nevertheless, supporting server anonymity and efficient search concurrently remains an open issue.

There exists a middle-ground: we might be able to provide some level of server anonymity by assigning pseudonyms to each server, albeit at the cost of search efficiency. If an adversary is able to determine the pseudonym for the server of a controversial document, the adversary is still unable to map the pseudonym to the publisher's true identity or location. The document can be accessed in such a way as to preserve the server's anonymity by requiring that a reader (a potential adversary) never directly communicate with a server. Instead, readers only communicate with a server through a chain of intermediate proxy nodes that forward requests from the reader to the server. The reader presents the server's pseudonym to a proxy to request communication with the server (thereby hiding a server's true identity), and never obtains a connection to the actual server for a document (thereby hiding the server's location). Reader anonymity can also be provided using a chain of intermediate proxies, as the server does not know who the actual requester of a document is, and each proxy does not know if the previous node in the chain is the actual reader or is just another proxy. Of course, in both these cases, the anonymity is provided based on the assumption that proxies do not collude. The degradation of anonymity protocols under attacks has been studied in [205], and this study suggests that further work is necessary in this area.

Free Haven and Crowds [166] are examples of systems that use forwarding proxies to provide various types of anonymity with varying strength. Each of these systems

differ in how the level of anonymity degrades as more and more potentially colluding malicious nodes take on the responsibilities of proxies. Other techniques that are commonly found in systems that provide anonymity include mix networks (e.g. [84]), and using cryptographic secret-sharing techniques to split files into many shares (e.g. [186]).

1.3.4 Access Control

Intellectual property management and digital rights management issues can be cast as access control problems. We want to restrict the accessibility of documents to only those users that have paid for that access. P2P systems currently cannot be trusted to successfully enforce copyright laws or carry out any form of such digital rights management, especially since few assumptions can be made about key management infrastructure. This has led to blatant violation of copyright laws by users of P2P systems, and has also led to lawsuits against companies that build P2P systems.

The trade-offs involved in enforcing access control in a P2P system are challenging because if a system imposes restrictions over what types of data it shares (i.e., only copy-protected content), then its utility will be limited. On the other hand, if it imposes no such restrictions, then it can be used as a platform to freely distribute any content to anyone that wants it [16].

Further effort must go into exploring whether or not it is reasonable to have the P2P protocols enforce access control, or if the enforcements should take place at individual nodes. Only users that own (or have paid for) the right to download and/or access certain files should be able to do so to legally support data sharing applications.

If the benefits of P2P systems are to be realized, we need to explore the feasibility of and the technical approaches to instrumenting them with appropriate mechanisms to allow for the management of intellectual property.

1.4 Related Security Work

In this section we provide a brief review of relevant related security research by organizing it into a framework that we call FLI.

Security architects need to address many types of security concerns in designing trusted distributed computing systems. We categorize these as concerns involving failure (F), lies (L), and infiltration (I). Hence, we call our framework FLI. As we present related security techniques, we will also discuss whether they allow a system to prevent, detect, manage, and/or recover from the FLI security problems. Keep in mind that many ideas or techniques may fall in different "regions" of our map, so there is more than one way to organize the constituent components.

1.4.1 Failure

To start with, by "failure" we mean the halt of processing of one or more system components as a result of expected or unexpected shut down or malfunction. For example, a file server may temporarily fail due to a user inadvertently kicking the power cord. A user or a node "lies" when it provides false information or pretends to be someone else. For example, two inventors may both claim that their invention was submitted first to an online patent registration system. In an infiltration, an adversary is attempting to "break-in" to the system, and use one or more resources or capabilities available to the system to their advantage. For example, an adversary may want to infiltrate a Microsoft Exchange email server to cause it to attach a virus to all outgoing emails. In this example, the server's ability to send mail to users is the capability that the adversary takes advantage of.

Note that there are relationships between these various threats. For example, infiltration can be used to cause a failure. Similarly, lying can be used to infiltrate a system.

Most tools and techniques that security designers have invented for dealing with FLI generally attempt to: 1) prevent security concerns from ever becoming a problem by designing the system to make it theoretically or practically impossible for the problem to occur, 2) detect the problem such that administrators and users can be

	Failure (Process/Storage)	Lies	Infiltration
Prevention	Physical Security Uninterruptible Power Firewalls	Authentication Authorization Access Control Non-Repudiation Time-Stamping Digital Signatures	Hardware Protection Firewalls
Detection	Watchdog Processors Polling, Beacons	Fail-Stop Digital Signatures	Virus Scanners Tripwire
Containment	RAID Non-Stop Processes Fault-Tolerance Replication, Backups	Byzantine Agreement Reputation Systems	Intrusion Tolerance Virus Cleaners
Recovery	Fail-Over Hot Swapping Key Escrow Rebooting/Restarting	Auditing	Certificate Revocation

Table 1.2: The FLI Conceptual Framework

made aware of the problem, 3) contain the problem once it does occur such that the system can continue to function correctly even in light of the problem, and 4) recover from damage that has been caused by the problem after the problem has occurred. Table 1.2 shows several tools and techniques that can be used to prevent, detect, manage, and recover from FLI.

Under the "Failure" column of the table, we have listed various tools and techniques that can be used to prevent, detect, manage, and/or recover from failure of system components. Replication, for instance, helps manage failure of a storage device by creating copies of information on other storage devices. If a particular storage device fails, either due to a head crash or an earthquake or due to an adversary that causes physical harm to it, the loss can be "managed" by retrieving the information from a storage device that replicates it.

Even if a piece of information has been lost due to failure, it is possible to recover from the problem. Assume that a user has encrypted many files with her private key and that the user stores her private key on a floppy disk. The floppy disk can fail for one reason or another, thereby making it potentially impossible to decrypt her files. We can recover from the situation by taking advantage of key escrow techniques [11, 203, 202, 110, 52, 19, 97, 187, 51, 12, 3, 98, 144] to salvage the user's private key

and any information that may have been encrypted with it.

1.4.2 Lies

Under the "Lies" column of the table, we list various security techniques that can be used to deal with both users and adversaries that attempt to "lie" to a system. Consider the concept of digital signatures [47, 161, 171, 125, 162, 45, 131, 46, 127] which can prevent users from lying about statements that other users make. A system that employs digital signatures may require users to digitally sign statements, thus preventing tampering with those statements. For example, assume that Alice wants to say "here is the answer to your query" to Bob, but can only communicate with Bob through Eve. Alice sends a message containing the answer and her digital signature on the message to Eve. If Eve attempts to modify Alice's message, Bob will notice that the digital signature does not match the message. Hence, digital signatures prevent Eve from lying about what Alice said.

If we are unable to prevent lies, we can design the system so it can tolerate lies without affecting the correctness. To "contain" lies, a Byzantine agreement protocol [103, 71, 69, 62, 105, 24, 25, 58, 154, 104, 117, 68, 132, 77] can be used where even if Eve modifies a message that Alice wants to pass on to Bob, all the people involved in the decision can come to agreement, say on the number of replicas to make for a document, so long as the number of people involved in the decision is greater than three times the number of liars, and all participants can communicate with one another without restriction.

1.4.3 Infiltration

Tools such as firewalls [218, 29, 163, 149, 14, 100] and certificate revocation [80, 127, 141, 99, 128, 206, 126] techniques help a security designer defend against attempts at system infiltration. Firewalls are composed of one or more system components that prevent an adversary from "breaking in" to computers within the perimeter of the firewall, such that data and computational resources available within the perimeter of the firewall cannot be used to an adversary's advantage. (Firewalls can also be used

to prevent failure due to DoS attacks. For example, ICMP-based smurf attacks can be prevented by configuring the firewall to drop ICMP ping requests sent to broadcast network addresses.)

Once a system has been infiltrated, a technique such as certificate revocation may help us recover from the infiltration. If an adversary gets access to and compromises a user's private key after a system infiltration, certificate revocation allows us to break the association between that user and his or her public key, such that the private key becomes ineffective.

1.4.4 Other Techniques

Additional relevant security techniques such as auditing [22, 28, 156, 143, 18, 17, 81, 182, 90, 87, 48, 8], intrusion detection [50, 204, 106, 113, 180, 138, 108, 49, 107, 109, 101, 114, 157, 158, 185, 211], non-repudiation [217, 179, 213, 32, 214, 123, 215, 216, 197, 122], and others have also been added to the appropriate cells of Table 1.2 to show that a wide variety of techniques fit well within the context of the FLI framework. At the same time, we realize that the cells in Table 1.2 do not have absolutely precise boundaries, and it is sometimes arguable as to whether or not a given security problem or solution fits into one cell or another, or may even span across multiple cells. Yet, we find the FLI framework useful to conceptually organize the space of problems and solutions.

1.4.5 P2P and FLI

A P2P system presents special challenges in some of the FLI regions. For example, many P2P networks wish to support anonymity. Anonymity can, for example, help prevent attacks on a particular node. However, a potential downside of anonimity is that it can enable lying: if we could hold nodes accountable for their lies, the network would not really be providing anonymity.

Thus, the *containment* of lies region is more critical in such systems. Indeed, as we will see, much of the content of this dissertation seeks to address the containment of lies in P2P systems. In particular, malicious nodes in a P2P system may generate

"useless" queries (that are indistinguisible from bona fide queries issued by legitimate nodes) with the goal of conducting a DoS attack. These "useless" queries use resources that would otherwise have been used to service queries issued by legitimate nodes. With respect to our FLI framework, malicious nodes are implicitly "lying" about the legitimacy of their queries, and the techniques that we develop in the body of this dissertation address the containment of lies in our FLI framework.

Moreover, due to the level of anonymity provided by some P2P networks, it will be hard to both detect and prevent failure due to DoS attacks by attempting to identify the source of the attack or filter out requests coming from that source. In addition, without a central authority, it becomes harder to protect against lies, or to reconfigure a system after a failure.

Most research that has been done in the area of P2P systems has been in the areas of efficient search, routing, and indexing. Beyond what systems like Gnutella and Morpheus do, systems such as Chord [193], CAN [164], Pastry [176], Tapestry [212], and Viceroy [38] can provide guarantees on the maximum number of nodes that need to be queried in order to find an answer to the query if an answer exists in the network. These guarantees are usually provided at the expense of reduced autonomy; restrictions on how nodes must connect or where documents must be stored are imposed to enforce guarantees.

Much of the existing security-related research that has taken place in the P2P area has focused on providing anonymity to users of the system, and ensuring fair resource allocation. Free Haven [57, 174, 55], Freenet, and to some extent Gnutella provide "reader" anonymity in that they are designed to prevent a third-party from determining which node a query originated at. Some of these systems provide other types of anonymity as well. Publius [121], Freenet, and Free Haven, for example, prevent censorhip by providing "publisher" anonymity. Systems such as Mojo Nation [134] and Reputation Server [111, 168] are designed to allocate resources fairly. Mojo Nation does this by issuing a form of digital cash called "Mojo" that must be "spent" to issue queries and to publish documents. (This also enables Mojo Nation to deal with some types of DoS attacks.) Reputation Server keeps track of user's "reputation," such that users are given the incentive not to abuse the system else

their reputations will be downgraded, thereby affecting the future utility that they will be able to derive from using the system.

There are many open security problems in P2P systems. The key to securing a P2P network lies in designing mechanisms that ensure availability, file authenticity, anonymity, and access control. In this introduction, we have illustrated some of the trade-offs at the heart of security problems in P2P systems, and outlined several major areas of importance for future work.

1.5 Electronic Commerce

After exploring application-layer DoS issues in P2P networks in Chapters 2 through 5 of this dissertation, we turn our attention to how to support various forms of electronic commerce (e-commerce) in P2P networks in Chapters 6 and 7. Clearly, if application-layer DoS issues are not first addressed, then P2P networks may not be an attractive platform for e-commerce. However, once we are able to mitigate DoS attacks in P2P networks, it is worthwhile to then investigate how e-commerce can be supported by P2P networks.

E-commerce can be described as the exchange of goods and services between buyers and sellers with the aid of computers and computer networks. E-commerce typically takes place between two parties at a time, and either occurs between a business and a consumer (B2C), between two businesses (B2B), or between two consumers (C2C). Commerce transactions that occur between two consumers are very often also referred to as person-to-person, or P2P, commerce. However, to avoid overloading the acronym P2P, which refers to peer-to-peer in this dissertation, we will refer to person-to-person commerce as C2C commerce.

We define P2P commerce as an exchange of goods or services that takes place between any two peer computers on a network, in which peers can sometime function as buyers and sometime function as sellers. The peer computers may be owned by consumers or businesses, and P2P commerce does not necessarily infer e-commerce between two consumers.

E-commerce is a deep field in itself, and we only provide a brief overview of B2B,

B2C, and C2C e-commerce with the intention of placing our contributions in the area of P2P commerce in context.

B2B commerce traditionally has taken place between proprietary computer systems owned by businesses using a protocol such as EDI (Electronic Document Interchange) [207]. Common business documents such as invoices and purchase orders are exchanged between businesses, and the corresponding monetary transfers occur over out-of-band financial networks, such as the Automated Clearing House (ACH) [140] network. Recently, open B2B exchanges that take advantage of XML [200] documents as the new format of document interchange have been launched by companies such as Ariba and CommerceOne.

Most B2C commerce to date takes place via a consumer's web browser and a business' web server using the SSL (Secure Sockets Layer) [67] / TLS (Transport Layer Security) [53] protocols. After the customer's web browser authenticates the web server by challenging it to provide a digital signature attesting its domain name, the customer sends a credit card number (and related information such as the expiration date and billing zip code) to the business using the popular HTTP [64] protocol over a secure SSL channel. The business submits the credit card number to a separate, proprietary financial network to authenticate its validity, and upon receiving a positive authorization, provides the product or service to the customer. Credit cards are typically well-suited for payments above some threshold dollar amount due to the processing overheads incurred by credit card issuers and banks. Hence, while credit cards might be suitable for purchasing books online that cost at least a few dollars each, they might not be suitable for purchasing small, granular items such as access to an individual web page.

Various micropayment schemes have been suggested that would allow for efficient purchases of small amounts [172, 118, 129, 88, 23, 35, 37, 54] between clients and servers. Some related work has also taken place on how to incorporate micropayment schemes into P2P networks. For instance, Mojo Nation [133] was a P2P system in which peers issued micropayments using a digital currency called "mojo" to conduct searches, and to receive payment for storing documents. Yang and Garcia-Molina

in [209] develop and evaluate a P2P micropayment protocol called PPay that eliminates the need for a centralized broker.

C2C commerce occurs when one consumer would like to purchase a product or service from another consumer. C2C commerce usually involves the consumer interacting with each other through electronic mail, and an online broker. C2C commerce is often the method of choice for payment when a consumer wants to pay another consumer for a good or service that was auctioned at an online auction site. For example, after one consumer (the buyer) has successfully bid for a product being sold by another consumer (the seller) on eBay [61], the seller sends an email to the buyer to request payment. The email contains a link to the web site of an online broker such as PayPal [152]. The online broker contains the financial profiles of both the buyer and seller including bank account and credit card information. The seller instructs the online broker to issue payment via credit card or directly from the seller's bank account, and the online broker transfers the requested funds into the bank account of the seller. The seller receives an email when the transfer is complete, and then ships the product purchased to the buyer.

In this dissertation, we explore how B2B, B2C, and C2C commerce can be supported by P2P networks. Nodes in a P2P network may be owned and operated by both businesses and consumers, and while traditional e-commerce has occured between nodes that take on rigid buyer (client) or seller (server) roles, we explore how traditional e-commerce can be complemented by payments that can occur between more flexible peers that can take on different roles at different times in a P2P network.

1.6 Thesis Statement and Outline

We now state our thesis, and outline how the chapters of this dissertation support our thesis statement.

In this dissertation, we put forth that application-layer denial-of-service attacks in peer-to-peer networks can be contained by taking advantage of load balancing techniques. In addition, we propose that digital wallets and digital cash protocols can be used to enable commerce between peers in a P2P network.

To put our thesis in perspective, we develop solutions required to ensure availability in a P2P network while it is under attack. We seek to contain such attacks using load balancing techniques described in Chapters 2, 3, 4, and 5. We also outline an architecture for commerce in P2P networks that can help address the problem of access control by allowing users to pay for content using a peer-to-peer approach.

This dissertation is divided into two parts. In the first part, we specifically focus on exploring attacks against availability in P2P networks, and how to contain them. In the chapters that make up the first part of this dissertation, we study application-layer attacks against each of the three major types of P2P networks: unstructured, structured, and non-forwarding. In the second part of this dissertation, we propose a software architecture for peer-to-peer commerce, and report on a prototypical implementation of it.

The first part of this dissertation is organized as follows:

- Chapter 2 motivates the problem of application-layer denial-of-service in unstructured (Gnutella) P2P networks, develops a model and metrics that can be used to analyze traffic management policies that may address the problem, and presents the results of evaluations of some basic traffic management policies run on small representative network topologies with a single malicious node.

- Chapter 3 extends the model in Chapter 2 to incorporate malicious nodes that are significantly more powerful, develops the least-queries-first, probabilistic acceptance, and TTL-shaping techniques to mitigate attacks, evaluates these techniques on real snapshots of Gnutella networks, and discusses the trade-offs between the proposed techniques.

- Chapter 4 extends the model developed in Chapters 2 and 3 to study the problem of application-layer DoS in structured P2P networks. Some basic techniques are proposed to deal with the problem, and these techniques are evaluated on Chord DHTs [193].

- Chapter 5 studies application-layer DoS in a non-forwarding P2P network called

GUESS, and identifies a cache poisoning attack that can be used as a precursor to a successful DoS attack. An ID smearing algorithm and dynamic network partitioning scheme are developed to address cache-poisoning-based DoS attacks, and these techniques are evaluated.

The susceptiblity of a P2P network to DoS attacks could preclude e-commerce from taking place at all. Various techniques can be used to prevent, detect, contain, and recover from such attacks. While we focus on studying attack containment techniques in the first part of this dissertation, we expect that prevention, detection, and recovery techniques developed by others will also need to be employed.

After studying the containment of DoS attacks in P2P networks, we shift gears in the second part of this dissertation to study how we might support e-commerce in P2P networks. In particular, we propose a commerce architecture for P2P networks based on digital wallets, and describe an example implementation of a digital cash system based on that architecture. The remaining two chapters are organized as follows:

- Chapter 6 proposes a peer-to-peer based commerce architecture that takes advantage of the symmetry of peers in a P2P network, allows for support of new payment protocols as they are developed, takes into account that peers may be PDAs or other types of mobile devices in addition to more "standard" clients, and proposes that the control of initiating purchase transactions be kept with clients.

- Chapter 7 describes a prototypical implementation of the architecture in the previous chapter to allow peers to efficiently make payments on PDAs. The implemenation uses a digital cash protocol that takes advantage of a hybrid of two public-key cryptosystems to satisfy the performance and memory constraints of PDAs and other "lightweight" peers.

Chapter 2

DoS In Gnutella

In the next four chapters we study application-layer denial-of-service (DoS) attacks in peer-to-peer (P2P) systems, and propose simple techniques to cope with such attacks. In this chapter and the next, we chose to study Gnutella P2P systems, because they are very vulnerable to attack and are very popular. Gnutella systems are one of the most prevalent P2P systems today, with over 25 million client downloads, and 300,000 concurrent users during peak periods. In Chapters 4 and 5, we consider other types of P2P networks.

A Gnutella network is made up of a number of regular nodes and supernodes. The supernodes are typically connected to the Internet via high-bandwidth links, and can typically process an order of magnitude more queries in a given unit of time than regular nodes can. Supernodes form the "core" of the network and are responsible for processing and routing queries. Regular nodes send their queries to supernodes.

In Gnutella (as well as in Morpheus [137] and KaZAA [93]), a client submits a query (e.g., looking for a file) to a supernode (server). That supernode performs a local search, and also broadcasts the query to its neighboring supernodes. Thus, a large number of queries from a malicious node or supernode can exponentially multiply throughout the system, consuming resources that other clients cannot use.

In this dissertation, we advocate application-layer "load balancing" techniques that attempt to give nodes a "fair share" of available resources, thus making it harder for malicious nodes to deny service. We also occasionally refer to our load balancing

techniques as "query acceptance" techniques, as they help nodes decide which queries to accept and which to drop. Most DoS work to date does not fall in this category: current techniques tend to be either recovery-oriented, in which in-progress attacks are detected and service is denied to offending clients, or preventative, in which security mechanisms prevent clients from gaining access to resources [94]. In contrast, *we do not require servers to distinguish attack queries from bona fide ones*, and indeed, malicious clients will be able to receive some service. However, the load balancing policies try to make sure that offending clients do not receive an inordinate amount of service. Of course, in a Gnutella P2P system the challenge is to maintain a fair load distribution in spite of the multiplicative effect of query broadcast.

Clearly, load balancing policies do not eliminate the need for preventative and recovery-oriented techniques. We believe that all three types will be needed for protection against DoS attacks. In this dissertation we focus on the load balancing techniques because we feel they have not been studied adequately in this context. Because we are studying load balancing techniques, in our evaluations we focus on flooding-type DoS attacks, as opposed to attacks that are better dealt with by other techniques. (For example, if a single malicious query can crash a node, then we clearly need to ensure, using a preventative approach, that such a query is never executed.)

We also note that some of the load balancing techniques we advocate are not new. These types of techniques have been used for many years in network management, processor scheduling, and many other applications. Here we are simply applying these techniques to a P2P environment, and extending them to handle requests originating via flooding from a malicious node that may be multiple hops away.

One area in which we have had to go beyond the current state of the art is in the *evaluation* of DoS load balancing techniques. In particular, we needed and developed techniques for modeling and quantifying the "usefulness" of load balancing DoS techniques. With our model and metrics we can compute how much "damage" a malicious node may cause, which network topologies may be more vulnerable to attacks (allowing greater damage), which nodes within a network are the most vulnerable, and which load balancing techniques may be most effective. We believe that such an evaluation is essential in order to fully understand flooding-based DoS attacks.

An attacker could conduct a network-layer DoS attack by sending out enough network packets to exhaust the network bandwidth of a number of supernodes, and effectively shut down those supernodes. However, a much more powerful attack would involve sending fewer packets containing application-layer queries that will be broadcast to an order of magnitude more supernodes. Such an application-layer attack can waste a significant fraction of the total aggregate processing capacity of *all of the supernodes on the network*, and will be more effective from the standpoint of an attacker that wants to maliciously impact the entire network instead of just a few supernodes.

Compared to network-layer DoS attacks, little is known about application-layer DoS attacks. Even if a network is immune to a network-layer DoS attack, it will still be susceptible to an application-layer attack because the number of query packets that the attacker needs to inject need not approach the network bandwidth available– the queries contained in the packets consume a significant number of CPU cycles and disk I/Os that would have otherwise been used to service legitimate queries.

In this chapter, some of the challenges that we start to address are developing a traffic model that captures the essential details of the problem, outlining a few options for some of the key parameters in the model, and defining metrics that will help us assess which options perform the best.

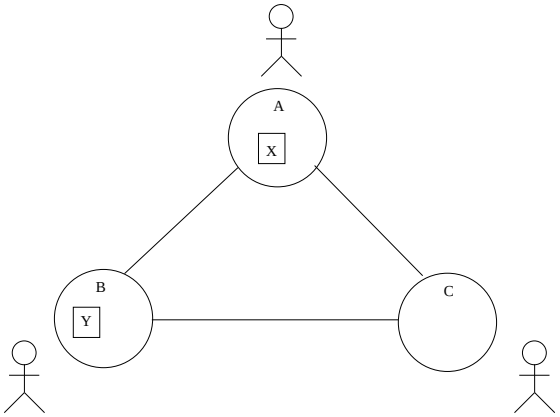

Figure 2.1: An example of three node network

To further motivate the problem that we are solving, consider the oversimplified

network of three nodes as shown in Figure 2.1. Each of the nodes is represented by a circle, and stores data items (depicted by squares) in a database. Each node can process up to C queries per day, and has a single user. Users can enter queries at their own node, and can get search results from their own node. For instance, if the user of node A issues a query for data item X, the user will receive a search result from node A itself. However, nodes may also send queries to other nodes such that users can benefit from search results that may be available elsewhere. For example, if the user of node A issues a query for data item Y, node A can forward the user's query to node B, as node A would not be able to answer the query itself. (Node B would, of course, forward the result back to node A.)

Assume that each user has $\frac{1}{3}C$ queries per day, and when a user issues a query at her node, the query is simultaneously forwarded to the other two nodes in the network. In such a situation, each node is processing the maximum number of queries that it can ($\frac{1}{3}C$ from its own user, and $\frac{2}{3}C$ queries from the users at the other two nodes).

Consider what happens if one of the nodes, say node B, is malicious. Node B might be able to carry out a variety of attacks, but here we are interested in one of the simplest, easiest-to-conduct denial-of-service attacks that node B can conduct. Node B might simply issue more than $\frac{1}{3}C$ queries, and forward them to the other nodes with the intention of wasting query processing resources at nodes A and C. Legitimate queries could be denied those processing resources. If node B issues more than $\frac{1}{3}C$ queries, nodes A and C will receive more queries than they can handle from B, and may have to drop some legitimate queries to handle the increased load.

In a real network, node B may also be forwarding queries on behalf of other nodes (not shown in the figure), and nodes A and C may not be able to distinguish whether the increased load originates at B or whether it is simply forwarding queries on behalf of other nodes. In addition, because nodes in a Gnutella network broadcast queries to all their neighbors, a single useless query can be replicated many times and deny service to many legitimate queries. One "solution" might be to avoid a flooding-based protocol, but flooding-based protocols are simple to build, easy to deploy, obtain search results very quickly, are highly resilient to nodes joining and leaving the system, and have been successfully used in many real systems. Furthermore, in

systems such as wireless ad-hoc networks, flooding may be the only option on top of which to build higher-level protocols, and we may not have the option of simply choosing a non-flooding-based protocol.

In our example, to deal with the extra load, nodes A and C will be forced to select some of the queries being sent to them for processing, and will drop others. That is, nodes A and C need to use a *query selection policy* to decide which queries to process, and which queries to drop in an overload situation. Since nodes A and C may not know whether or not any of the other nodes they are connected to are malicious, they may, for instance, decide to "fractionally divide" their capacity. That is, they could decide to process at most $\frac{C}{3}$ queries from each of their links (and reserve $\frac{C}{3}$ capacity for their own user's queries). Doing so may reduce the number of malicious queries they accept, but could also reduce the number of legitimate queries they accept from other nodes.

In this chapter, we study the option of fractionally dividing capacity, as well as several other methods that nodes may use to select queries for processing. The query selection policy that nodes use may have a significant impact on the success that a malicious node may have in conducting such a denial-of-service attack in which it attempts to flood the network with "useless" queries. A query selection policy may also have an impact on the performance of the network in terms of the number of legitimate queries that are answered. It will be important to develop, identify, and choose query selection policies that minimize the number of malicious queries processed, while at the same time maximize performance.

The types of query selection policies that a node might employ will differ depending upon whether or not it is believed that extra load is due to increased user queries somewhere in the network, or that malicious nodes are attempting to conduct a DoS attack. In the former case, it may be useful for nodes to execute queries that make up the extra load because it may mean that some legitimate user simply has "bursts" of queries that need to be answered to support some application. However, in the latter case, the queries that make up the extra load could be ignored in an attempt to prevent capacity from being used up by useless queries. It may be difficult to distinguish the two cases, and our goal in this chapter is to develop and use query selection

techniques that attempt to balance load with the bias that highly unbalanced load may be the result of malicious nodes.

A malicious node may use various *flooding strategies* to create overload situations. For example, it could decide to use all of its processing resources to forward useless queries, or could tailor its attack against particular query selection policies that good nodes might use. Also, although the three-node network in our example is perfectly symmetric, a malicious node might choose to take on different locations in a real network to deny good nodes of query processing resources. In this chapter, we study representative topologies in which a single malicious node uses all of its processing capacity to forward useless queries. In the next chapter, we study larger topologies in which many malicious nodes use different flooding strategies to attack the network.

Our approach to mitigate application-layer flooding attacks is to have supernodes employ various traffic management policies to "fairly" balance the query load to minimize the effect that one or more malicious nodes can have on the network. We do not require supernodes to distinguish attack queries from bonafide ones. Detection techniques that attempt to discern attack queries from bonafide ones may be overly challenging to devise correctly and may suffer from a high rate of false positives or negatives. In addition, such detection techniques may also be computationally expensive.

Our main contributions in this chapter are as follows:

- We define a simple but effective traffic model for query flow in Gnutella networks, and outline policies that nodes may use to manage query flow. We define expected behaviors for "good" and "malicious" nodes, and metrics that we use to evaulate the impact that malicious nodes have by flooding the network. (Sections 2.1, 2.1.2, and 2.2)
- We evaluate how network topology affects the ability of a malicious node to flood the network. In our evaluations, we study the vulnerability of complete, cycle, wheel, line, star, grid, and power-law topologies under various flow management policies. (Sections 2.3.1, 2.3.2, and 2.3.3)
- We evaluate how different combinations of flow management policies can be used

to manage the distribution of damage across the nodes in a network. (Section 2.3.4)

2.1 Gnutella Traffic Model

In this section, we define a simple, discrete-event-based traffic model for Gnutella networks that models query flow and query load in the nodes of the network.

The model that we present is an intentionally coarse-grained and relatively simple model whose goal is to capture the important features of query traffic [1]. We do not expect the model to predict actual query loads (as might be observed in a real network), but we do expect it to tell us about relative query loads at nodes of the network by using different application-layer policies that we will consider in Section 2.1.2.

As mentioned earlier, a Gnutella network is made up of two types of nodes: regular nodes and supernodes. Supernodes have access to more network bandwidth and CPU cycles, and are able to do more disk I/Os per unit time than regular nodes. In a typical P2P network, supernodes are able to process a few orders of magnitude more queries per unit time than regular nodes. As such, we only explicitly account for supernodes in our model. We assume that regular nodes connect to supernodes, and they submit their queries to supernodes to have them processed. The set of regular nodes that are connected to a supernode are called its *local peers*, and the set of supernodes that are connected to a supernode are called its *remote peers*. We assume that each supernode v in the network is able to process C_v queries per time unit, and we say that C_v is the supernode's *query bandwidth*. When a supernode processes a query, we say that the supernode has executed one unit of *work*. If the query was sent to it by a local peer, we say that one unit of *local work* has been done, or if the query was sent to it by a remote peer, we say that one unit of *remote work* has been done.

The topology of the network is modeled as a graph (V, E) where V is the set of supernodes in the topology, and E is the set of bidirectional links between the

[1] While the Gnutella protocol does include messages other than queries and query-hits, we only consider queries in our traffic model, as processing queries takes significantly more network, disk I/O, and CPU resources than processing other messages in the protocol.

supernodes of the graph. Let $N = |V|$ be the total number of nodes in the network. We let G denote the set of good supernodes that, in general, follow the protocols we describe, and M denote the set of malicious supernodes, which may or may not follow the protocols. Of course, at any instant $G \cup M = V$ and $G \cap M = \emptyset$. Also, if there exists an edge $(u, v) \in E$, supernodes u and v are informally said to be "neighbors."

Time is divided into a number of discrete- time intervals, $t = 0, 1, 2....$ Queries may be sent from one supernode in the graph to another supernode in the graph within one unit of time only over one of the links in E. For example, if during the first time interval $t = 1$, a supernode $v_1 \in V$ sends a query to another supernode $v_2 \in V$, then the query arrives at v_2 at time $t = 2$. As we only explicitly model supernodes, the term node will be used to refer to a supernode in the remainder of this chapter, unless we state otherwise.

In a real Gnutella network, nodes join and leave the system over time. For the purposes of our study, and determining which traffic management policies are most successful at mitigating query floods, it is sufficient to model a static network. Since the traffic management policies that we consider in Section 2.1.2 do not depend on time, the results of which traffic management policies perform the best in our experiments in Section 3.2 will not be different for networks whose topologies vary dynamically.

During each discrete time interval, four actions take place at each node.

1. *Query Admission / Injection.* In this work, we are interested in studying how the network performs when it is under stress. Certainly, the effects of a query-flood DoS attack will be most significant when the network is already under a maximal load, and all of its processing capacity is expended doing useful work. Under such a scenario, each useless query that is injected into the network by a malicious node will end up displacing or denying a good, useful query the opportunity of being processed. As such, we assume that all nodes have an infinite number of queries that need to be processed, and good nodes need to decide how many queries to *admit* into the system in a particular time interval to keep the network operating at maximum throughput. While admitting a query and and injecting a query mean the same thing, in the remainder of this

dissertation, we will use the term *admit* to refer to a good node submitting queries into the system, and *inject* to refer to a malicious node submitting a query into the system.

Therefore, the first action that takes place at a node in a particular time interval is deciding how many queries it will *admit* or *inject* into the network for processing.

2. *Receive and Decide on Queries.* Each node receives queries from its local peers and from its remote peers. A node may receive more queries from its local and remote peers than it can process in a given time interval. Hence, each node needs to apply traffic management policies to inspect all the queries that it receives, and decide which queries to process and which queries to drop based on traffic management policies.

3. *Process Queries.* For each query that a node has decided to process, it must look for matches for that query against its local repository or database of files.

4. *Forward Queries.* For each of the queries that the node has decided to process, it may forward the query to its neighbors.

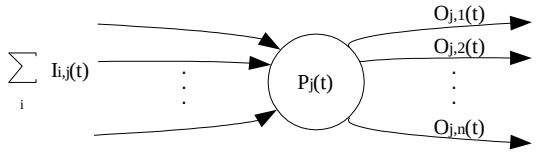

Figure 2.2: DoS Model

We now describe our model more formally. We will use the following definitions later in the chapter to describe our query processing strategies and performance metrics.

We model a query as a four-tuple $q = (id, o, t, k)$ where:

- id is a 16-byte query descriptor that is unique amongst all queries in the network (with very high probability)

- o is the node at which the query originated[2].
- t is the query's current Time-To-Live (whose function will be described shortly), and
- k is a list of keywords.

We will refer to each of the components that make up a query as $q.id$, $q.o$, $q.ttl$, and $q.k$ to refer to the query's id, origin, current TTL, and keywords.

Let $O_{u,v}(t)$ be the multi-set of queries that node u sends to node v at time t. Let $L_v(t)$ be the multi-set of queries that node v receives from local peers at time t. Node v may not be able to process all of the queries it receives. Node v examines the incoming queries contained in $O_{u,v}(t-1)$, $\forall u$ where $(u,v) \in V$, as well as those in $L_v(t-1)$. (Of course, if $(u,v) \notin E$, then $O_{u,v}(t-1) = 0$). At time t, v may have to choose some subset of the queries in the set $\bigcup_{u \epsilon V} O_{u,v}(t-1) \cup L_v(t-1)$ for processing. Let $I_{u,v}(t)$ be the multi-set of queries that node v actually processes from node u at time t. ($I_{u,v}(t) \subseteq O_{u,v}(t-1)$.) Similarly, let $P_v(t)$ be the multi-set of local queries that v actually processes. ($P_v(t) \subseteq L_v(t-1)$.) During each time step, a node selects at most C_v queries for actual processing. In other words, nodes have the following *capacity constraint*: $\sum_{\forall u} |I_{u,v}(t)| + |P_v(t)| \leq C_v$. A diagram depicting the query flows processed is shown in Figure 2.2. For the purposes of our study, we assume that processing queries dominates the computation that takes place at nodes. We also assume that all queries can be processed within one discrete time interval. After processing these queries, node v broadcasts the processed queries to all of the nodes to which it is connected: $O_{v,w}(t) = \bigcup_u I_{u,v}(t) \cup P_v(t)$ where $(v,w) \in E$ and $u \neq w$. (The broadcast is done with the exception that queries are not sent back along the paths from which they arrived.)

As we have seen, nodes in the Gnutella network rely on a flooding-based protocol to propagate queries through the network. To keep queries from propagating through the network endlessly, we describe two basic traffic management mechanisms that

[2]Current Gnutella networks do not stamp queries with the nodes at which they originated, but this feature could be added. However, we do not assume that the integrity of the source id is protected by an authentication mechanism.

are specified as part of the Gnutella protocol: TTL stamping and duplicate query elimination.

We first describe how TTL stamping takes place. Each query is given a TTL (time-to-live) by the node that originally injected the query into the network. We assume that all peers in the network inject queries into the network with the same TTL, τ. After a node processes a query, it decrements the TTL of the query. If the TTL of the query is non-negative, the query is broadcast to all of its remote peers. The second mechanism, duplicate elimination, assigns a unique GUID (Global Unique IDentifier) to each query. (The GUID is typically the output of a hash function, and is only globally unique with high probability.) Each node records the GUIDs of queries that it has recently seen and never selects queries that it has seen recently to be processed a second time. (Duplicate queries may arrive at a node due to cycles of length less than τ in the topology.)

2.1.1 Reservation Ratio (ρ)

Nodes must provide some level of fairness between servicing local and remote queries. If supernodes only service local queries, then local peers will not benefit from query results that could be obtained from other supernodes in the network. If supernodes service only remote queries, then local peers will be neglected.

To allow a supernode to decide how to split its processing capacity we define ρ to be the fixed fraction of query bandwidth that a supernode reserves to service local queries ($0 \leq \rho \leq 1$). A supernode v uses ρ to determine how many queries to accept from local peers and how many queries to accept from remote peers when the total number of queries sent to it exceeds C_v. (If the total number of queries sent to it does not exceed C_v, the supernode will simply process all the queries sent to it.)

While current Gnutella clients do not support the ability to use a reservation ratio (ρ) to divide up their available query bandwidth, we will be interested in studying how different policies that use ρ might be used to maximize the number of useful (non-malicious) queries processed by network, ensure fairness in servicing queries, and manage floods of attack traffic.

Formally, if $|G_j(t-1)| \leq \rho C_v$, then $P_j(t) \leftarrow G_j(t-1)$. Otherwise, we select ρC_v queries from $G_j(t-1)$. Queries that are not selected from $G_j(t-1)$ for processing at time t can be queued for future time steps (or even discarded). The remaining capacity, $C_v - |P_j(t)|$ is now allocated among the queries arriving from remote peers. We will refer to ρC_v as node v's local query bandwidth (LQB), and $(1-\rho)C_v$ as node v's remote query bandwidth (RQB).

There are several combinations of policies that we shall consider for allocating the RQB among queries arriving from remote peers. There are two questions that these policies answer:

1) how many queries should be accepted from each remote peer?, and

2) if there are more queries arriving from a remote peer than we decide to accept, which ones should we accept?

The policy used to answer the first question is the *incoming allocation strategy*, and the policy used to answer the second question is the *drop strategy*.

2.1.2 Query Selection Policies

Deciding which queries to drop and which queries to process is the most critical decision that a node needs to make to mitigate a query-flood DoS attack. Certainly, if a node processes too many queries that were issued by a malicious node, then the node will be highly affected by the attack since it is not doing "useful" work by processing queries that were issued by legitimate, good nodes. Not only will such a node be wasting its processing capacity, but it may then (inadvertently) forward those queries onto other nodes. Should those nodes decide to process the query, additional processing resources in the network will have been wasted. Due to the multiplicative effects of flooding, a linear increase in the number of malicious queries can cause an exponential waste in processing resources throughout the system.

In this chapter, we study various *query selection policies* that seek to reduce the multiplicative effect of query-floods. Query selection policies manage the floods after malicious nodes have injected the queries into the network. Since we make the assumption that a good query is indistinguishable from a malicious query, the highest

level of success that can be acheived occurs when a linear increase in the number of malicious queries issued results in no more than a linear drop in performance. Some of the query selection policies that we evaluate in Section 3.2 acheive this goal while others do not.

During each time interval of our model, TTL stamping is applied before query selection policies, and duplicate query elimination is done after query selection policies are applied.

We divide a node's query selection process into two parts. First, a node assigns a "link quota" to each of its remote peers that serves as a maximum for how many queries it will accept from each of them. Then, if particular remote peers send more queries than allowed by their link quota, the node must decide which queries to actually process and which queries to drop to meet the link quotas The policy that is used to decide link quotas is called the *incoming allocation strategy (IAS)*, and the policy that is used to allocate which queries to drop from remote peers that exceed their quota is called the *drop strategy (DS)*.

More precisely, an IAS is a function that takes as input a vector of sets $< O_{u_1,v}(t-1), O_{u_2,v}(t-1), ..., O_{u_k,v}(t-1) >$ and returns a vector of integers $< q_1, q_2, ..., q_k >$, $0 \leq q_i \leq |O_{u_i,v}(t-1)|$, specifying the maximum number of queries node v can accept from node u_i. A DS is a function that takes a set (such as $O_{u_i,v}(t-1)$) and an integer (such as q_i) and returns a set $I_{u_i,v}(t)$ specifying exactly which queries from its remote peers node v should process. We consider various IAS and DSes in the next section, and illustrate how they work with a running example.

We make the assumption that a node has enough processing capacity to at least examine all of the queries arriving on all of its links to make its decisions using IAS and DS policies. The time required to process a query typicially involves searching for a keyword in a hash table, inverted index, or some other main memory data structure. However, keyword search for content has its limitations, and over time more sophisticated search mechanisms will be employed. For instance, some Gnutella clients use other metadata in addition to keywords as part of searches [112]. As the search mechanisms become more complex, query processing will dominate the time required to service incoming queries. In turn, the time to examine all incoming

queries will become negligible compared to the time required to process queries that are chosen to be serviced.

2.1.3 Incoming Allocation Strategy

In this section we define how varioius IASes work in general, and we also illustrate them with a specific example. For a particular node v, let p_i be the number of queries arriving on v's ith edge. That is, $p_i = |O_{u_i,v}(t-1)|, 1 \leq i \leq d(v)$, where $d(v)$ is the degree of v. For example, consider a node A that has two edges to nodes that are sending it queries. Node A has a link to node B that is sending it 16 queries, and also has a link to node C that is sending it 4 queries. We say that $p_1 = 16$, $p_2 = 4$, and $d(A) = 2$. Also, for the purposes of this example, we assume that node A has a RQB of 10 ($(1-\rho)C_A = 10$), and it must decide what quotas to set for each of its links.

There are two key incoming allocation strategies (IASes) that we cover in this chapter: weighted and fractional. In this section, we describe how each of these strategies selects which adjacent nodes to process queries from. As mentioned in Section 2.1, we refer to the components of a query using a dot notation such that $q.o$ refers to the origin node (at which the query was first admitted or injected) and $q.t$ refers to the TTL. In the policies that we describe, the origin and TTL of queries will be of interest, while the ids and keywords of the queries will not. Many distinct queries can have the same origin and TTL, and we will refer to η distinct queries with the same origin and TTL as ηq.

For a given query q, the node at which the query originated is $q.o$, the time at which it originated is $q.t$, its time-to-live is $q.t$. Depending on the P2P protocols in use, a query may or may not carry all the information with it. Furthermore, the information carried by q may be false, having been modified by some rogue node that handled it earlier.

Weighted IAS. We use the weighted IAS to approximate the case in which nodes simply choose to service queries on a first come first serve (FCFS) basis. Weighted IAS is intended to model a "naive" Gnutella node in which the likelihood that a query from a particular incoming link will be accepted is proportional to the number

of queries arriving on that link. We assume that all queries arriving at a particular node j from remote peers arrive according to a uniform random distribution. Hence, a remote peer that is sending j many queries per time step will have more of its queries arrive at j than those of a remote peer that is sending j just a few queries per time step. As such, weighted IAS gives more weight to processing queries from remote peers that send more queries, and less weight to processing queries from remote peers that send less queries.

We assume that queries arriving at node j from remote peers are equally likely to be accepted. Thus, the more queries a neighbor sends, the more will get accepted.

If fewer than $(1-\rho)C_v$ queries are sent by remote peers, then all queries that are sent are accepted for processing. If more than $(1-\rho)C_v$ queries are sent by remote peers, then the number of queries accepted from each remote peer is weighted by the fraction of the total queries sent. If a node has κ remote peers that send it $\alpha_1, \alpha_2, ...\alpha_\kappa$ queries, it will accept up to $\frac{\alpha_\lambda}{\Sigma_{\forall i}\alpha_i}(1-\rho)C_v$ queries[3] from the λth remote peer, $1 \leq \lambda \leq \kappa$. On average, this models a node choosing C_v queries from $\bigcup_{i\epsilon V} I_{i,j}$ using a uniform random distribution.

For instance, in our example, the weighted IAS divides the RQB such that $|I_{B,A}(t)| = \frac{16}{20}(1-0.2)100 = 67$ and $|I_{C,A}(t)| = \frac{20}{120}(1-0.2)100 = 13$.

Fractional-Spillover IAS ($Frac-Spill-IAS$). Fractional-Spillover IAS attempts to give each of a node's incoming links an equal fraction of query bandwidth. If a node has κ remote peers, a *fractional* IAS allocates up to $1/\kappa$ of its query bandwidth for handling the queries from each of its remote peers. Any extra query bandwidth that is unused by a remote peer sending less that $1/\kappa$ queries is allocated to remote peers that are sending it more than $1/\kappa$ queries. Also, if the LQB is not completely utilized by local peers, any leftover LQB is allocated to servicing queries from remote peers.

To illustrate Fractional-Spillover IAS by example, the IAS divides its RQB of 10 across its two links, thereby assigning a link quota of 5 queries each to nodes B and C. However, since node C is only sending 4 queries, the extra unused unit of capacity

[3]To keep our explanation of policies conceptually clear, we will not add floors and ceilings to quantities. In our simulations, floors are taken for most calculated quantites, and policies are run in multiple "rounds" to ensure that all available query bandwidth is used up.

```
C_r ← C_v { C_r is the capacity remaining }
N_r ← k { N_r is the number of nodes that still have queries to send }
for i = 0 to k do
  q_i ← 0
end for
while N_r > 0 and C_r > 0 do
  m ← C_r/N_r
  for i = 0 to k do
    N_i ← min(m, p_i)
    q_i ← q_i + N_i {update link quota for N_i more queries}
    C_r ← C_r − N_i
    if p_i = q_i then
      N_r ← N_r − 1
    end if
  end for
end while
```

Figure 2.3: Fractional-Spillover IAS

is allowed to spillover to be used by node B. Therefore, Fractional-Spillover IAS sets a link quota of $|I_{B,A}(t)| = 6$ for node B and $|I_{C,A}(t)| = 4$ for node C.

In the remainder of this chapter, we will refer to Fractional-Spillover IAS as simply Fractional IAS. In the next chapter, we will, however, study the trade-offs between using a Fractional IAS in which RQB that is unused by a peer sending less than $1/\kappa$ queries is wasted versus one in which RQB unused by one peer is allowed to spillover to other peers.

Pseduo-code for the $Fractional - Spill - IAS$ is given in Figure 2.3.

When we refer to fractional IAS in the remainder of the chapter, we are referring to fractional spillover IAS unless we state otherwise.

Other policies. Other choices for incoming strategies include least-queries-first and preferred neighbors. In a least-queries-first strategy, a node may choose to first process queries from the adjacent node that sends it the least number of queries, then process queries from the adjacent node that sends it the next highest number of queries, and so on until its query bandwidth is fully utilized. Alternatively, a preferred neighbors strategy involves a node first choosing to process queries from an adjacent node that

it "prefers" the most (using some definition of preference), followed by processing queries from the next most preferred adjacent node, etc. until its query bandwidth is fully utilized. For the remainder of this chapter, we only consider fractional and weighted IASes. However, other IASes warrant examination in Chapter 3.

2.1.4 Drop Strategy (DS)

This section describes drop strategies (DSs). When the IAS used for node j determines that no more than m queries may be accepted from a remote peer i, and i sends $|O_{i,j}(t-1)| = m + \Delta$ queries (where $\Delta > 0$), node j uses a DS to determine specifically which Δ queries to drop. In our examples below, node j receives $O_{i,j}(t-1) = \{2q_1, 2q_2, 6q_3\}$ where $q_1 = (id_1, a, 5, k_1)$, $q_2 = (id_2, a, 4, k_2)$, and $q_3 = (id_3, b, 4, k_3)$. That is, j receives 4 queries that originated at node a where two of them have a TTL of 5 and two of them have a TTL of 4, and 6 queries that originated at node b and have a TTL of 4.

To briefly illustrate these strategies at work, we will denote a query that originates at node a and that has a TTL of ϕ when it arrives at node j by a_ϕ.

The proportional and equal strategies described below make decisions about which queries to drop by considering the nodes at which the queries in $O_{i,j}(t-1)$ originated as well as their TTL.

Proportional. Let node j receive $O_{i,j}(t-1) = \{\eta_1 q_1, \eta_2 q_2, ...\eta_n q_n\}$. If j uses a proportional DS, it will accept up to $\frac{\eta_\chi}{\Sigma_{\chi=0}^{n}\eta_\chi}$ queries of type q_χ, $1 \leq \chi \leq n$.

In our example, if $m = 5$, then proportional DS chooses $I_{i,j}(t) = \{q_1, q_2, 3q_3\}$.

Equal. The equal DS chooses queries uniformly based on the origin of the query. If queries arrive at j from β different sources (not necessarily neighboring nodes), the equal DS will attempt to choose $\frac{m}{\beta}$ queries from each source. If some sources sent fewer than $\frac{m}{\beta}$ queries, then the extra query bandwidth will be shared equally across queries from sources that sent more than $\frac{m}{\beta}$ queries.

For instance, if $m = 4$, then the equal DS chooses $I_{i,j}(t) = \{q_1, q_2, 2q_3\}$.

OrderByTTL (PreferHighTTL / PreferLowTTL). The OrderByTTL strategy is used to drop either those queries with the lowest or highest TTLs, regardless of the nodes

at which they originated. We will refer to these strategies as PreferHighTTL and PreferLowTTL, respectively.

If $m = 1$, PreferLowTTL gives either $\{q_2\}$ or $\{q_3\}$. (Ties are broken arbitrarily.) Alternatively, for $m = 1$, PreferHighTTL gives $\{q_1\}$.

2.2 Metrics

To evaluate whether or not (and how well or how badly) the policies above may help us manage queries distributed by malicious nodes, we define a work metric, the concept of a service guarantee, and a damage metric that allows us to quantitatively determine the service loss a malicious node may be able to inflict on a Gnutella network. In addition, we will describe how we model "good" nodes (that use our policies) and how we model "malicious" nodes (that attempt to flood the network, possibly ignoring reasonable policies). Given a scenario (a set of nodes, a topology, a load of admitted queries) and the policies in use, we can simulate operation of the system. For some simple scenarios, we are also be able to obtain analytical results. Our metrics tell us how the system performs.

2.2.1 Work

Our definitons for work broadly measure the number of queries processed by one or more nodes in the network. More specifically, for a particular node j, $W_j(t)$ is the *work* or cumulative number of queries processed at node j from time 0 to time t. Furthermore, we distinguish *local work* from *remote work*. The local work, $L_j(t)$ is the cumulative number of queries that node j receives from its local peers and processes from time 0 to time t. Similarly, remote work, $R_j(t)$ is the cumulative number of queries that node j receives and processes from its remote peers from time 0 to time t. Of course, $W_j(t) = L_j(t) + R_j(t)$.

To understand how local and remote work changes with ρ, let us consider what happens if we start with $\rho = 0$ and slowly increase it.

If $\rho = 0$ for all nodes, then each of the nodes allocates all of its query bandwidth

to queries arriving from remote peers. Unfortunately, nodes send out $\rho C_v = 0$ queries during each time step. While each node is "all ears," no node is sending out any work. As a result, the total local work and total remote work are both 0.

As ρ increases, more and more queries are accepted from local peers, and more and more queries will be processed by the network. Both the local and remote work will increase. However, at some point, each node will be processing the maximum number of queries possible (as specified by its capacity, C_v). After this point, if ρ increases any further, nodes will have to start dropping remote queries, and the amount of remote work will start decreasing. However, since we make the assumption that local peers always admit ρC_v queries, the amount of local work will continue increasing as ρ increases.

Once $\rho = 1$, then each of the nodes allocates all of its query bandwidth to queries arriving from local peers, and do not service any queries from each other. The total local work will be maximum and the total remote work will be 0.

Consider the network topology $K_3 = (V = \{1, 2, 3\}, E = \{(1, 2), (1, 3), (2, 3)\})$ in which we have a network of three nodes with three edges completely connecting the nodes, and $C_v = 100, 1 \leq v \leq 3$.

To illustrate our definitions of local and remote work, let us calculate $L_1(2)$ and $R_1(2)$ when $\rho = \frac{1}{4}$ for all nodes. Additionally, we assume that local peers have an infinite supply of queries. At time $t = 0$, 25 queries per node are received from local peers, and $|G_1(0)| = |G_2(0)| = |G_3(0)| = 25$. At time $t = 1$, each of these sets of 25 queries are accepted for processing, and $|P_1(1)| = |P_2(1)| = |P_3(1)| = 25$. Nodes then broadcast these queries to each other, $|O_{i,j}(1)| = 25, 1 \leq i, j \leq 3, i \neq j$. Of course, at time $t = 1$, all local peers continue generating queries such that $|G_1(1)| = |G_2(1)| = |G_3(1)| = 25$. Hence, in the next time step $t = 2$, each node accepts local queries admitted in the previous time step, and remote queries arriving at $t = 2$. The cumulative local work processed at node 1 is $L_1(2) = |P_1(1) + P_1(2)| = 25 + 25 = 50$, and the cumulative remote work processed is $R_1(2) = |I_{2,1}(2)| + |I_{3,1}(2)| = |O_{2,1}(1)| + |O_{3,1}(1)| = 25 + 25 = 50$.

Note that none of the remote queries need to be dropped since $(1-\rho)c_1 = 75 > 50$. On the other hand, if we recomputed our example with $\rho = \frac{2}{3}$ for all nodes, then node

1 receives 66 queries from each of node 2 and node 3, and can only accept a combined total of $(1 - \frac{2}{3})100 = 33$ queries from both of them. Hence 99 queries are dropped, and $R_1(2) = min(|I_{2,1}(2)| + |I_{3,1}(2)|, (1 - \rho)c_1) = 33$.

2.2.2 "Good" nodes

In our model, "good" nodes have two important characteristics. Firstly, we make the simplifying assumption that the processing capacity C_v is the same for all nodes in the graph. In particular, $\forall v \epsilon V, C_v = C$, where C is some constant. Secondly, good nodes in the network are compelled to find a setting for ρ that maximizes the remote work. [4]

Definition 2.2.1 Optimal Rho, $\hat{\rho}$. *Let $\hat{\rho}$ be the setting for ρ that maximizes $\Sigma_{v \epsilon V} R_v(t)$.*

We may analytically solve for $\hat{\rho}$ in some cases assuming that we have knowledge of the topology of the network, as we will demonstrate shortly.

Also, for certain topologies the optimal value for ρ may be different for different nodes in the network. However, for simplicity, we will assume that we would like to have a common setting for ρ for all nodes. We will have to sacrifice some remote work to have a common ρ, but doing so will simplify the implementation of our load balancing policies in a real network. In [194], we prove that using the same $\hat{\rho}$ for all nodes in the network results in remote work that is no less than $\frac{\tau}{\tau+1}$ of the remote work that can be acheived if nodes were allowed to use different ρs. Furthermore, in real snapshots of Gnutella network topologies, the remote work lost by using the same $\hat{\rho}$ for all nodes is typically less than 2 percent of the maximum acheivable.

Consider the network topology $K_3 = (V = \{1, 2, 3\}, E = \{(1, 2), (1, 3), (2, 3)\})$ in which we have a network of three nodes with three edges completely connecting the nodes, and $C_v = 100, 1 \leq v \leq 3$. For K_3, the reader can verify that the setting at which ρ maximizes the remote work is $\frac{1}{3}$. (At this setting, the amount of new work

[4] Alternatively, we can maximize the total work, but maximizing the remote work has the benefit that it gives us the smallest possible setting for ρ for which the total work is maximized and the minimum number of remote queries are dropped. A more detailed discussion appears in [159].

admitted and sent to any given node is exactly equal to the amount of work that it can accept.)

While the appropriate setting for ρ might be obvious in our small example, it is important for good nodes in our network to be able to compute $\hat{\rho}$ for arbitrary networks. We provide a general formula for computing $\hat{\rho}$ for symmetric networks below, following some elementary definitions. In each of the following definitions, we assume a network topology (V, E) and time to live τ.

Definition 2.2.2 Distance, $d(j,k)$. *Let $d(j,k)$ be the length of the shortest path between nodes j and k. Note that $d(j,j) = 0$.*

Definition 2.2.3 Radial-Node-Set, $\delta(j,h)$. *Let $\delta(j,h) = \{\ v \mid d(j,v) = h\ \}$. That is, $\delta(j,h)$ is the set of nodes v such that the shortest distance between j and v is exactly h.*

Definition 2.2.4 Arial-Node-Set, $D(j,h)$. *Let $D(j,h) = \bigcup_{i=1}^{h} \delta(j,i)$. That is, $D(j,h)$ is the set of nodes v such that the distance between j and v is greater than or equal to 1 but less than or equal to h. Note that $D(j,h)$ does not include j itself. That is, $j \notin D(j,h)$. Informally, $D(j,h)$ is the set of nodes that are within h hops of j, not including j itself.*

Theorem 2.2.1 Optimal Rho, $\hat{\rho}$. *If all nodes v in V have $C_v = C$ for some constant $C > 0$ and we require that all nodes have ρ set to the same value, then*

$$\hat{\rho} = \frac{|V|}{\Sigma_{v \epsilon V} |D(v,\tau)| + |V|}.$$

Proof

To ensure that the amount of new work admitted in the network is exactly equal to the amount of work that each node can accept, capacity must be reserved to service any particular query at each node within τ hops of where the query originated. Each node j processes a maximum of $(1-\rho)C$ queries, and there are $|D(j,\tau)|$ nodes within distance τ. During each time step t, node j needs to service $\rho C|\delta(j,i)|$ queries that were admitted at time $t-i$ from nodes that are i hops away, $1 \leq i \leq \tau$. Hence,

for each node $j \epsilon V$, the number of queries that it processes must exactly equal the number of queries admitted within τ hops, if indeed j is to be able to process the new work admitted in a given time step. This yields the equation $|D(j,\tau)|\rho C = (1-\rho)C$ which can be solved to determine the optimal setting for ρ for node j. However, we require that all nodes share the same ρ, so we simply sum each side of the equation across all nodes: $\Sigma_{v \epsilon V}|D(v,\tau)|\rho C = |V|(1-\rho)C$ which as desired yields our theorem upon solving for ρ.

In summary, good nodes in our model set $C_v = C$, and $\rho = \hat{\rho}$ to maximize the remote work done by the network.

2.2.3 Malicious Nodes

We are interested in studying flooding-based attacks, and we model a malicious node such that it injects as many queries as it is capable of. However, there exist many other behaviors that a malicious node may engage in to cause harm to other nodes in the network.

For example, a malicious node may 1) amplify the TTL of such queries already flowing through the network, 2) conduct a "smurf"-attack in which it sends query-hit messages claiming that some victim node has the answers to every other node's queries, or 3) inject just a few queries that will have so many results that nodes may become overloaded forwarding query-hit messages. While there are many such options available to the adversary, we focus specifically on query floods.

Some of these other attack models are worthy of study in future research, and it is likely that we will be able to apply some of the lessons that we learn from our current study to minimize the effect of these other attacks.

To construct a query flood attack, a malicious node dedicates all of its processing capacity to injecting "useless" queries. Malicious nodes may be able to inject more than C queries. That is, a malicious node may be able to inject more queries than a good node is able to accept for processing. However, since a good node knows that other good nodes can send at most C queries, it only examines the first C queries from each incoming link during a given time step, and ignores the rest. While a

malicious node can inject more than C queries, the effect will be the same as if it injects exactly C queries. Hence, we set $c_m = C$, where m is a malicious node.

After injecting queries in a given time step, a malicious node has no processing capacity left over. In addition, it does not have any incentive to process or forward queries that are sent to it by remote peers. To model a flood injected by a malicious node, we have the malicious node set its reservation ratio ρ to 1, whereas good nodes typically have their reservation ratio set to a significantly lower value. Also, when good nodes process queries that are injected by malicious nodes, the processing of those queries does not contribute to remote work.

Alternatively, a node can statically allocate more capacity to incoming links from nodes in a way proportional to the number of hosts each of them are connected to (assuming such information is available and reliable). If such a strategy results in more resilience to DoS attacks, it will then be worthwhile to design mechanisms into P2P protocols that make authentic topology-related information available to nodes. In any case, it will be worthwhile to experiment with various capacity allocation strategies in our simulations as a first step to determining what enhancements to add to P2P protocols to achieve resilience to DOS attacks.

When there are more queries that can be handled, we also need a policy for selecting queries. There are many choices here. For instance, preference could be given to younger (or older) queries, to queries with a shorter (or longer) time-to-live, that have longer (or shorter) disjoint paths. (Of course, queries that have been seen before, that arrived from another neighbor, should be eliminated.) Preference could also be given to queries that originate from nodes that are "trusted" or have better "reputations" than others. For example, a node A could give priority to queries injected by a node B that has successfully responded to A's own queries. Or, node A could give priority to nodes that have produced many useful responses for other nodes. (Some P2P systems such as Mojo Nation [133] have this type of capability.)

Finally, there are the actual P2P protocol policies, i.e., what happens to queries once they are processed. For example, a query may be routed to all neighbors, unless its time-to-live has expired. Or the query may be sent to a subset of neighbors, based perhaps on the load they are generating or based on the state of routing indexes [7].

2.2.4 Service

A key metric that we can use to understand the effects of a malicious node in the network will be *service*, $S_{i,j}(t)$, the number of queries that originate at node i and are processed at node j at time t. The service $S_{i,j}(t)$ tells node i how many of its queries are processed by node j at time t. For example, if node 2 processes 5 of node 1's queries at time $t = 3$, then $S_{1,2}(3) = 5$.

We now more formally define the notion of service, and two variations of it, radial and arial service, that we use in our evaluations.

Definition 2.2.5 Service, $S_{i,j}(t)$. *Let* $S_{i,j}(t) = \sigma_{q.o=i}(\bigcup_{v \epsilon V} I_{v,j}(t))$. *Note that we use* σ *to be selection over multi-sets, as defined in bag-relational algebra.*

Definition 2.2.6 Radial Service, $R_j(h,t)$. *Let* $R_j(h,t) = \Sigma_{v \epsilon \delta(j,h)} S_{j,v}(t)$. $R_j(h,t)$ *denotes the total service that node* j *receives from all of the nodes whose shortest distance from* j *is exactly* h. *(Informally,* $R_j(h,t)$ *is the total service that* i *receives from all nodes that are exactly* h *hops away from* i *in the network.)*

Definition 2.2.7 Arial Service, $S_j(h,t)$. *Let* $S_j(h,t) = \Sigma_{v \epsilon D(j,h)} S_{j,v}(t)$. $S_j(h,t)$ *denotes the total service that node* j *receives from all of the nodes within* h *hops.*

While service might be an interesting metric in itself, we will most often use service in a comparative fashion in which we compare the service that a node receives under "normal" circumstances to the service that a node receives when there is a malicious node present in the network.

First, we can evaluate the *service guarantee* for each node. We assume no rogue nodes in the system, but we assume that all nodes admit the maximum number of local queries. We can then compute $S_{i,j}(t)$ as the number of queries with $q.o = i$ processed at node j at time t. We call $S_{i,j}(t)$ a guarantee because it represents the service j provides to i even when the system is fully loaded. We can compute a steady state guarantee, and we can also compute the average guarantee for i at all nodes.

We can also model a system with, say, one rogue node that does not follow the protocols, and compute the guarantees. For example, rogue nodes can increase the

time-to-live field of all queries that are forwarded through it, thereby amplifying the life of queries and generating unnecessary traffic. Rogue nodes that do not follow protocols can also launch any number of other DoS attacks. Each of these DoS attacks can be modeled, simulated, and the reduced guarantee can be calculated. The reduced guarantee can be considered the *damage* inflicted by the rogue node. For example, say node j is guaranteed 10 queries per time unit at node i, when no rogue node exists. If a rogue node causes i to only process 5 of j's queries, then the damage would be 50%.

2.2.5 Worst-case and Best-case Scenarios

In the damage distribution experiment described in Section 2.3, we will consider two scenarios in which a malicious node is present in the network: a best-case scenario, and a worst-case scenario. In both scenarios, there is a malicious node in the network. In each of the scenarios, we will study the effect of the malicious node on the network from the point of view of a "victim" node. In particular, we will be interested in the reduction in service that the victim node receives if there is a malicious node present in the network.

As mentioned above, a malicious node in our experiments is one that sets $\rho = 1$ in an attempt to flood the network with "useless" work. The malicious node does not carry out any behavior that explicitly attacks the node that we will call the victim. Nevertheless, we will still use the term victim for the node from whose point of view we are studying the impact of the malicious node's behavior. More specifically, we study the degradation in service that the victim node experiences due to the presence of the malicious node. It is most likely the case that other nodes suffer degradation in the service they receive from other nodes as well. However, by studying service degradations for different victim nodes in the network, we build an understanding of how much impact a malicious node has on various relative placements of the malicious and victim nodes in a particular topology. Hence, when we say best-case and worst-case scenario, we mean best-case and worst-case scenario from the standpoint of the victim when there is a malicious node in the network.

In the best-case scenario for the victim, we assume that none of the other non-malicious nodes are generating any queries, such that all nodes (except the malicious one) can spend their resources servicing queries from the victim. Such a case may occur in a real network at, say, 4 A.M. when the victim sends out queries to an unloaded network. To simulate the best-case scenario in our experiments, we set $\rho = \hat{\rho}$ for the victim (it is a good node), $\rho = 1$ for the malicious node, and $\rho = 0$ for all other nodes in the network.

On the other hand, the worst-case scenario for the victim occurs when all other nodes in the network are broadcasting as many queries as they are allowed by the optimal reservation ratio. In other words, $\forall j, t\ G_j(t) = \infty$ where j is a good node, and the reservation ratio that the good nodes choose is $\rho = \hat{\rho}$. The malicious node, as before, has its reservation ratio set to $\rho = 1$. The worst-case might be said to model a real network at 4 P.M. when it is at its peak load.

In the evaluations described in Section 2.3, we will consider a worst-case scenario in which there is a single malicious node in a small network of fully-loaded nodes.

In our worst-case scenario, we assume that all good nodes in the network are broadcasting as many queries as they are allowed to as per the optimal reservation ratio.

2.2.6 Victim Nodes

In some of our evaluations, we will study the effect of the malicious node on the network from the point of view of a victim node. In particular, we will be interested in understanding what is the reduction in service that the victim node receives if there is a malicious node present in the network.

As mentioned above, a malicious node in our evaluation is one that sets $\rho = 1$ in an attempt to flood the network with useless work. The malicious node does not carry out any behavior that explicitly attacks the node that we will call the victim. Nevertheless, we will still use the term victim for the node from whose point of view we are studying the impact of the malicious node's behavior. More specifically, we study the degradation in service that the victim node experiences due to the presence

of the malicious node. It is most likely the case that other nodes suffer degradation in the service they receive from other nodes as well. However, by studying service degradations for different victim nodes in the network, we build an understanding of how much impact a malicious node has on various relative placements of the malicious and victim nodes in a particular topology.

We define the service that the victim node receives from the network in a worst-case scenario as a *service guarantee*.

Definition 2.2.8 Service Guarantee, $S_j(t)$. *Let* $S_j(t) = S_j(\tau, t)$. $S_j(t)$ *denotes the total service that node* j *receives from all of the nodes within* τ *(TTL) hops.*

Our experiments in the next section will tell us whether the malicious node flooding the network will have more or less of an affect in this worst-case scenario.

2.2.7 Damage

With all of this machinery in place, we are now ready to quantify the degradation in service that might be brought about by a malicious node. We call this degradation in service *damage*.

In the following definitions, $S_j(t)$ refers to the service guarantee that j receives from the network when there is no malicious node present in the network, and $\overline{S_j(t)}$ refers to the same quantity when there does exist a malicious node in the network.

Damage with respect to a victim node j, $D_j(t)$, is defined as follows:

Definition 2.2.9 Damage for Victim Node j

$$D_j(t) = \frac{S_j(t) - \overline{S_j(t)}}{S_j(t)}$$

If $S_j(t) = \overline{S_j(t)}$, then the malicious node is not able to affect the service guarantee that j receives from the network at time t, and the corresponding damage is 0. On the other hand, if $\overline{S_j(t)} = 0$, then the malicious node is able to prevent j from receiving any service at all at time t, and the corresponding damage is 1.

We define cumulative network damage as the sum of the loss in service incurred by every non-malicious node in the network from time 0 to time t. Cumulative network damage is used in the experiments presented in Section 2.3.1 and 2.3.3.

Definition 2.2.10 Cumulative Network Damage

$$D(t) = \frac{\Sigma_{i=0}^{t}\Sigma_{j\epsilon V, j\notin M}(S_j(i) - \overline{S_j(i)})}{\Sigma_{i=0}^{t}\Sigma_{j\epsilon V}S_j(i)}$$

(where M is the set of malicious nodes.)

Similarly, the damage is 0 if the malicious node is not able to have an effect on the network, while the damage is 1 if the malicious node is able to prevent all remote work from taking place in the network.

Finally, we define cumulative radial damage as the reduction in service that a node j experiences at nodes h hops away due to the presence of a malicious node.

Definition 2.2.11 Cumulative Radial Damage. $D_j(h,t) = \frac{\Sigma_{i=0}^{t}\Sigma_{v\epsilon\delta(j,h)}(R_j(h,i)-\overline{R_j(h,i)})}{\Sigma_{i=0}^{t}\Sigma_{v\epsilon\delta(j,h)}R_j(h,i)}$. *$D_j(h,t)$ denotes the damage, or reduction in service that node i receives from all of the nodes whose shortest distance from j is exactly h.*

2.3 Results

In this section, we present the results of evaluations run using a simulator that we developed called Fargo.[5] Fargo implements the Gnutella traffic model described in Section 2.1, allows us to choose any of the policies described in Section 2.1.2 for a given network topology, and measures the metrics defined in Section 2.2.

In this chapter, the goal of our experiments is to understand the impact that a single malicious node can have when it is placed in different positions in various simple network topologies, and to study how we may employ incoming allocation and drop strategies to minimize the damage that a malicious node in the network

[5]Our simulator is named after a city in North Dakota that is frequently flooded by the Red River of the North that runs through it. More information about Fargo, North Dakota and the Red River is available at http://www.ndsu.nodak.edu/fargoflood.

can cause. Future work can build on this understanding to determine how best to construct larger, more complex networks that are resilient to flood-based DoS attacks with multiple malicious nodes.

We chose to evaluate small network topologies and a single malicious node to build a fundamental understanding of the issues and trade-offs that a system architect would need to keep in mind to design a flood-tolerant system.

All of our evaluations were run on small networks of either 14 nodes (for complete, cycle, wheel, line, and star topologies) or 16 nodes (for grid and power-law topologies[6]) with a single malicious node in the graph[7], and all queries were constructed with a TTL (τ) of 7. Therefore, in the small respresentative networks that we experiment with here, queries generated in any node of the network are capable of reaching every other node in the network. In addition, a setting of $\tau = 7$ matches that which is currently used in deployed Gnutella networks. A network size of 14 nodes was chosen for the complete, cycle, wheel, and star topologies because it was the smallest possible network that allowed us to cleanly determine the effect of the malicious node on victim nodes that were from 1 to TTL hops away. A network size of 16 nodes was chosen for the grid topology as it was reasonably small, yet allowed us to have a full 4 x 4 grid. The power-law topology was chosen to be 16 nodes such that comparisions could be made with the grid topology, but other sizes may have worked well also.

In the simulations, each node is given a maximum processing capacity of $C = 100$ queries per time step. Each of the evaluations was run for $t = 100$ time steps which was sufficient for the network to attain steady state in all of the cases.

Table 2.3 shows the cumulative damage incurred for different network topologies for the strategies outlined in Section 2.1.2. In the simulation results shown in this table, we assumed a worst-case scenario as defined in Section 2.2.

The first column of the table lists the topology used for a particular simulation. For topologies for which it made sense, simulations were done in which the malicious node was placed in different positions in the network, and the position of the malicious

[6]For the results shown in this chapter, we used a particular instance of a power-law topology as shown in Appendix A.

[7]For the reader that is interested in "brushing up" on graph theory, we provide the definitions for each of these topolgies in Appendix A.1.

	Fractional				*Weighted*			
Topology (Location)	*Prop*	*Equal*	*PfHighTTL*	*PfLowTTL*	*Prop*	*Equal*	*PfHighTTL*	*PfLowTTL*
Complete	0.143	0.143	0.143	0.143	0.633	0.633	0.633	0.633
Cycle	0.407	0.319	0.319	0.533	0.539	0.459	0.399	0.699
Grid (Center)	0.340	0.243	0.331	0.360	0.545	0.511	0.555	0.685
Grid (Corner)	0.282	0.232	0.266	0.405	0.455	0.372	0.378	0.613
Grid (Edge)	0.310	0.220	0.306	0.429	0.519	0.406	0.433	0.633
Line (Center)	0.393	0.360	0.387	0.457	0.500	0.426	0.458	0.616
Line (End)	0.162	0.126	0.135	0.299	0.225	0.185	0.165	0.366
Power-Law (High)	0.288	0.279	0.307	0.333	0.573	0.550	0.530	0.684
Power-Law (Low)	0.260	0.189	0.227	0.237	0.478	0.423	0.445	0.589
Star (Center)	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Star (Edge)	0.143	0.143	0.143	0.143	0.595	0.595	0.595	0.595
Wheel (Center)	0.354	0.354	0.354	0.354	0.756	0.776	0.756	0.777
Wheel (Edge)	0.440	0.359	0.387	0.412	0.560	0.508	0.561	0.672

Table 2.1: Total Cumulative Network Damage as a function of topology, IAS, and DS

node is indicated in parenthesis.

For example, for a star topology, the malicious node could either be the center of the star or one of the spokes. As might be expected, when the malicious node is in the center of a star topology, nodes at the spokes are unable to answer each other's queries at all and the resulting damage is 1.

The results in Table 2.3 help us answer the following questions in the indicated sections:

- Which IAS and DS strategies minimize damage the best? (and which strategies are the worst at minimizing damage?) Does the best IAS / DS strategy depend on the topology? Or do different IAS/DS strategies work better for different topologies? (Section 2.3.1)
- For a given topology, how much can the damage be reduced or mitigated by using the best IAS/DS compared to other strategies? How much better at reducing damage is the best IAS/DS compared to the others? (Section 2.3.2)
- For a fixed IAS/DS strategy, how does topology affect damage? Are there certain topologies that are less prone to damage than others? Are there particular nodes that are particularly susceptible to attack? (Section 2.3.3)
- How is damage distributed across the network? How do different combinations of

Topology (Location)	*Frac/Equal*	*Wght/Prop*	*Dmg Red Ftr*
Complete	0.143	0.633	4.4
Cycle	0.319	0.539	1.7
Grid (Center)	0.243	0.545	2.2
Line (Center)	0.360	0.500	1.4
Power-Law (High)	0.279	0.573	2.1
Star (Center)	1.000	1.000	1.0
Wheel (Center)	0.354	0.756	2.1

Table 2.2: Damage Reduction Factor using Frac/Equal IAS/DS

policies affect the distribution of damage? (Section 2.3.4)

2.3.1 IAS/DS Policies and Damage

Fractional IAS and Equal DS minimize damage independent of network topology. Weighted IAS and PreferLowTTL DS maximize damage independent of network topology.

From Table 2.3, we can see that the combination of the fractional IAS and equal DS minimize damage independent of graph topology and the location of the malicious node in the network. The fractional IAS limits the maximum number of queries that arrive from a particular link in the face of an overabundance of queries. All nodes that are adjacent to a malicious node will accept only some fraction of the malicious node's queries, and all nodes that are two hops away from the malicious node will only accept some fraction of that fraction. As such, the number of malicious queries that are received by a node drops off quickly with the node's distance away from the malicious node. Of those queries that are received from adjacent nodes, the equal DS fairly distributes available query bandwidth based on the origin of the queries, so the malicious node's queries are given the same weight as queries from other nodes, even if the malicious nodes sends many, many more of them.

Weighted IAS always incurs more damage independent of DS and topology. The weighted IAS allows queries that are part of a flood to have a significantly higher chance of being chosen for processing relative to legitimate queries.

In general, when nodes use the weighted IAS, damage increases as the average connectivity of the nodes increases. On the other hand, when nodes use the fractional IAS, damage decreases as the average connectivity of the nodes increases.

The PreferLowTTL DS never reduces damage, and often results in significantly more damage as compared to the other drop strategies. One potential reason to use the PreferLowTTL strategy might be to attempt to increase the "reach" of a query, and to attempt to obtain as many search results from nodes a great distance (but less than TTL hops) away from the originator of the query. We are aware of at least one company that is considering using PreferLowTTL as part of their flow control algorithm for this reason. Unfortunately, when the malicious node is the originator of a query, the PreferLowTTL strategy allows its queries to be spread as far as possible and incur a large amount of damage.

2.3.2 Damage Reduction

Damage reductions of 1.4 to 4.4 times can be achieved with Fractional/Equal IAS/DS, depending upon topology (see Table 2.2).

Table 2.2 shows the damage reduction factors that can be achieved by switching from a weighted/proportial IAS/DS to a fractional/equal IAS/DS for all of the topologies considered with the malicious node in the most threatening position. For example, employing fractional IAS and equal DS for the power law topology results in reducing damage by at least a factor of 2 as compared to weighted/proportional IAS/DS when the malicious node is highly connected. When the malicious node is not highly connected, damage can be reduced by a factor of 2.5.

To put this damage reduction factor in perspective, it is worthwhile to remember that we measure damage in a worst-case scenario, when the network is "fully-loaded" as defined in Section 2.2.5. At a time at which the network is *not* heavily loaded (and has no malicious node), a node is able to have many of its queries serviced; the number of queries that it has serviced by other nodes is greater than its service guarantee. When the network is at its busiest (4 P.M. on a weekday), again with no malicious node, a node receives an amount of service that is exactly equal to its

service guarantee. A node might have, for instance, 200 of its queries processed at other nodes. Our damage metric (as shown in Table 2.3) tells us how many queries a malicious node is able to rob the good node of at this busiest time. If the damage is 0.5, then the malicious node is able to rob the good node of 100 queries. By using a better IAS and DS policy, we might be able to reduce the damage. If the new damage using the better policies is 0.75, then we are able to recover 50 queries for the victim node; that is, other nodes will service 50 additional queries for the victim by using better policies when the malicious node is present. The damage reduction factor in this case is $\frac{0.75}{0.5} = 1.5$.

The damage reduction factors for various topologies and policies are shown in Table 2.2.

Also, when fractional IAS is used, we note that the damage incurred when using the PreferHighTTL DS is no more than 5 percent greater than the damage incurred when using the equal DS in most graph topologies (the grid is the exception). An implementation of the PreferHighTTL DS is likely to be simpler and more efficient than that of an equal DS, and might be used in place of it if implementation complexity is a concern.

2.3.3 Damage vs. Topology

The complete topology under the fractional/equal IAS/DS is the least prone to damage (compared to other topologies under the same IAS/DS), and is insensitive to the position of the malicious node.

Figure 2.4 shows how damage varies with topology and placement of the malicious node under the fractional/equal IAS/DS. These figures graphically depict the results in the fractional/equal column of Table 2.3. The results corresponding to the star topology with the malicious node in the center have been excluded as the damage is always 1, and the exclusion allows the reader to see other results with better resolution. Also, the names of the graph topologies have been abbreviated (K=Complete, C=Cycle, W=Wheel, L=Line, S=Star, P=Power-Law, G=Grid).

From Figure 2.4, we can see that if the malicious node can take on any position

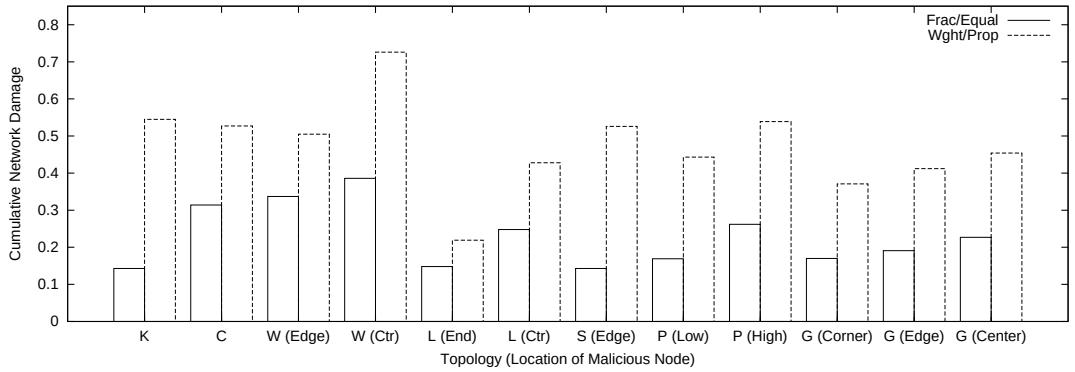

Figure 2.4: Damage vs. Topology for Fractional/Equal and Weighted/Proportional IAS/DS

in the network, then the complete topology minimizes damage. Of course, the use of fractional IAS plays a significant role in the complete topology's ability to minimize damage. The more links that a node using fractional IAS has, the less negative of an impact can be caused by a single malicious node connected to it. In Table 2.3, it is interesting to note that due to the symmetry of the actions taking place at each node, all of the drop strategies that we consider perform equivalently in a complete network.

From Figure 2.3, we also learn that topology alone cannot significantly reduce damage if bad policies are used. If a weighted/proportional IAS/DS is used instead of fractional/equal, we can see that the attacker is able to cause a damage of at least 0.5 for all topologies (in which the malicious node is in the most threatening position). By contrast, if fractional/equal IAS/DS is used, then the worst possible damage is 0.36. Hence, it is important to use good policies regardless of the topology of the network.

In all topologies, we find that damage increases as the connectivity of the malicious node increases. In addition, we find that the closer the malicious node is to the "center" of the network, the more damage it can cause. Therefore, when new "untrusted" nodes join a Gnutella network, they should be confined to the "edges" of the network. Over time, nodes that persist in offering service to the network can be moved towards the center. In today's Gnutella networks, nodes can join at any

random location and no explicit mechanism exists to control the position of or incrementally move nodes towards the center of the network based on past service that a node has offered.

Of course, a malicious node can "act" good until it finds itself in a central position in the network, and can start flooding at that time. Hence, while good policies can minimize the damage, it will be important to develop techniques that can detect malicious nodes such that they can be disconnected. Since good nodes in our model should admit no more than ρC_v new queries per time step when there is high load, it might be worthwhile to disconnect from any node that is sending more than ρC_v queries under a high load condition. However, in a real network, malicious nodes can easily forge the source addresses of queries, and can make it appear as if they are "good nodes" that are just forwarding queries that were admitted elsewhere. Nevertheless, while the idea of moving "trusted" nodes to the center does not prevent bad nodes from masquerading as good ones, it does "raise the bar" for an attacker to move into a more threatening position.

It is most likely that a cryptographic mechanism needs to be used or developed to authenticate the source of queries. Many network-layer DoS attacks take advantage of the fact that IP addresses can be spoofed (forged) in the same way. If packet creation could be authenticated, the same mechanism could be used to authenticate the source of queries in Gnutella, and it would become much easier to detect malicious nodes that inject floods.

In contrast to Fractional IAS (as shown in Figure 2.4), if a weighted IAS is used, then the complete network is at a disadvantage since the damage increases with the connectivity of each node. The only case that incurs even more damage than the complete network when a weighted IAS is used is the wheel topology with the malicious node in the center. In the wheel, the malicious node has the highest outdegree reaching every other node in the graph, and each other node gives the malicious node query bandwidth proportional to the number of queries it sends!

From Table 2.4, we can also see that some topologies are more sensitive to the position of the malicous nodes than others. When the malicious node takes on a high connectivity position or a "central" position in the topology, it is able to incur

	Fractional				Weighted			
topology	Prop	Equal	HighTTL	LowTTL	Prop	Equal	HighTTL	LowTTL
grid	0.058	0.023	0.065	0.069	0.090	0.139	0.177	0.072
line	0.231	0.234	0.252	0.158	0.275	0.241	0.293	0.250
power	0.028	0.090	0.080	0.096	0.095	0.127	0.085	0.095
star	0.857	0.857	0.857	0.857	0.405	0.405	0.405	0.405
wheel	0.086	0.005	0.033	0.058	0.196	0.268	0.195	0.105

Table 2.3: Damage Sensitivity as a function of topology, IAS, and DS

more damage in most topologies. In Table 2.3, we summarize how sensitive each topology is to the position of the malicious node under each of the IAS/DS policies. Rows for the complete and cycle topologies have been excluded as these networks are not sensitive to the position of the malicious node. Each entry of the table contains the difference between the highest and lowest amount of damage that can be caused by a malicious node somewhere in the network. For example, under the Fractional/Equal IAS/DS policy, we find that the wheel topology is least sensitive to the position of the malicious node, followed by the grid, power-law, line, and star topologies in order of increasing sensitivity.

Using the combination of Table 2.3 and Table 2.3, we can make statements about the trade-offs between damage and sensitivity to the position of the malicious node. For instance, while the complete topology offers the least damage under fractional/equal IAS/DS, it might be impractical to have every node connected to every other node for large networks. At the same time, one would like to have minimal damage and have a topology that is relatively insensitive to the position of malicious nodes. In this case, the grid topology seems to offer an interesting compromise– it incurs relatively low damage compared to the other topologies, and it is relatively insensitive to the position of a malicious node.

2.3.4 Damage Distribution

Fractional/Equal IAS/DS minimizes "flood" damage distributed throughout the network. The remaining damage that is not eliminated by Fractional/Equal IAS/DS is

"structural" damage that occurs because malicious node does not forward queries.

In this section, we measure how damage due to a single malicious node is distributed across the network.

Damage distribution is measured with respect to various "victim" nodes in the network. We examine the relation between incoming allocation / drop strategies and damage distribution across victim nodes.

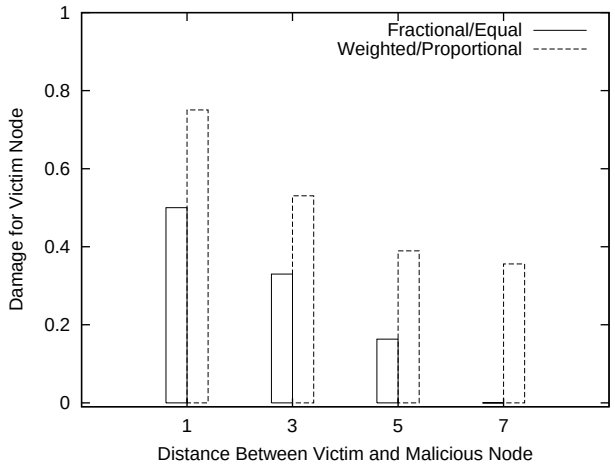

Figure 2.5: Damage vs. Distance from Malicious Node in a Cycle Topology

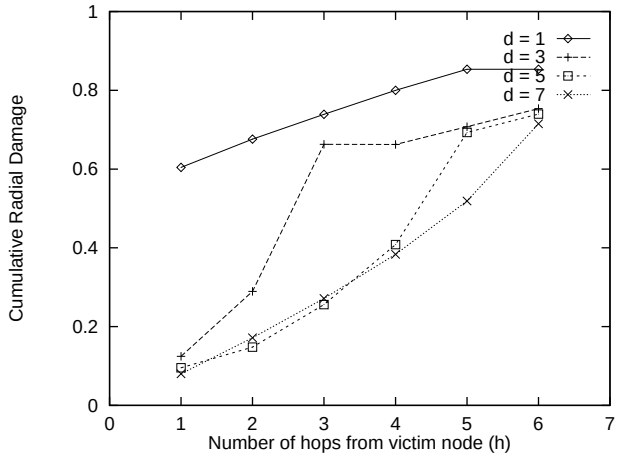

Figure 2.6: Damage Distribution for a Cycle with Weighted/Proportional IAS/DS

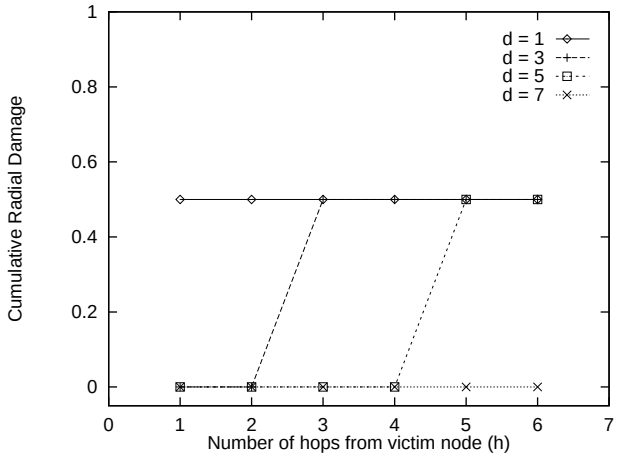

Figure 2.7: Damage Distribution for a Cycle with Fractional/Equal IAS/DS

Figure 2.5 shows the network damage incurred at the victim node when the victim and malicious nodes are separated by 1, 3, 5, and 7 hops in a cycle topology for various IAS/DS strategies. For all of the policies, damage decreases as the distance between the victim and malicious node increases.

Since damage decreases as distance from a malicious node increases, good nodes should attempt to make new connections in a way that distances them from malicious nodes. One method by which nodes can attempt to distance themselves from malicious nodes is by connecting to nodes that they "trust." That is, if a node i is reasonably sure that another node j is not malicious, then i should connect to j. Node j may be run by a friend of node i, or, in an enterprise setting, node j may have a business relationship with i. In either case, if node i connects to a "trusted" node j, then i can be reasonably sure that it has inserted at least one hop between itself and some malicious node that is part of the topology. Node j benefits from the same.

In the case that a node does not have any friends on the network, but can use a fractional IAS, then it should make many connections to shield itself from a potential flooding attack. If it makes m connections, then it accepts a maximum of $\frac{1}{m}$ useless queries from a malicious node. However, if a "friend-less" node is only capable of using a weighted IAS, then it should connect to just a few nodes. The more nodes that

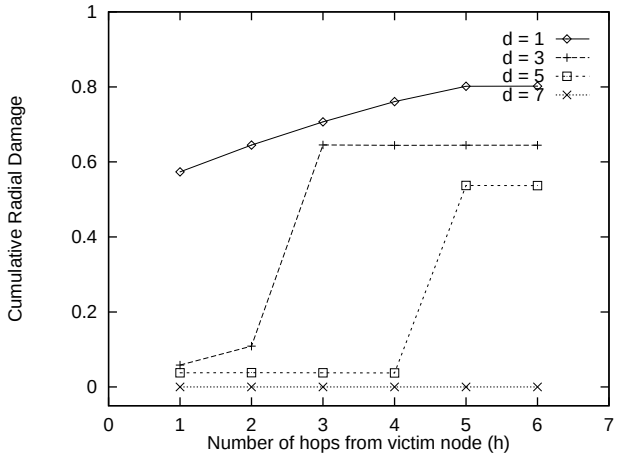

Figure 2.8: Damage Distribution for a Cycle with Fractional/Proportional IAS/DS

it connects to, the higher the probability that it will connect to a malicious node. For instance, if there is a single malicious node in a network of n nodes, then the probability that our friend-less node connects to a good node is $\frac{n-1}{n}$. The probability that it connects to a second good node is $(\frac{n-1}{n})(\frac{n-2}{n-1}) = \frac{n-2}{n}$ which is less than $\frac{n-1}{n}$.

In addition, nodes should attempt to connect to other nodes that are either themselves highly connected and using a fractional IAS, or lowly connected and using a weighted IAS. The less "exposed" that a node's neighbors are to flooding, the less exposed the node itself will be to flooding.

Figures 2.6 and 2.7 show how the damage incurred by the victim node in Figure 2.5 is distributed from 1 to τ hops away. Lines are plotted for different configurations of the victim and malicious node in which the distance (d) between them are 1, 3, 5, and 7 hops. Each (x,y) point on a line shows the reduction in service (y) that the victim receives x hops away.

In our following discussion of the results, we will use the terms "upstream" and "downstream." Upstream refers to the direction closer to the malicious node, and downstream refers to the direction farther away from the malicious node.

Similar to the bar chart in Figure 2.5, Figures 2.6 and 2.7 show that more damage is incurred at upstream nodes that are closer to the malicious node (and further from the victim).

However, they also show how the damage is distributed at various distances away from the victim node. The damage (y) in Figure 2.6 that is incurred by the victim is averaged over both nodes that are x hops away from the victim in the cycle. When the distance (d) between the victim and the malicious node is 1, the damage is always greater than the damage when the distance between the two is 7, as can be seen by the fact that the $d = 1$ line is always higher than the $d = 7$ line.

What we could not see in Figure 2.5 is that the extent of the damage one hop away from the victim is much more significant when the malicious node is one hop away than when it is seven hops away. When the malicious node is one hop away, the victim can only receive service from the good node that is one hop away, so the damage is at least 0.5. In addition, the victim is forwarding the flood queries from the malicious node to the good node that is one hop away. The good node is using the proportional DS, and, as a result, drops some of the victim's queries while attempting to process the flood of queries that arrives at its door. The damage one hop away is therefore more than 0.5 in the $d = 1$ case; it is 0.61 in Figure 2.6. The damage two hops away is even more (0.68). Firstly, the victim's queries are never able to reach the upstream node two hops away because the malicious node never forwards them (contributing 0.5 to the damage). Secondly, since the downstream node one hop away does not accept all of the victim's queries, it does not forward all of the victim's queries to the downstream node two hops away, and, of those that are forwarded, the downstream node two hops away accepts only a proportion of the victim's queries (incuring an additional 0.18 damage). On the other hand, the damage one hop away from the victim is much less significant when the malicious node is seven hops away as can be seen by the $d = 7$ line. Of course, the damage increases (the victim receives less service) at nodes that are closer to the malicious node.

There are two types of damage that are caused by the malicious node. *Structural damage* is caused because a malicious node does not process or forward queries itself. When the malicious node is one hop away from the victim, the structural damage is 0.5 one hop away since the malicious node does not process any of the victim's queries. A second type of damage, *flood damage*, is caused by the traffic that the malicious node creates. When the malicious node is one hop away, and we are using

a proportional DS, there is flood damage that occurs at the good node that is one hop away. Due to the malicious query traffic that is forwarded to the good node, the good node cannot process all of the victim's queries. The flood damage in this case is 0.11.

Looking at Figure 2.7, we can see that by switching from a weighted/proportional IAS/DS to a fractional/equal IAS/DS, we are able to avoid flood damage, and we are only left with structural damage. In particular, when the equal DS policy is used, the good node that is one hop away processes one of the victim's queries for each of the malicious node's queries before it uses all its remaining query bandwidth to service additional queries from the malicious node. Therefore, all of the victim's queries are processed at the good node, and the only damage that the victim suffers one hop away is structural.

By analyzing the damage distribution, we are able to see that good policies (in particular, the fractional/equal IAS/DS) are able to contain flood damage. However, other mechanisms need to be developed to contain structural damage. Malicious nodes need to be detected and disconnected from the network to deal with structural damage. The curves corresponding to the configurations where the separation distance between the victim and malicious nodes are 3 and 5 have spikes in the level of damage 3 and 5 hops away from the victim, respectively, because the malicious node does not process any of the victim's queries. Some service is still received at distance 3 and 5 from the corresponding downstream node. Since the quantity of the malicious node's queries is larger than the quantity of the queries from good nodes, more malicious queries are processed at upstream nodes in weighted IAS.

Once a fractional IAS is used as shown in Figure 2.8, the upstream nodes are able to significantly contain the damage. All of the queries that the victim broadcasts towards upstream nodes are serviced because the fractional IAS guarantees that the receiving node will first provide service to all of the victim's queries before allocating any remaining query bandwidth the the malicious node's queries. However, some damage still occurs. If the victim node is within τ hops of the malicious node, it receives a significant number of malicious queries. It then forwards the combination of these malicious queries and its local queries downstream. Nodes receiving this traffic

do not have the processing capacity to service all of these queries. In Figure 2.8, these nodes use the proportional drop strategy to determine which of the combination of these queries to service. A few of the victim's nodes queries are inevitably dropped, resulting in some amount of damage.

The equal drop strategy is able to further eliminate this damage, as shown in Figure 2.7. The equal drop strategy does not choose which queries to service from the combination of malicious and good queries based on the respective quantity of them, but instead gives equal opportunity to queries that originated at different nodes. Therefore, downstream nodes select at least as many good queries as malicious queries under the equal drop strategy. This minimizes the "downstream" damage to the victim to 0. There still exists damage "upstream" since the malicious node blocks the victim's queries from being forwarded past it, but the equal drop strategy minimizes the damage to the victim to the best extent possible given the presence of the malicious node.

2.4 Discussion

One concern that one might have regarding the use of a reservation ratio is that it restricts the number of queries that a supernode can process from its local peers during a given time step to a maximum of ρC. If a supernode was to discard queries arriving from local peers, those queries would "never see the light of day" and would most likely result in an unsatisfied user that perceives the system as unreliable. Of course, we would recommend queueing these queries either at the supernode or at the client local peer until a future time step at which point the query could be accepted for processing. However, we would not want the sizes of these queues to grow too quickly.

To determine how potentially serious of an issue this is, we can do a back of an envelope calculation to understand approximately how many local queries might need to be queued. From the data in [208], we can calculate that the approximate number of machine instructions required to process a query is 10^5. A 1GHz Pentium III supernode can execute 3 instructions per cycle, and can therefore service $\frac{3*10^9}{10^5} = 30,000$

queries per second. Hence, in our model $C = 30,000$, the supernode will be able to accept $30,000\rho$ queries from all of its local peers per second. Based on a informal study of supernodes on a Gnutella network, each supernode has approximately 50 local peers connected to it [43], and a supernode will be able to process 600ρ queries per second from each local peer. From [208], we also find that the expected number of queries that are submitted by an average user per second is $9.26 * 10^{-3}$. As long as $\rho > 1.5 * 10^{-5}$, none of the queries from a local peer will have to be queued or dropped. Even for a large power-law network in which tens of thousands of nodes are reachable within a TTL of 7, the optimal value of ρ will be large enough to accept all of the queries from a local peer on average.

Nevertheless, there exist at a number of options for how to handle local peers that admit a significantly above average amount of traffic. Here are some of them:

- A supernode can require that a local peer queue any more than ρC queries.
- A supernode that is currently accepting ρC queries from its existing local peers can refuse to engage in connections with other local peers. If other local peers attempt to connect to it, the supernode can respond with an "I'm too busy" message.
- Local peers that consistently expect to be generating more than ρC queries per time step can connect to multiple supernodes, and send each of them ρC queries.

2.5 Related Work

Most of the denial-of-service research to date has focused on network-layer attacks [136, 191, 192, 157, 158, 190, 63, 181, 70, 27, 146, 86, 145, 4, 210, 147]. There have been mutliple proposals to build IP Traceback mechanisms to manage network-layer DoS attacks including [191] and [189].

Savage et al. [136] uses backscatter to measure the frequency with which network-layer DoS attacks take place on the Internet. We are not aware of any similar

work that measures the frequency of either flooding-based or non-flooding-based application-layer attacks. We are unsure of how many application-layer attacks are currently taking place on P2P networks. However, we do know that the Recording Industry Association of America (RIAA) has considered using DoS-like attacks to prevent users from trading copyrighted songs, although they have not gone forward with their plans due to concerns regarding violating cyberterrorism legislation [148, 165]. We believe P2P networks should be proactively designed early in their evolution to be able to contain such attacks, as opposed to dealing with the problem with only after wide deployement as happened with protocols such as TCP, IP, and ICMP.

Osokine [150] proposes a Q-algorithm intended for solving traffic management problems in Gnutella, but the algorithm could also be used to address DoS attacks. Rohrs [175] proposes a simplified version of Osokine's work that has been implemented in the LimeWire Gnutella client. No evaluation has been published on either proposal.

Some of the policies we propose to use to manage query floods are similar to those that have been used in link scheduling for years [155]. Algorithms such as weighted fair queuing (WFQ) have been shown to optimally allocate a fair share of link bandwidth with respect to weights. We could use WFQ to manage query flow in Gnutella nodes, but we would still need to decide on how to choose weights to minimize the damage from DoS attacks. The IASs that we use in our work can be viewed as choosing different weights for incoming query flows.

Also, some related work has been done in the area of traffic shaping (i.e., [170]). However, most traffic shaping work deals with how to classify, police, schedule, and shape network-layer traffic travelling from one node to another. Traffic shaping research has typically assumed that nodes are not malicious in that they do not spoof fields in an IP packet. Because the first step of traffic shaping, classification, relies on truthful information in the IP header, traffic shaping techiques are inapplicable to the problem of mitigating application-layer DoS attacks in P2P networks, in which malicious nodes can spoof information in the data packets they send. By contrast, in our work, we assume that packets sent by malicious nodes are indistinguishable from packets sent by good nodes.

Additional related work has also taken place in the area of load balancing (i.e.,

[85]). The typical problem in the load balancing literature involves a master server and a number of slave servers. The master is usually referred to as the load balancer, and the slaves could, for instance, be web servers. The problem that the master needs to solve is how to assign the handling of requests to the slaves in such a manner as to ensure that the load (i.e., in terms of CPU utilization, IOs per second, and/or other metric(s)) is balanced across the slaves. Of course, in a P2P system, it is unclear as to which nodes should be trusted to be masters, and which nodes would accept roles as slaves. As client autonomy is often a goal of P2P systems, that goal would be in conflict with a master-slave approach towards load balancing. In our work, we do not require nodes to take on master or slave roles. Instead we suggest that good nodes send no more than $\hat{\rho}C$ queries while concurrently using policies that mitigate potential overloads caused by malicious nodes that send even more queries.

Much of the security-related research that has taken place in the P2P area has focused on providing anonymity to users of the system, and ensuring fair resource allocation. Free Haven [57, 174, 55], Freenet, and to some extent Gnutella provide "reader" anonymity in that they are designed to prevent a third-party from determining which node a query originated at. Some of these systems provide other types of anonymity as well. Publius [121], Freenet, and Free Haven, for example, prevent censorhip by providing "publisher" anonymity. Systems such as Mojo Nation [134] and Reputation Server [111, 168] are designed to allocate resources fairly. Mojo Nation does this by issuing a form of digital cash called "Mojo" that must be "spent" to issue queries and to publish documents. (This also enables Mojo Nation to deal with some types of DoS attacks.) Reputation Server keeps track of user's "reputation," such that users are given the incentive not to abuse the system else their reputations will be downgraded, thereby affecting the future utility that they will be able to derive from using the system.

Most other research that has been done in the area of P2P systems has been in the areas of efficient search, routing, and indexing. Beyond what systems like Gnutella and Morpheus do, systems such as Chord [193], CAN [164], Pastry [176], Tapestry [212], and Viceroy [38] can provide guarantees on the maximum number of nodes that need to be queried in order to find an answer to the query if an answer exists

in the network. These guarantees are usually provided at the expense of reduced autonomy; restrictions on how nodes must connect or where documents must be stored are imposed to enforce guarantees.

2.6 Chapter Summary

Gnutella networks are highly susceptible to application-layer, flooding-based DoS attacks if good load balancing policies are not employed by nodes on the network. In this chapter, we have defined a model and metrics that allow us to concretely measure the amount of damage that a malicious node can cause with query flooding. Through simulations on small representative networks, we have determined how damage can be minimized with load balancing policies, how damage varies as a function of network topology, and how damage is distributed.

Chapter 3

DoS in a Real Gnutella Topology

In the previous chapter, we studied the fundamental effects of query floods on small, representative graph topologies (complete, cycle, wheel, line, star, power-law, and grid) of 14 or 16 nodes. We studied the problem of how to use traffic management policies to balance excessive load that may be injected by a single malicious node in these networks. We found that if nodes use certain policies, such as prefering queries that have high TTLs (times-to-live) and carefully choosing how many queries to process from each of their links, the damage to the network can be reduced by a factor of two to four, depending on the network topology and location of the malicious node.

In this chapter, we evaluate the effects of query-floods issued by many malicious nodes on a "real" topology with over 1750 nodes obtained by crawling the Gnutella network. We develop and experiment with a number of query selection policies, and we study the damage that occurs to the network when malicious nodes are able to take on positions of their choice in the network.

Our main contributions in this chapter are:

- We evaluate various query selection policies on a real topology snapshot obtained from the Gnutella network.
- We propose and evaluate least-queries-first, probabilistic acceptance, and TTL-Shaping query selection policies. We show that they outperform policies studied

in the previous chapter in some scenarios with respect to mitigating query-flood DoS attacks.

- We evaluate the impact of various flooding strategies and attack models that malicious nodes can employ, and explore the trade-offs involved in using several query selection strategies against them.

3.1 Additional Policies

In this section, we describe a number of additional IASes and DSes that we experiment with to mitigate floods by many malicious nodes in larger Gnutella networks.

In describing these additional policies, we use the same traffic model for the Gnutella network as was described in Section 2.1, and the same notation. Furthermore, we illustrate our policies with the same running example as was used in Section 2.1.2. For convenience, we repeat the set up for the running example here. Consider a node A that has two edges to nodes that are sending it queries. Node A has a link to node B that is sending it 16 queries, and also has a link to node C that is sending it 4 queries. We say that $p_1 = 16$, $p_2 = 4$, and $d(A) = 2$. Also, for the purposes of this example, we assume that node A has a RQB of 10 ($(1-\rho)C_A = 10$), and it must decide what quotas to set for each of its links.

We now describe the additional IASes and DSes that we study in this chapter.

3.1.1 Incoming Allocation Strategies (IAS)

In the last chapter, we studied Weighted and Fractional-Spillover IASes. In this section, we introduce four new IASes. While we studied only Fractional-Spillover IAS in the last chapter, we extend our study to include Fractional and a Null IAS in this chapter, in addition to two other IASes.

In our study in this chapter, we will no longer consider Weighted IAS due to its dismal performance. Weighted IAS was used as a benchmark for the case in which queries were accepted on a uniform random basis– the more queries that were received from a particular link, the more queries were processed from that particular link. In

this chapter, we instead use the Null IAS as a benchmark for reasons we describe shortly.

- *Null IAS ($Null - IAS$).* In a Null IAS, link quotas are trivially set to be the number of queries arriving on that link. A maximum of p_i queries can be accepted from the ith link. That is, $Null - IAS(\langle O_{u_1,v}(t-1), O_{u_2,v}(t-1), ..., O_{u_k,v}(t-1)\rangle) = \langle p_1, p_2, ..., p_k\rangle$. When Null IAS is used, a DS is responsible for dropping queries if the sum of the link quotas exceed a node's capacity constraint. In our example, node A's link quotas are $p_1 = q_1 = 16$, and $p_2 = q_2 = 4$. Null IAS does not allow a node to discriminate amongst which queries it accepts based on the link on which the queries arrive.

  Note that the Null IAS is distinctly different from the Weighted IAS. A Weighted IAS would have set link quotas of $q_1 = \frac{16}{20}10 = 8$ and $q_2 = \frac{4}{20}10 = 2$ respectively for nodes B and C. Based on the link quotas set by Weighted IAS, 8 queries from node B and 2 queries from node C would have to be dropped. If a PreferHighTTL DS is used, the two queries from node C with the highest TTLs would be accepted using the link quotas set by Weighted IAS. However, if a Null IAS is used, then if the four queries sent by Node C all had higher TTLs than those queries sent by Node B, *all four* of the queries sent by Node C would be accepted instead of just two of them. Therefore, Weighted IAS and Null IAS are distinctly different IASes.

After conducting our work in the previous chapter, we conducted some theoretical analysis on our Gnutella traffic model in [194] using the Null IAS [1]. We will use the following three results from [194] in this chapter:

1. We proved that using a Null IAS with a PreferHighTTL DS maximizes remote work when all nodes in the network set $\rho = \hat{\rho}$;

2. We provided and proved the correctness of a procedure that can be used to determine $\hat{\rho}$ for an arbitrary network topology; and,

[1] The work in [194] can also be found in Qi Sun's Ph.D. dissertation [196].

3. We showed that when Null IAS and PreferHighTTL DS are used and all nodes set $\rho = \hat{\rho}$, the network converges to steady-state within τ time steps.

Furthermore, comparing Null IAS to Weighted IAS, we note that Weighted IAS does not offer the guarantees we proved in [194] with regards to maximizing remote work. As such, we choose to study Null IAS instead of Weighted IAS in this chapter.

While the combination of Null IAS and PreferHighTTL DS can be used to maximize remote work when all of the nodes in the network are good, malicious nodes can have a disastrous effect on the network when these policies are used. As such, we study additional IASes and DSes that might result in non-optimal performance when all of the nodes are good, but which significantly mitigate the effects that malicious nodes can have.

- *Fractional IAS.* Fractional IAS is geared at giving each of a node's incoming links an exactly equal fraction of remote query bandwidth. If a node v has k remote peers ($k = d(v)$), a Fractional IAS allocates a link quota of $\frac{C_v}{d(v)}$ to each of its remote peers. In other words, $Frac - IAS(\langle O_{u_1,v}(t-1), O_{u_2,v}(t-1), ..., O_{u_k,v}(t-1)\rangle) = \langle min(\frac{C_v}{k}, p_1), min(\frac{C_v}{k}, p_2), ..., min(\frac{C_v}{k}, p_k)\rangle$. Any extra query bandwidth that is unused by a remote peer is *not* allocated to other remote peers. For instance, if node A were to use a Fractional IAS, it would set a link quota of $min(\frac{10}{2}, 16) = 5$ queries for node B and $min(\frac{10}{2}, 4) = 4$ queries for node C. Note that one unit of RQB goes unused with Fractional IAS since node C is sending only 4 queries.

  In addition to studying the Fractional IAS in which extra query bandwidth is left unused, we also study Fractional-Spillover IAS. While Fractional-Spillover IAS was defined in the previous chapter, we repeat the definition in this chapter for convenience.

- *Fractional-Spillover IAS.* This IAS works exactly like the Fractional IAS, except that any extra query bandwidth that is unused by a remote peer *is* allocated to other remote peers. That is, any extra capacity that is not used by one remote peer "spills over" to other links that might be able to use that capacity. Pseudocode for the Fractional-Spillover-IAS is given in Figure 3.1. When node A used

```
{ Input: ⟨O_{u1,v}(t−1), O_{u2,v}(t−1), ..., O_{uk,v}(t−1)⟩ }
{ Output: ⟨q1, q2, ..., qk⟩ }
C_r ← C_v { C_r is the capacity remaining }
N_r ← k { N_r is the number of nodes that still have queries to send }
for i = 0 to k do
  q_i ← 0
end for
while N_r > 0 and C_r > 0 do
  m ← C_r/N_r
  for i = 0 to k do
    N_i ← min(m, p_i)
    q_i ← q_i + N_i {update link quota for N_i more queries}
    C_r ← C_r − N_i
    if p_i = q_i then
      N_r ← N_r − 1
    end if
  end for
end while
```

Figure 3.1: Fractional-Spillover IAS

Fractional IAS in our running example, one unit of its capacity was wasted since node C's link quota was 5 queries, but it only sent 4. In Fractional-Spillover IAS, node A would allow the one extra unit of query bandwidth to spillover to node B such that node A would set the link quota for node B to be 6 queries, and the link quota for node C to be 4 queries.

- *LQF-IgnoreLastLink IAS.* In query-flood attacks, malicious nodes may be sending significantly larger numbers of queries than good nodes, creating a bi-modal distribution of the number of queries forwarded from one node to another. LQF-IgnoreLastLink hypothesizes that links that have large numbers of queries being sent over them are likely to be carriers of malicious queries, while links that have few queries being sent over them may not be as likely to carry malicious queries[2].

  We now describe how a node v uses LQF-IgnoreLastLink to set link quotas. Node v chooses to allocate a link quota of p_i for queries arriving from node

[2]In Section 3.2, we will explore under what conditions this hypothesis holds.

$C_r \leftarrow C_v$ {C_r is the capacity remaining}
for $i = 0$ to k **do**
  $q_i \leftarrow 0$
  {The tuples of A_i will be referred to as $A_i.index$ and $A_i.value$}
  $A_i(A_i.index, A_i.value) \leftarrow (i, |O_{u_i,v}(t-1)|)$
end for
Sort A_i by $A_i.value$ in ascending order.
$j \leftarrow 1$
while $(j \leq k)$ and $(C_r - A_j.value \geq 0)$ **do**
  $q_{A_j.index} \leftarrow A_j.value$
  $C_r \leftarrow C_r - A_j.value$
  $j \leftarrow j + 1$
end while

Figure 3.2: LQF-IgnoreLastLink IAS

u_i where p_i is minimum, $\forall i$. After allocating the link quota for node u_i (i.e., the node sending the least queries), if there is remaining capacity, node v will repeat the process and will allocate a link quota for the link with the next least number of queries. The process continues until the amount of remaining capacity is less than the number of queries arriving on any unselected link. Any left over capacity is *not* used, but is intentionally "wasted." That is, the queries from the last link considered are ignored, by design, under the assumption that the queries arriving on that link may be part of a query-flood sent by a malicious node.

Applying LQF-IgnoreLastLink to our running example, node A would first choose to set its link quota for node C to be 4 queries, since node C is sending the least number of queries. Since node A's remaining capacity is then only 6, the 16 queries sent to it by node B on its "last link" are ignored, and the link quota for node B is set to 0. Pseudo-code for the $LQF-IgnoreLastLink-IAS$ is shown in Figure 3.2.

The IASes that we have considered thus far are "static" in the sense that given a set of queries arriving on a good node's incoming links, the good node will always allocate the same link quotas. However, malicious nodes could potentially use this

predicatability to their advantage. We therefore develop and study a *Probabilistic Accept IAS* in which good nodes allocate link quotas probabilistically.

- *Probabilistic Accept IAS.* Since nodes that send too many queries could be doing so as part of a query flood attack, an IAS may want to penalize nodes that send too many queries, but not in as predictable fashion as in the LQF-IgnoreLastLink IAS. In a *probabilistic accept (PA)* IAS, a node does the following: 1) it assigns probabilities to all its links (these probabilities sum up to 1.0), 2) it chooses a random number between 0 and 1, and 3) it allocates a link quota for one of its links based on the random number. The link for which a quota is allocated is then removed from consideration, and the process is repeated until the remaining capacity is smaller than the number of queries arriving from the next candiate link. Similar to the LQF-IgnoreLastLink IAS, this last set of queries is dropped.

  The probabilities assigned during step 1 are inversely proportional to the number of queries that are sent over that link. In particular, the probability is proportional to $\frac{1}{q_u}$ where q_u is the number of queries arriving from node u.

  Pseudo-code for the PA IAS algorithm is shown in detail in Figure 3.3.

  While good nodes set link quotas probabilistically, it is still possible for a malicious node to maximize the expected number of its queries that are selected by good nodes. We describe such a "PA-Flood" attack in Section 3.1.3.

  In our running example, if node A were to use PA IAS , the probability with which a link quota would be set for node B is $\frac{\frac{1}{16}}{\frac{1}{4}+\frac{1}{16}} = 0.2$, and the probability with which a link quota would be set for node C is $\frac{\frac{1}{4}}{\frac{1}{4}+\frac{1}{16}} = 0.8$. Note that since node B is sending four times as many queries as node C, the probability that its link quota is set is four times smaller than the probability that node C's link quota is set. Which link quotas are actually set is decided at run-time depending upon the outcome of a random number generator.

{Input: $\langle O_{u_1,v}(t-1), O_{u_2,v}(t-1), ..., O_{u_k,v}(t-1)\rangle)\}$
{Output: $\langle q_1, q_2, ..., q_k\rangle$ }
$C_r \leftarrow C_v$ {C_r is the capacity remaining}
repeat
  $T \leftarrow 0$
  for $i = 1$ to k **do**
    $q_i \leftarrow 0$
    $A_i \leftarrow (i, \frac{1}{p_i})$ { The tuples of A_i will be referred to as $A_i.index$ and $A_i.value$}
    $T \leftarrow T + A_i.value$
  end for
  $p \leftarrow rand(0, 1)$ {$rand(a, b)$ returns a real random number between a and b, inclusive.}
  $S \leftarrow 0$
  $i \leftarrow 0$
  repeat
    $i \leftarrow i + 1$
    $A_i.value = A_i.value/T$
    $S \leftarrow S + A_i.value$
  until $(i > k)$ or $(S > p)$
  $q_i \leftarrow p_i$
until $(C_r - A_i.value < 0)$

Figure 3.3: Probabilistic Accept IAS

{ Input: $O_{u_i,v}(t-1), q_i$) }
{ Output: $I_{u_i,v}(t)$ }
Sort the queries in $O_{u_i,v}(t-1)$ by their TTL field in descending order.
Let $I_{u_i,v}(t)$ be a set of queries of size q_i from $O_{u_i,v}(t-1)$ with highest TTLs.
{Ties are broken arbitrarily.}

Figure 3.4: PreferHighTTL DS

3.1.2 Drop Strategy (DS)

Reusing the notation introduced in the previous section, if $\Sigma_i q_i > C_v$, then we use a DS to determine which queries to drop. There are many potential DSes that nodes might choose to employ. For instance, a DS may choose to accept queries from distinct sources with equal probabilities. (While we could stamp queries with the id of the source node at which the query was originally admitted, keep in mind that this information can easily "spoofed.") In the previous chapter, we studied four different DSes including ones that weight queries based on their sources, and ones that prefer queries with higher (or lower) TTLs. We found that choosing queries with high TTLs performs comparably to equally weighting queries based on their source ids, and since source ids are spoofable anyway, choosing queries with high TTLs is a good choice for a DS. (Note that TTLs are harder to spoof since as long as there are some good nodes on the path that a query traverses, the TTL of the query will be decremented.) Pseudo-code for such a *PreferHighTTL* policy is shown in Figure 3.4.

In addition to studying *PreferHighTTL* DS, we also study a *TTLShaping* DS. The *TLLShaping* policy is based on the observation that a good node g should receive at most ${d_\alpha}^{(\tau-t)}\hat{\rho}C$ queries with TTLs of t, where τ is the maximum TTL used in the network, $\hat{\rho}$ is the reservation ratio that we expect good nodes to use, and d_α is a function of the degrees of the nodes along the path from the node that issued the query to the node that is receciving it.

For example, consider the seven-node network in Figure 3.6, and a setting of $d_\alpha = 2$. In the figure, node v is connected to two other nodes u and w. If nodes u and w are not malicious, they should admit at most $\hat{\rho}C$ queries each that they send to u. Since v is directly connected to u and w, it should accept ${d_\alpha}^{(\tau-t)}\hat{\rho}C =$

procedure $TTLShaping - DS(O_{u_i,v}(t-1), q_i)$ returns $I_{u_i,v}(t)$
  Let $I_{u_i,v}(t) \leftarrow \emptyset$
  Sort the queries in $O_{u_i,v}(t-1)$ by their TTL field in descending order.
  $C_r \leftarrow q_i$
  for $t = \tau$ downto 1 **do**
    $T \leftarrow \sigma_{q.ttl=t} O_{u_i,v}(t-1)$
    if $|T| > {d_\alpha}^{(\tau-t)} \hat{\rho} C$ **then**
      Remove any arbitrary $|T| - {d_\alpha}^{(\tau-t)} \hat{\rho} C$ queries from T.
    end if
    Let $C_t \leftarrow min(C_r, |T|)$
    Let U be a subset of C_t queries from T
    $I_{u_i,v}(t) \leftarrow I_{u_i,v}(t) \cup U$
    $C_r \leftarrow C_r - C_t$
  end for

Figure 3.5: TTLShaping DS

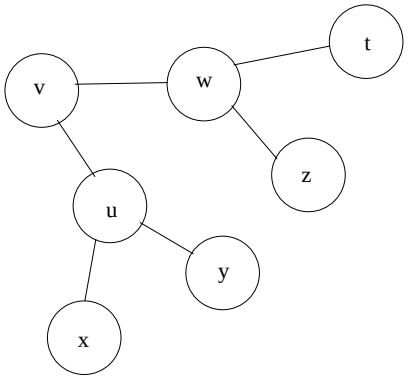

Figure 3.6: TTLShaping Example

$2^{(\tau-(\tau-1))}\hat{\rho}C = 2^1\hat{\rho}C$ queries with TTLs of $\tau - 1$, as per the TTLShaping DS. In addition, from the nodes two hops away (t, x, y and z), v should accept at most ${d_\alpha}^{(\tau-t)}\hat{\rho}C = 2^{(\tau-(\tau-2))}\hat{\rho}C = 2^2\hat{\rho}C = 4\hat{\rho}C$ queries. However, if y is malicious, and sends C queries, then v will receive $3\hat{\rho}C + 1C = (3\hat{\rho} + 1)C$ queries with TTLs of $\tau - 2$ (assuming that all other nodes in the network are good). While v might not be able to distinguish which of the $(3\hat{\rho} + 1)C$ are good, and which are malicious, v knows that it should process no more thatn $4\hat{\rho}C$ queries with TTLs of $\tau - 2$. When v uses a TTLShaping DS, it drops $(3\hat{\rho} + 1)C - 4\hat{\rho}C = (1 - \hat{\rho})C$ queries with TTL $\tau - 2$. TTLShaping does not offer any guarantees about whether or not malicious queries with TTL $\tau - 2$ will get dropped, but, as a heuristic, it can limit the amount of v's query bandwidth that gets used up by malicious nodes at varying distances away from it.

Note that the nodes in the network shown in Figure 3.6 have a constant degree of 2, which happens to correspond to the setting of d_α that we used in this example. We can use TTLShaping as a heuristic in networks that do not have a constant node degree, but we need to approximate a setting of d_α that is effective. A choice of d_α that is too high will not have the intended effect of setting an upper-bound on the amount of traffic processed from increasing distances, and a choice of d_α that is too low will end up dropping too many legitimate queries. We consider various choices for d_α, and its impact on minimizing query-floods in Section 3.3.1. Pseudo-code for the *TTLShaping* policy is shown in Figure 3.5.

3.1.3 Threat Model

While we considered a simple threat model in the last chapter in which an adversary may "seize" one malicious node, and set that node's $\rho = 1$, we extend our threat model in this chapter.

In our extended threat model, an adversary may present the following threats:

- *Number of Malicious Nodes.* An adversary may choose to replace many good nodes (instead of just one) with malicious nodes in the topology.

- *Positional Attack Model (PAM).* An adversary may choose to replace good nodes in particular positions in the topology. We call the method that the adversary uses to decide which positions in the topology to replace the *positional attack model (PAM).*

- *Flooding Strategy (FS).* An adversary may have malicious nodes choose to use different flooding strategies, and/or set ρs to values other than $\rho = 1$.

For evaluation purposes, we assume that all good nodes use the same IAS and DS policies, and all malicious nodes use the same FS.

We now describe a number of different PAMs and FSes that might be employed.

Positional Attack Model (PAM)

We study the following PAMs:

- *AttackRandom.* Malicious nodes are placed in random positions in the network. Good nodes are chosen replacement uniformly at random based on their node id, and independent of their particular position in the topology.

- *AttackHubs.* Malicious nodes are placed in the most highly connected positions in the network. We assume the adversary has complete knowledge of the topology of the network. The adversary orders all of the nodes in the network in descending order of their degree, and places malicious nodes in the positions with the highest degrees.

- *AttackHubNeighbors.* In the design of some P2P networks, the "hubs" (or most highly connected nodes) of the network may be more trusted, protected, or resilient to attacks than other nodes. In the AttackHubNeighbors PAM, the adversary is not be able to take over the hubs, but is able to place malicious nodes in positions that are directly connected to the hubs. The adversary orders all of the nodes in descending order of their degree. The adversary then starts with the highest degree node, and replaces good nodes in all the positions connected to the highest degree node with malicious nodes. The highest degree

node is then removed from consideration and the process is repeated for the next highest degree node.

Flooding Strategy (FS).

We conservatively assume that malicious nodes may have the capability to determine what algorithms the good nodes adjacent to them are using to manage query traffic. For instance, malicious nodes may know what IASes the good nodes they are connected to are using, and they may use that knowledge to determine how many queries to send each of their adjacent nodes to maximize the expected number of their queries that are accepted.

We also conservatively assume that malicious nodes have knowledge about the future. Specifically, if a good node u_g has neighbors $u_1, u_2, ..., u_k$, where one of those nodes u_m is malicious ($\exists m$ such that $1 \leq m \leq k$ and $u_m \in M$), then at time t, u_m has knowledge about how many and which queries each of u_g neighbor's will send it at time $t+1$. That is, at time t, u_m knows $O_{u_i,g}(t+1), 1 \leq i \leq k, i \neq m$. Node u_m may use this knowledge to determine which queries $O_{u_m,g}(t+1)$ to send to u_g to maximize the damage incurred.

In practice, a malicious node may not be able to realize these capabilities, but we choose to be conservative and study the "worst-case."

Based on how many queries u_g's neighbors are sending to it, u_m may have different options, or *flooding strategies (FSes)*, available to it as to how many malicious queries to transmit on each of its outgoing links. In this chapter, we study three FSes:

- *StandardFlood.* Malicious nodes are interested in minimizing the amount of remote work, and maximizing the distribution of damage. In some cases, it is to the malicious nodes' advantage to inject as many queries as possible in the hopes that many of their queries are processed instead of queries issued by good nodes. In this flooding strategy, a malicious node u_m sets $\rho_m = 1$, and broadcasts C_m queries to each of its outgoing links. That is, $O_{u_m,g}(t) = \rho_m C_m = C_m, \forall t$. Each of these queries has its TTL set to τ, the maximum TTL that nodes in the system are allowed to use.

- *LQF-IgnoreLastLink-Flood*. This FS is designed to specifically give the malicious nodes the best possible advantage in attacking good nodes that attempt to utilize the LQF-IgnoreLastLink IAS. In this attack model, we assume that malicious nodes have access to an "oracle" that tells them exactly how many queries are being sent and received on each link of every neighbor to which they are connected. Malicious nodes use this information to determine exactly how many queries to send to their neighbors such that their illegitimate queries will always be accepted, processed, and forwarded by neighboring nodes.

  Consider a small example in which a good "victim" node u_v has three other good nodes u_1, u_2, u_3, and one malicious node, u_m, connected to it. Assume that u_v's capacity $C_v = 12$, $\rho_v = \frac{1}{6}$, and u_v can process $C_v(1 - \rho) = 10$ units of remote work. Also, assume that u_1 is sending 4 queries, u_2 is sending 5 queries, and u_3 is sending 6 queries to u_v. If u_v uses an LQF-IgnoreLastLink IAS, queries from u_1 and u_2 will be processed by u_v. The queries from u_3 will be ignored because LQF-IgnoreLastLink IAS chooses to process queries from nodes that are sending the least queries first, and LQF-IgnoreLastLink ignores the queries sent on the last link that would cause the node to exceed its capacity. Note that if u_m has a capacity of $C_m = C_v = 12$, and u_m simply floods u_v with $\rho_m C_m = 1(12) = 12$ queries, then u_v will ignore all of u_m's queries.

  The key idea behind LQF-IgnoreLastLink-Flood is that if a malicious node knows exactly how many queries u_1, u_2, and u_3 are sending u_v, and the malicious node u_m would like to ensure that queries that it sends the largest number of queries that will get processed, it can send one less query than u_2 is sending. When u_m uses LQF-IgnoreLastLink-Flood instead of StandardFlood, it sends 4 queries, and u_m and u_1's queries will be accepted for processing while u_2 and u_3's queries will be ignored.

  In a real-world system, malicious nodes may be able to do passive monitoring to determine how many queries other good ndoes are sending to victim nodes by running a packet sniffer on a compromised host and viewing all Ethernet traffic being sent to or from neighboring hosts on a LAN. Note that in the

for each good node u_g to which u_m is connected **do**
  $n \leftarrow 0$
  $C_r \leftarrow C_{u_g}$
  while $C_r > 0$ **do**
    Choose j such that $|I_{j,u_g}(t+1)|$ is minimum. {least queries first}
    {if queries from link j will be accepted}
    if $C_r - |I_{j,u_g}(t+1)| > 0$ **then**
      $n \leftarrow |I_{j,u_g}(t+1)| - 1$
      $C_r \leftarrow C_r - |I_{j,u_g}(t+1)|$
    end if
  end while
  $O_{m,u_g}(t) \leftarrow createQueries(n)$ { creates n queries }
end for

Figure 3.7: LQF-IgnoreLastLink Flood

LQF-IgnoreLastLink flooding strategy, the malicious node is not required to broadcast an equal number of queries on each of its outgoing links.

Pseudo-code that malicious nodes use to determine $O_{u_m,u_g}(t+1)$ is shown in Figure 3.7.

- *PA-Flood*. When good nodes use the PA IAS, exactly which links good nodes choose to accept queries from is not deterministic. As such, regardless of how many queries a malicious node sends to a good node that is using PA IAS, the malicious node cannot be absolutely sure how many of its queries will be accepted by the good node. However, if a malicious node sends a good node just one query while other nodes are sending tens or hundreds of queries to the good node, the malicious node's query will get processed with high probability. If a malicious node sends thousands of queries while other nodes are sending tens or hundereds, the malicious ndoe's query will get processed with a relatively low probability. Malicious nodes that would like to have a maximum number of their queries processed by a good node can experiment with different values of ρ_m to determine $\hat{\rho_m}$, the value of ρ_m that maximizes the expected number of their queries that will probabilistically be accepted by the good node. A malicious node sets $\rho_m = \hat{\rho_m}$ in a PA-Flood.

- *TTLShaping-Flood*. In this attack model, a malicious node may be aware that a good node is using a TTLShaping DS to mitigate floods. To counter the TTLShaping DS, the malicious node injects query traffic that has a TTL distribution that does not exceed the upper bounds that the TTLShaping DS checks for. Specifically, we assume the malicious node knows the choice of d_α that good nodes use, and the malicious node sends exactly ${d_\alpha}^{(\tau-h)}\hat{\rho}C$ queries with TTL h for each $h \in [1, \tau]$ to each good node to which it is connected. While these queries will not be dropped by the good node based on TTLShaping DS, they will not have as detrimental an effect as a flood of queries in which each query has a TTL of τ.

3.1.4 Metrics

We use two key metrics to evaluate the performance of IASes and DSes: remote work and damage distribution.

Remote Work (RW)

While we did introduce the concept of remote work in the last chapter, we did not define it formally, and mainly focused on the metrics of service and damage. In this section, we review the concept of remote work, and provide formal definitions for instantaneous and steady-state remote work.

Informally, the remote work (RW) done by a system is the sum of all of the remote work that is done by each of the good nodes in the system, and is a measure of the overall throughput with which queries are processed. Malicious nodes do not process queries on behalf of others, and they do not contribute to remote work.

When a good node issues a query, and that query is processed by some other good node, one unit of remote work is done. However, when a good node processes a query that was injected by a malicious node, the good node's query bandwidth is wasted, since the work done is "useless." In a real system, it may be difficult to tell whether or not a particular query was admitted by a good node or injected by a malicious node, but it is important to know the true origin of a query if we are

to compute remote work. As such, we assume the existence of an "oracle" function, Ω, that is able to distinguish between queries admitted by good nodes, and queries injected by malicious nodes (irrespective of what the $q.o$ field purports, as it may have been spoofed). The function Ω takes as input a set of queries, and returns the subset of queries that were admitted by good nodes. We use the oracle function in the following definition of Instantaneous Remote Work.

Definition 3.1.1 Instantaneous Remote Work for Node j. $RW_j(t) = \Sigma_{v \in V} |\Omega(I_{v,j}(t))|$ *where* $\Omega : S \rightarrow S$ *is an "oracle" function that takes as input a set of arbitrary queries and returns the subset of those queries that were admitted by good nodes.*

We now define RW by building on the previous definition, and the following definition for Steady-State Remote Work.

Definition 3.1.2 Steady-State Remote Work for Node j. $RW_j = RW_j(t_0)$, *where* t_0 *is a time such that* $|RW_j(t) - RW_j(t_0)| < \varepsilon, \forall t > t_0$ *for some small* ε, *and non-negative integer,* t_0. *Note that* RW_j *is only well-defined if* t_0 *exists.*

Now, we define the Total Steady-State Remote Work for all nodes in the network, which hereafter will be referred to as simply remote work (RW).

Definition 3.1.3 Total Steady-State Remote Work (RW). $RW = \Sigma_{j \in G} RW_j$. *(Recall that* G *is the set of good nodes in the network.)*

When malicious nodes flood the system with queries, good nodes may process and forward malicious nodes' queries instead of good ones, and this will be reflected by a drop in RW.

There are several advantages of using RW as a performance measure, as compared to other measures such as the number of search results received. Firstly, RW is simple to compute. We do not need to model how files are distributed across the nodes in the network, the popularity distribution with which particular queries are asked, or the probability that a particular node will have a particular file. Secondly, RW is protocol independent and highly correlated with the number of search results– the

more remote nodes that process a query, the more search results will be generated. While RW does not directly capture how many search results a good node obtains for its queries, it does allow us to study the flood-tolerance of the system independent of the search protocol. Fourth, a loss in RW can effectively capture the impact of a query-flood. In particular, if a query-flood prevents a query from traveling from one node to another node elsewhere in the network, this event will be captured by a drop in remote work. Finally, if we measure the performance of the system by RW, it will encourage good nodes to process work from remote nodes instead of just processing queries from their local peers. If each node only processed queries from their local peers, the network would function as a disjoint set of many client-server systems where the node is the server and the local peers are the clients. As a result, RW also serves as a measure of the utility gained from having a peer-to-peer system instead of a disjoint set of client-server systems.

On the other hand, using RW as a performance metric does have a disadvantage– it does not give us information about how particular nodes suffer during a query-flood attack. For instance, when malicious nodes issue a flood, the nodes closest to the malicious nodes may be affected more so than nodes farther away. Alternatively, the use of a particular IAS may uniformly even out the extent to which nodes at different distances from malicious nodes are affected. The RW metric would not allow us to see how the effects of the flood are distributed across the nodes in the network. As such, we use a second metric called damage distribution (that we will use in Section 3.2.2) to understand these types of effects.

In many of our simulations that involve varying numbers of malicious nodes, we use a metric called Remote Work Utilization (RWU) instead of RW to compare situations where there are differing numbers of malicious nodes. RWU is simply the ratio of RW to the amount of total work that is possible. Consider a network with N good nodes. If all the good nodes have the same capacity, the maximum RW possible might be βNC for such a network, for some constant, $0 \leq \beta \leq 1$. However, if one of the nodes is malicious, the maximum RW that we can expect might be $\beta(N-1)C$ since the malicious node cannot be expected to do any work As such, RWU is defined as follows to capture the fact that we do not expect malicious nodes to do work.

Definition 3.1.4 Remote Work Utilization (RWU).

$$RWU = \frac{\Sigma_{j \in G} RW_j}{\Sigma_{j \in G} C_j}$$

where G is the set of good nodes in the network.

For example, consider a complete network containing $N = 3$ good nodes where $C = C_1 = C_2 = C_3$. It can be calculated that such a network is able to accomplish a RW of $\frac{3}{2}C$ if each node sets $\rho = \frac{1}{2}$. The RWU is therefore $\frac{\frac{3}{2}C}{3C} = \frac{1}{2}$. However, if one of the nodes is malicious, and the good nodes can use an oracle ignore query traffic injected by the malicious node, the RW is C. Note that the RWU is still $\frac{C}{2C} = \frac{1}{2}$. In both cases, RWU=$\frac{1}{2}$. If the good nodes are able to use an oracle to tell which nodes are malicious, they are able to preserve RWU, even though we expect a loss in RW. RWU allows us to compare how well good nodes are able to perform regardless of the number of malicious nodes in the network.

3.2 Results and Discussion

In this section, we discuss the results of various discrete-event simulations that we conducted based on the Gnutella traffic model described in Section 2.1. After describing some details regarding the topology we studied, and our experimental setup, we answer the following research questions using results obtained from our simulations:

- What is the effect of varying τ, the maximum TTL allowed, on RW? Do large settings of τ give malicious nodes an advantage? Can IASes be used to control floods in systems where high τs are used?
- How are RW and damage distribution impacted as the number of malicious nodes increases? What IASes (if any) are effective in reducing the malicious nodes' impact on RW and damage distribution?
- How much more significant do query-flood attacks become when malicious nodes use different FSes and PAMs? How vulnerable are IASes to different flooding strategies and PAMs?

- Do we expect that the results obtained from the simulations run to answer the questions above will scale to larger-sized networks?

In the next section, we describe the setup for various simulations we conducted to answer the above research questions.

3.2.1 Simulation Setup

We start by describing the Gnutella topology that we used for most of our simulations, and we then discuss some of our simulation parameters.

Topology

Our results in the following sections are run on a snapshot of a Gnutella topology obtained in May 2001 by Saroiu et al. [178]. The largest snapshot with *complete topology information* gathered from that study was 2,433 nodes, and the largest connected component in that snapshot was 1,787 nodes [3]. We name this largest connected component Gnutella-1787, and use it in our simulations.

Gnutella-1787 was gathered before the deployment of supernodes (also known as Ultrapeers) in Gnutella. At the time, all the nodes in the network were considered equal, regular peers. While we would prefer to use topology data from today's existing network, it is not feasible to gather such data because in the current version of the Gnutella protocol, nodes do not report their presence if they are satisfied with the number of connections they have.

However, supernodes can use the same basic technique for topology construction as did the regular nodes in the network in May 2001. That is, supernodes can contact host caches available at central locations and attempt to connect to IP addresses from the cache, just as did regular nodes in the past. It is likely that supernodes in such a network form topologies similar to the topologies that regular nodes formed in the past. As such, we expect that our results accurately reflects trends and trade-offs

[3] Saroiu et al.'s study discusses measurements on networks with more nodes, but complete topology data was not gathered in those cases.

that may be seen in Gnutella networks where supernodes use the same bootstrap mechanism that regular nodes used to use.

Some studies claim that Gnutella networks have power-law topologies. Sariou et al. in [178] claim that Gnutella networks are similar to power law graphs with an average connectivity of $\alpha = 2.3$. We also ran simulations on a synthesized power-law topology with $\alpha = 2.3$ (see Section 3.2.2), and found the same trends that we did for Gnutella-1787.

Capacity and Policies

For simplicity, we assigned all of the nodes in the network the same capacity, $C = 10^6$. Hence, the maximum amount of total work, which includes local plus remote work, that can take place in an N-node network in one time-step is $10^6 N$. To keep our work figures reasonable and meaningful, however, we normalize all our results by dividing by C such that the maximum total work is N, but all simulations are carried out with a "precision" of $C = 10^6$. Therefore, for instance, the maximum TW that can be achieved in one time-step in a simulation of Gnutella-1787 is 1787.

We observe that in our current model, malicious nodes do not collude in any meaningful way. When good nodes employ some policies, there may not be any reasonable way for malicious nodes to collude. For instance, if good nodes are using a Fractional IAS, the best attack that malicious nodes can conduct is one in which they do a StandardFlood in the most highly connected positions of the network. On the other hand, if good nodes use an IAS such as LQF-IgnoreLastLink or PA, it may be possible for malicious nodes to construct more effective attacks by colluding. For instance, if a good node uses LQF-IgnoreLastLink, and malicious nodes can use LQF-IgnoreLastLink-Flood FS, then it will be beneficial for two malicious nodes to coordinate such that they do not flood the same good node. By doing so, the malicious nodes' traffic does not compete against each other when the good node is deciding which link's queries to ignore. We leave studying colluding malicious nodes to future work.

Reservation Ratios

In our simulations, we have all good nodes set $\rho = \hat{\rho}$ under the assumption that they altruistically want to maximize RW.

Maximum TTL

With the exception of Section 3.2.2, in which study the effects of varying the maximum TTL used in the network, we conduct most of our simulations with $\tau = 7$, as is used in real Gnutella network deployments.

Steady-State

While we mathematically proved that steady-state occurs after τ time steps for Null IAS and PreferHighTTL in [194], we empirically found that steady-state was achieved within 2τ time steps for *all* of the IASes and DSes we experimented with in this chapter. As such, we ran all our simulations for 2τ time steps. All simulations were confirmed to achieve steady-state.

3.2.2 Basic Results

Malicious nodes can incur significant damage to the network by mounting query-floods when Null IAS and PreferHighTTL DS is used by good nodes. In the following sections, we experiment with various IASes and DSes. Use of some combinations of IAS and DS policies may result in the network being able to process only a sub-optimal amount of RW, but do significantly lessen the number of malicious queries that are processed by good nodes. An ideal policy would process all of the good nodes' queries and none of the malicious nodes' queries, and would result in the maximum RW possible. We evaluate how well various real policies come to this ideal.

We initially assume that malicious nodes are using StandardFlood FS and the AttackRandom PAM.

In Section 3.2.3 and 3.2.4, we consider more powerful malicious nodes by studying the effect of varying FS and the PAM.

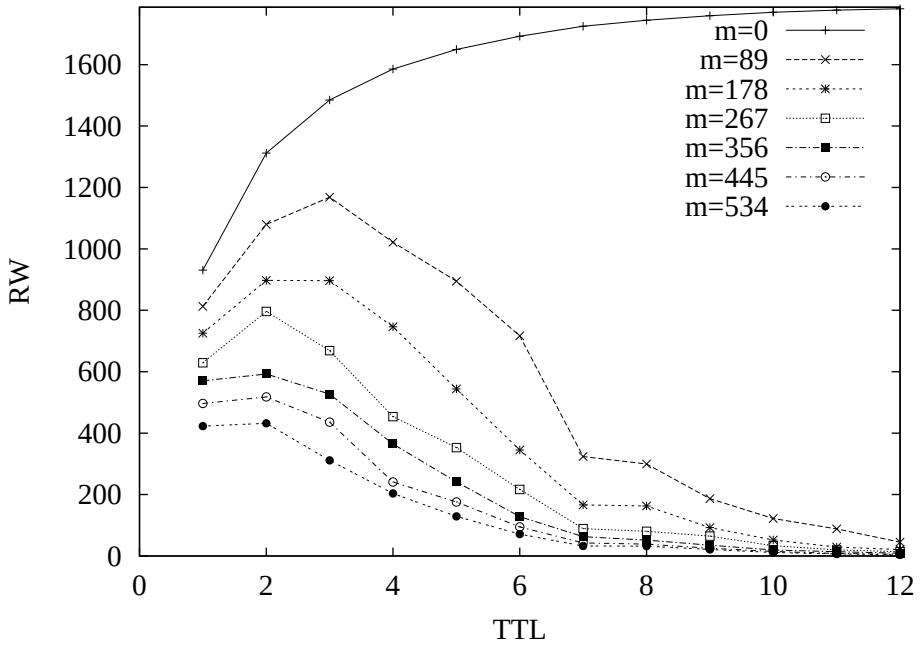

Figure 3.8: RW vs. TTL with Null IAS

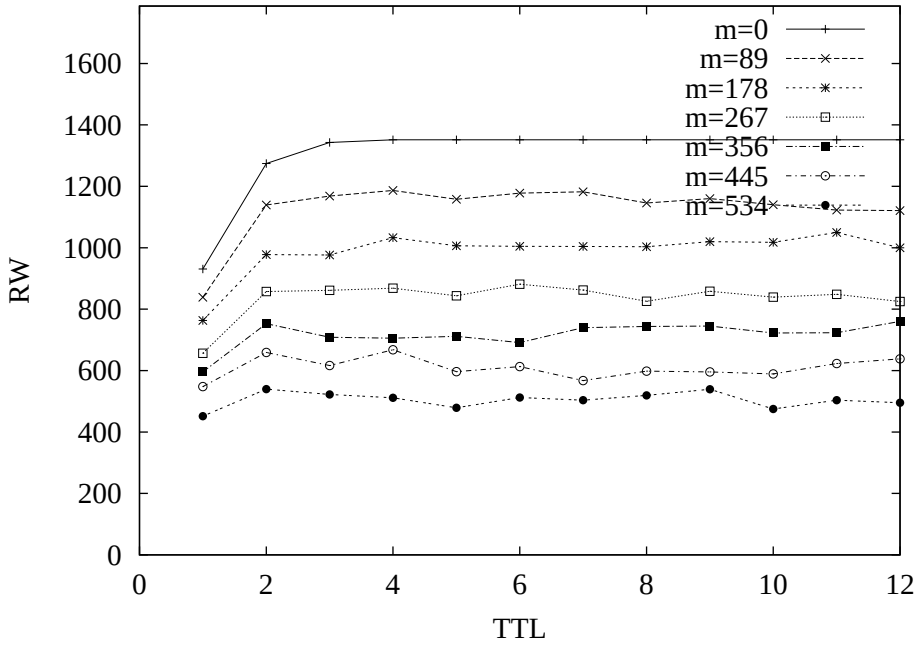

Figure 3.9: RW vs. TTL with Fractional IAS

IAS	*Optimal TTL for Gnutella-1787*	*Maxiumum RW*
Null	3	880
Fractional	N/A	1006
Fractional-Spillover	3	1096
LQF-IgnoreLastLink	N/A	917
Probabilistic Accept	N/A	1220

Table 3.1: Optimal TTLs for Various IASes in Gnutella-1787

Impact of TTLs

Fractional, LQF-IgnoreLastLink, and PA IASes mitigate query-floods when high settings of τ (maximum TTL) are used. Small settings for τ can also be used to mitigate query floods without sacrificing too much RW.

Practical implementations of nodes in the Gnutella network typically stamp newly admitted queries with a maximum TTL of $\tau = 7$, a value that was decided upon using empirical experimentation. In this section, we describe the results of simulations in which we varied τ to understand what is the effect of τ on RW, and how it affects the impact of query-floods.

When high τs are used in a network, malicious queries will travel farther and will traverse more nodes in the network (as do good queries). However, in a query-flood, malicious nodes are injecting many more queries than are good nodes, and, as a result, high τs give malicious nodes extra power in displacing legitimate queries from being processed.

As larger and larger τs are used, we find that IASes can be used to eliminate the extra power that malicious nodes would otherwise gain. In Figures 3.8, 3.9, and 3.10, we simulated the impact of using increasing τs in the Gnutella-1787 topology, and their impact on remote work. Figure 3.8 shows the outcome of our simulations when a Null IAS is used. When there are no malicious nodes present (m=0), as larger and larger τs are used, all queries reach almost all nodes in the network, resulting in a steady-state RW of 1782 at τ=12. However, when malicious nodes are present, as nodes increase the τs with which they stamp their queries, the malicious nodes' queries are able to displace more and more legitimate queries resulting in less and less

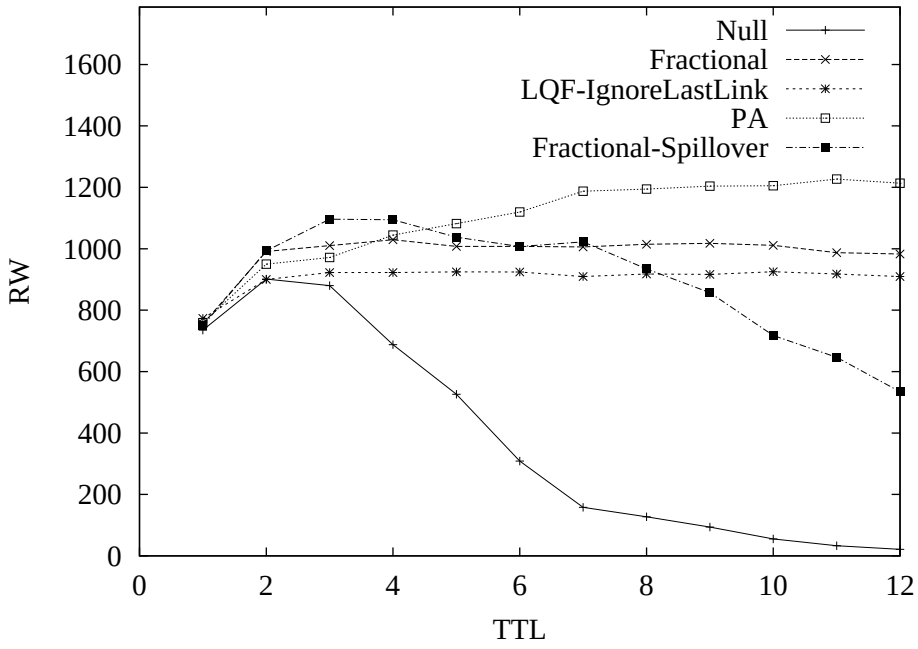

Figure 3.10: RW vs. TTL for Various IASes (10 Percent Malicious Nodes)

RW. For instance, if just 5 percent of the nodes are randomly chosen to be malicious ($m = 89$), at a τ of 12, the RW is merely 40.

Figure 3.9 shows how a Fractional IAS is able to mitigate the power of malicious nodes, even as increasing τs are used. At a τ of 4, the Fractional policy is able to stabilize the number of nodes to which malicious queries can spread. As τ increases beyond 4, RW stays constant for a given number of malicious nodes. For example, when 5 percent of the nodes are malicious ($m = 89$), and τs of 4 are used, the RW is approximately 1165. Even as τs greater than 4 are used, RW is still approximately 1165. The reason for this is that for every good node that is an additional hop away from a malicious node, a significant fraction of the malicious node's queries are dropped, and after four hops, relatively few of the malicious node's queries survive the drop process and are processed at good nodes. On the other hand, because good nodes are admitting relatively fewer queries, many of the queries admitted by good nodes do get processed and are not dropped due to the Fractional IAS employed at each node.

Note that using the Fractional IAS itself does result in some RW lost due to some good queries being dropped (even without the presence of malicious nodes). In Figure 3.8, when no malicious nodes are present, we can see that the RW is 1725 when τ=7. However, under the same conditions when the Fractional IAS is used, Figure 3.9 shows that the RW is only 1350. In the best-case scenario where no malicious nodes are present, using Fractional IAS has the downside that $\frac{1725-1350}{1725}$=22 percent of the RW is sacrificed.

By the same token, Fractional IAS can significantly pay off (as compared to using Null IAS) when a non-trivial number of malicious nodes are present. For instance, from Figure 3.8, we can see that when 10 percent of the nodes are malicious ($m = 178$), and Null IAS is used the RW is only 158 (at τ=7). From Figure 3.9, we can see that RW is over six times as much (at 1007) when Fractional IAS is used. Using Fractional IAS in a scenario where malicious nodes are present can increase RW by a factor of over six and drastically reduce the impact of query floods.

Figure 3.10 is similar to Figures 3.8 and 3.9, in that it plots RW vs. τ, except it does so for various IASes with 10 percent malicious nodes. Figure 3.10 shows that PA, LQF-IgnoreLastLink, and Fractional IAS are effective in acheiving increasing RW with increased τs when malicious nodes are present. If Null or Fractional-Spillover IAS is used, we can see from the figure that using a τ of greater than 4 results in decreasing RW. While the LQF-IgnoreLastLink, Fractional, and PA IASes are suitable for use with large τs, Null IAS and Fractional-Spillover are not so because significant numbers of malicious queries spillover and are rebroadcast by good nodes.

IAS Resilience to Malicious Nodes

LQF-IgnoreLastLink and PA IASes result in the best resilience to increasing numbers of malicious nodes (of all the IASes we considered under the StandardFlood FS).

Figure 3.11 shows that if there are no malicious nodes in the network, Fractional-Spillover IAS results in approximately 50 percent more RW than Fractional, and 10 percent more RW than PA. If even two or three percent of the nodes in the network are malicious however, we see that the LQF-IgnoreLastLink and PA IASes provide a higher RWU than other IASes. The LQF-IgnoreLastLink and PA IASes were designed

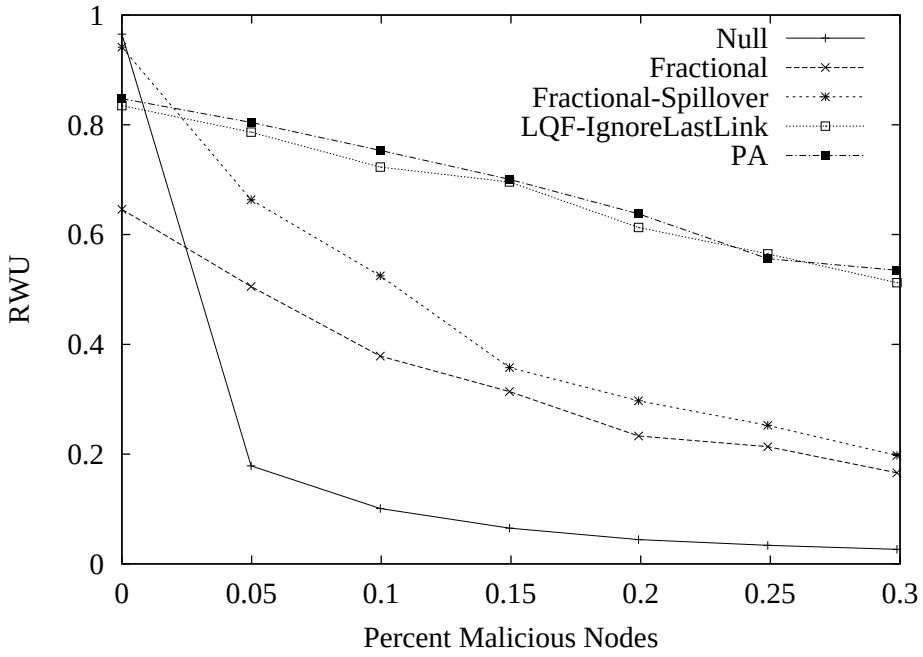

Figure 3.11: RWU vs. Fraction of Malicious Nodes for Various IASes (Gnutella-1787)

to distinguish queries arriving from malicious nodes based upon the number of queries that arrive on a particular link, and under a StandardFlood FS, we see that they are successful in doing so. While both LQF-IgnoreLastLink and PA IAS result in similar RWU, we will see in the next subsection that they have different damage distribution characteristics, and that they have different levels of vulnerability to FSes that target them in particular (see Section 3.2.3).

Sensitivity to Topologies

In this subsection, we assess the sensitivity of the IASes we study to different graph topologies. Two topologies that we consider in addition to Gnutella-1787 are a synthetic power-law topology (generated using the PLOD algorithm suggested in [151]) with the same outdegree as Gnutella-1787, and a 2000 node regular, random graph topology with outdegree 10. In both cases, we find that the basic trends that we observe with respect to RWU for different IASes is independent of the topologies we

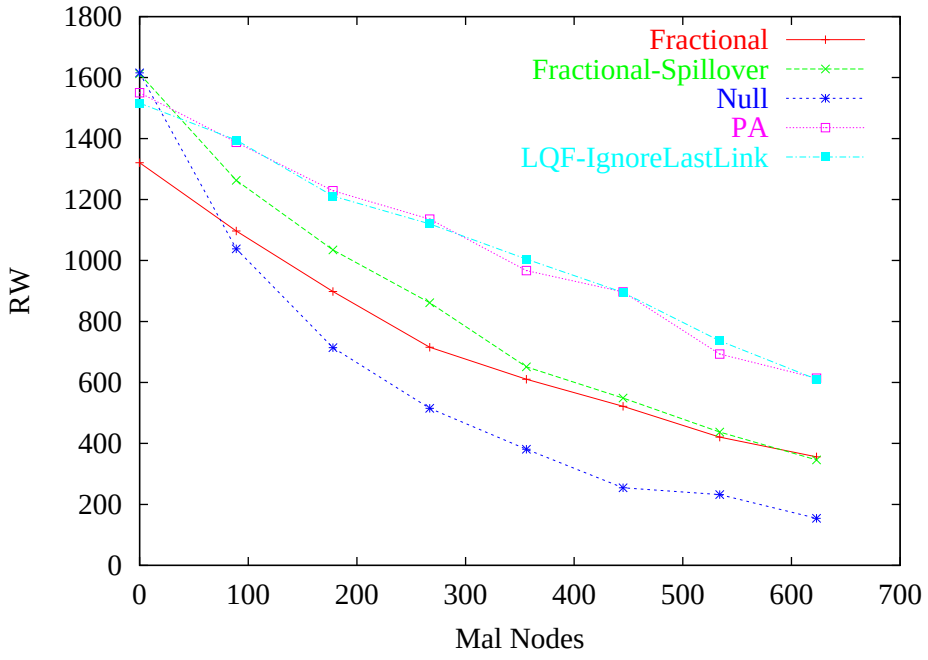

Figure 3.12: RWU vs. Fraction of Malicious Nodes for Various IASes (Synthetic Power-Law Topology)

experimented with.

Figure 3.12 shows the results of a simulation in which we compared the RW that resulted by using various IASes while introducing growing numbers of malicious nodes into the synthetic power-law topology. From the figure, we can see that the curves corresponding to the different IASes have the same shapes as in Figure 3.11 , and the relative order of performance of the IASes is still the same.

Damage Distribution

PA IAS does the best (out of the IASes we considered) to limit distribution of damage to the fewest nodes in the network.

As we mentioned in Section 3.1.4, one of the disadvantages of the RW and RWU metrics is that they do not give us information about how particular nodes suffer during a query-flood attack. For each node in the topology, we can calculate damage to understand how individual nodes are impacted by a query flood attack. However, we are also interested in understanding how damage is spread out throughout the topology, as opposed to just looking at damage at individual nodes.

In this subsection, we measure damage for each individual node in the topology, and plot the damage that individual nodes incur in rank order (from most damage to least damage). Our plots help us assess how far damage spreads using various IASes. The ideal damage plot (that can be obtained by using an oracle) would show a damage of 1 for m malicious nodes, and a damage of 0 for all other nodes. For example, an ideal plot for our 1787-node network in which 10 percent of the nodes are malicious is shown in Figure 3.13. In this ideal plot, the malicious nodes are not able to have any impact on the service received by each of the good nodes because the good nodes use an oracle to perfectly distinguish between good and malicious queries.

Figure 3.14 plots damage when there are 10 percent ($m = 178$) malicious nodes in the most highly connected positions of the network. Since there are 178 malicious nodes in the network, we expect that the damage for at least the first 178 nodes (in rank order) will be 1.0, which is indeed the case in the figure. After the 178th node, different IASes result in different damage distributions.

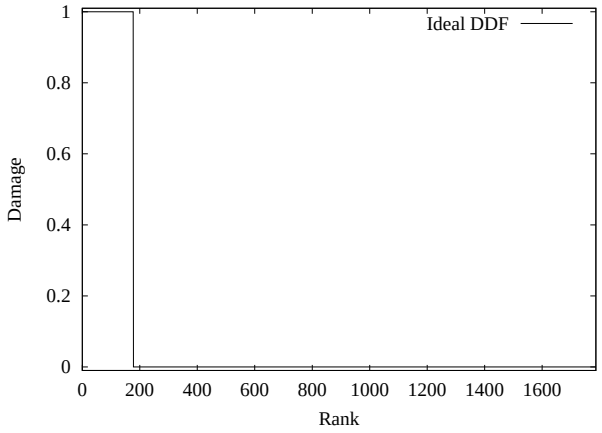

Figure 3.13: Ideal Damage: Damage vs. Rank for 10 percent malicious nodes

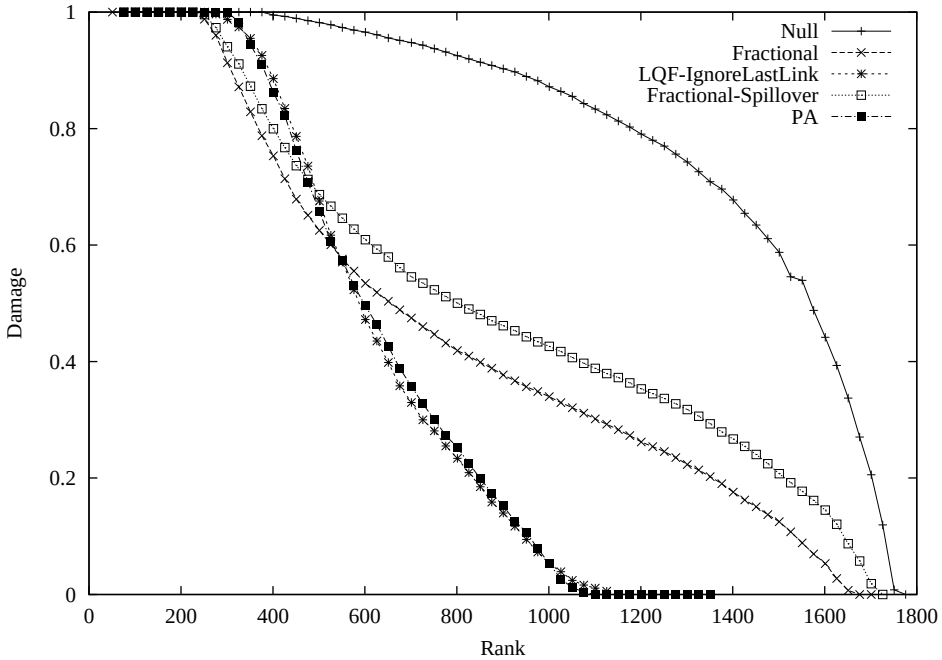

Figure 3.14: Damage Distribution for Various IASes, 10 percent malicious nodes, AttackRandom PAM

From the figure, we can see that Null IAS allows damage to spread to every node in the graph. The Fractional and Fractional-Spillover IASes spare just a few nodes from incurring any damage at all. In general, as compared to Null IAS, the Fractional and Fractional-Spillover IASes reduce the amount of damage that nodes incur, but more than half of the nodes still incur more than 50 percent damage. Note that the damage for Fractional-Spillover is slightly worse than for Fractional due to the fact malicious queries can take advantage of unused link quotas. In Figure 3.11, we can see that Fractional-Spillover results in 20 percent more RWU than Fractional at the cost of slightly more damage distributed throughout the network when there are 10 percent malicious nodes. From the two figures, we can therefore conclude that most of the queries that spillover when 10 percent of the most highly connected nodes are malicious are queries issued by good nodes.

3.2.3 Flooding Strategies

In the previous section, we evaulated IASes and DSes using the StandardFlood FS, in which malicious nodes set $\rho = 1$. We found that if malicious nodes use a StandardFlood FS, their attack can be contained by taking advantage of IASes such as PA and LQF-IgnoreLastLink. However, if malicious nodes are aware that good nodes are using such policies, they may explicitly target them using a PA-Flood or LQF-IgnoreLastLink-Flood, respectively. In this section, we study cases in which malicious nodes know what IASes and DSes good nodes are using, and they choose a FS that targets the good nodes' policies explicitly.

We note that it only makes sense for malicious nodes to use certain FSes when good nodes uses certain IASes or DSes. For instance, if good nodes use Fractional or Fractional-Spillover IAS, then the best attack that malicious nodes can mount is to use StandardFlood FS ($\rho_m = 1$). If malicious nodes set $\rho_m < 1$ when Fractional or Fractional-Spillover IAS are used by good nodes, then their attack will not minimize RWU or maximize damage. On the other hand, if good nodes use LQF-IgnoreLastLink IAS, then the best attack that malicious nodes can conduct is to use LQF-IgnoreLastLink-Flood FS to specifically target the good nodes. Since

Good Nodes: IAS	*Malicious Nodes: Best FS*
Null, Fractional, Fractional-Spillover	StandardFlood
LQF-IgnoreLastLink	LQF-IgnoreLastLink-Flood
PA	PA-Flood

Table 3.2: Best FSes for Malicious Nodes to Target Various IASes

Good Nodes: DS	*Malicious Nodes: Best FS*
PreferHighTTL	StandardFlood
TTLShaping	TTLShaping-Flood

Table 3.3: Best FSes for Malicious Nodes to Target Various DSes

good nodes accept queries from nodes that send the fewest queries first under a LQF-IgnoreLastLink IAS, malicious nodes' queries may not be accepted at all if they send too many. Under a LQF-IgnoreLastLink-Flood FS, malicious nodes send fewer queries to good nodes so that their queries will be accepted, and will deny service to legitimate queries sent by good nodes.

Also, it would not make sense for malicious nodes to use LQF-IgnoreLastLink-Flood FS when good nodes used, say, Fractional IAS because they could deny more service by sending out more queries with a StandardFlood FS. In Tables 3.2 and 3.3, we show the best flooding strategies for malicious nodes to use to maximize damage and minimize RWU when good nodes use various IAS and DS policies.

LQF-IgnoreLastLink-Flood

LQF-IgnoreLastLink performs comparably to Fractional when under a targeted attack by malicious nodes.

When good nodes use the LQF-IgnoreLastLink IAS, malicious nodes can use the LQF-IgnoreLastLink-Flood FS to send just enough queries so that their queries will

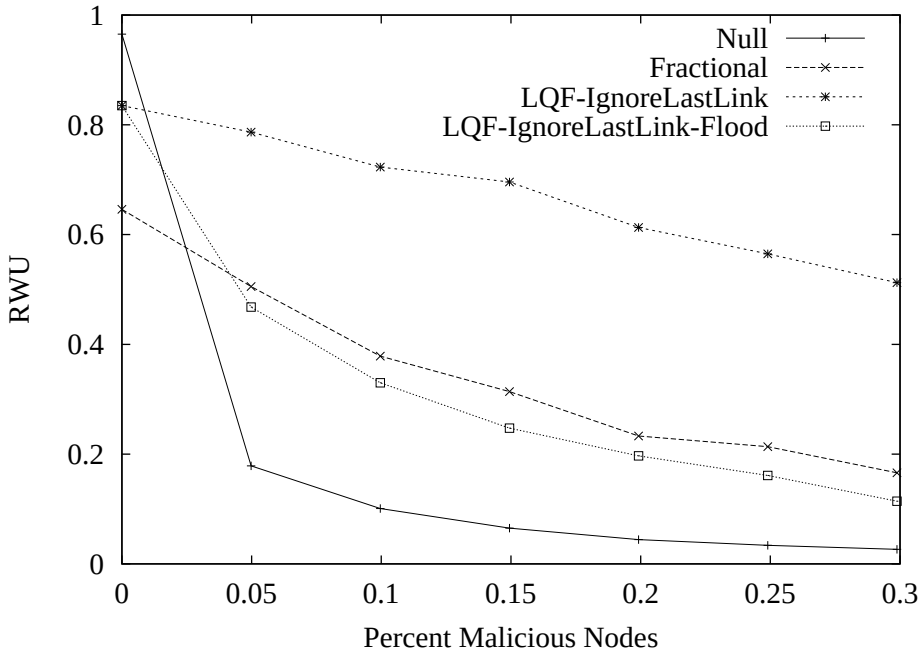

Figure 3.15: RWU vs. Fraction of Malicious Nodes for LQF-IgnoreLastLink

be accepted by the good node, and other good nodes' queries will be ignored. Figure 3.15 plots RWU versus the fraction of the nodes that are malicious for the LQF-IgnoreLastLink IAS, for both cases when malicious nodes use LQF-IgnoreLastLink-Flood FS and when malicious nodes use StandardFlood FS. the line marked "LQF-IgnoreLastLink-Flood" plots RWU when malicious nodes use LQF-IgnoreLastLink-Flood FS, and the line marked "LQF-IgnoreLastLink" plots RWU when malicious nodes use the StandardFlood FS. We observe that while the LQF-IgnoreLastLink IAS acheives high RWU under a StandardFlood attack, it does not do as well under a targeted LQF-IgnoreLastLink-Flood attack. For instance, if 5 percent of the nodes are malicious, RWU drops from just under 80 percent with StandardFlood to just under 50 percent with LQF-IgnoreLastLink-Flood.

However, to put the performance of LQF-IgnoreLastLink IAS under the targeted LQF-IgnoreLastLink-Flood attack in perspective, we also plot the performance of Fractional IAS in Figure 3.15. The best flooding strategy for malicious nodes to use when good nodes use Fractional IAS is StandardFlood. As such, we have also

plotted the RWU for Fractional IAS in Figure 3.15 when malicious nodes issue a StandardFlood. We notice that the performance of LQF-IgnoreLastLink IAS under a targeted LQF-IgnoreLastLink-Flood attack is similar to the performance of Fractional IAS under a StandardFlood attack.

It is therefore better for good nodes to use LQF-IgnoreLastLink IAS instead of Fractional because 1) LQF-IgnoreLastLink performs comparably to Fractional under a targeted attack, 2) it "raises the bar" for the actions that malicious nodes are required to conduct to perform a query-flood, and 3) in the average case that not all malicious nodes will be able to perform LQF-IgnoreLastLink-Flood, a significant amount of RWU can be gained over and above what can be acheived with Fractional IAS.

PA-Flood

PA IAS can be defeated when malicious nodes rate-limit their query-floods. Fractional-Spillover is the best strategy (of the IASes we considered) for good nodes to use if malicious nodes rate-limit floods.

In Section 3.2.2, we found that PA IAS is able to successfully mitigate query-floods when malicious nodes inject queries with $\rho = 1$. However, when malicious nodes do not inject queries "at full capacity," they are able to rob the network of a large fraction of remote work.

For the purposes of our evaluation of the PA-Flood FS, we assume that all malicious nodes use the same value of ρ_m, and we first determine the value of ρ_m that minimizes remote work. Figure 3.16 shows the results of a simulation that measures how RW varies as malicious nodes use a ρ_m that varies from 0 to 1. Each curve in the figure measures remote work for different IASes that good nodes use.

The figure allows us to see how different IASes perform when malicious nodes set $\rho_m < 1$. Instead of pumping out queries at maximum capacity, malicious nodes *limit the rate* at which they inject queries.

From the figure, we first notice that when a Null IAS is used, RW decreases steadily as the malicious nodes increase their ratios from 0 to 1. The first data point on this curve plots RW when malicious nodes set $\rho_m = 0.01$ and are injecting slightly

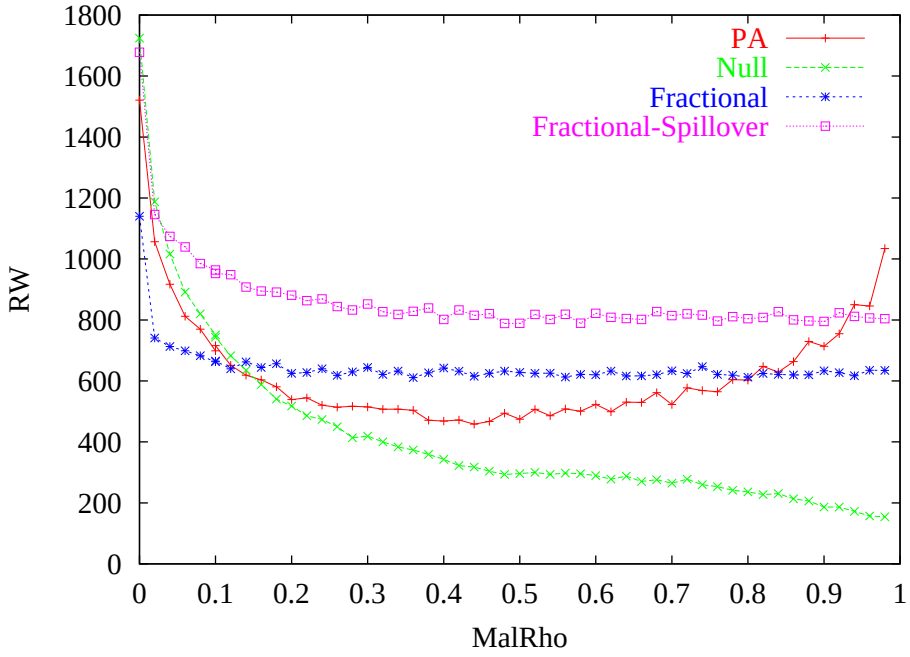

Figure 3.16: RW vs. Malicious Rho for Various IASes

fewer queries than good nodes that set $\rho = \hat{\rho}$. At that point, RW is approximately 1050. As the malicious nodes start injecting more queries, RW steadily decreases when good nodes use Null IAS until at $\rho_m = 1$, the malicious nodes have robbed the network of over 85 percent of the RW possible given their presence in the network. As before, good nodes must use another IAS if they are to mitigate the flood.

However, from the results in Figure 3.16, we can see that PA IAS is not effective against the query-flood if malicious nodes set their ρ_m's appropriately. In particular, if malicious nodes set ρ to a value between 0.1 and 0.88, less RW is done by the network when good nodes use PA IAS than with the other IASes plotted. At those settings of ρ_m, malicious nodes are injecting an amount of traffic that is less than the amount of traffic that good nodes are admitting. While each good node is only admitting $\hat{\rho}C$ local work, their local work is being sent to adjacent good nodes together with remote work that they are forwarding on behalf of other nodes– this total amount of local and remote work significantly exceeds the amount of work injected by malicious nodes that use small ρ_m's. As a result, the probability that the malicious traffic is

accepted by good nodes is greater than the probability that good traffic is accepted.

The value of ρ_m that maximizes the effectiveness of the malicious nodes' attack in this particular simulation is $\hat{\rho_m}$=0.43, but settings in the range 0.2 to 0.6 cause a roughly equivalent effect. PA IAS is not able to stand up to query-floods in which malicious nodes rate-limit the queries they inject.

Good nodes should instead use a Fractional-Spillover IAS to maximize RW when malicious nodes rate-limit their queries. From Figure 3.16, we can see that as malicious nodes start increasing ρ_m from 0.01 to 0.3, RW does decrease from just under 1200 to 750. However, as the malicious ρ continues to increase, the amount of service provided to malicious nodes steadies out, and an RW of approximately 750 is maintained. So while $\frac{1200-750}{1200}$=46 percent of the RW is lost even when good nodes use Fractional-Spillover IAS, we should keep in mind that the goal of the IAS is to contain the severity of an attack while it is in progress. In the case that malicious nodes have taken over 10 percent of the most highly connected nodes in the network, we are able to still provide over half of the throughput (remote work) as when there are no malicious nodes in the network. In the meantime, our Fractional-Spillover IAS is expected to be used in conjunction with detection protocols that work to determine exactly which nodes are malicious such that they can be disconnected from the network.

While Fractional-Spillover IAS is more effective than PA IAS in dealing with malicious nodes that rate-limit their floods, PA IAS is effective in dealing with "faulty" nodes. A faulty node is a good node that may have simply gone into an infinite loop, and is spending its processing capacity sending legitimate queries with a high value of ρ. From Figure 3.16, we observe that if malicious (or faulty) nodes send queries with $\rho > 0.9$, PA IAS results in higher RWU than Fractional, and when $\rho > 0.93$, PA IAS results in higher RWU than Fractional-Spillover. Hence, PA IAS is more effective than Fractional or Fractional-Spillover IAS in containing the effects of "faulty" nodes.

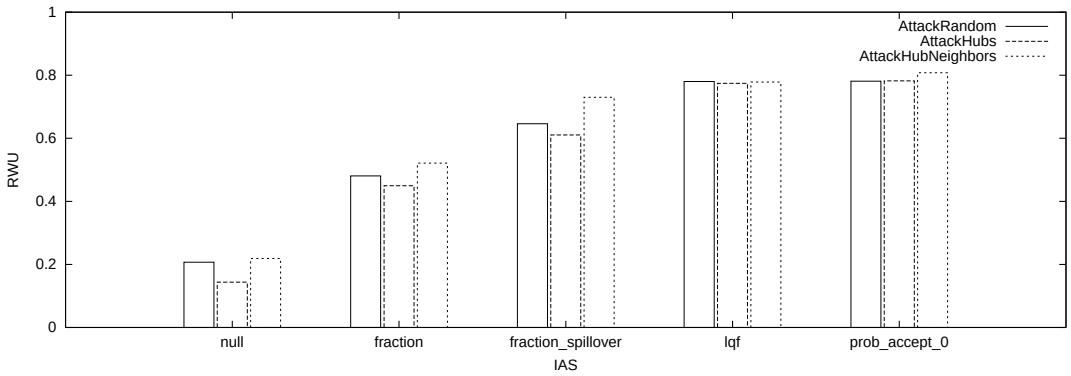

Figure 3.17: RWU vs. IAS for Various PAMs with 5 percent malicious nodes

3.2.4 Positional Attack Models (PAMs)

In our results thus far (with the exception of the TTLShaping results in Section 3.2.2), we have assumed that malicious nodes are randomly distributed throughout the topology. In this section, we study how the other attack models described in Section 3.1.3 effect our results.

Figure 3.17 shows the results of a number of simulations in which 5 percent of the nodes are malicious, and the malicious nodes take on different positions in the network. For each of the IASes that we study (on the x-axis), the y-axis measures the steady-state RWU for the AttackRandom, AttackHubs, and AttackHubNeighbors PAMs. For each IAS, we can see that the differences in RWU for the three PAMs are only marginally different, although AttackHubs systematically results in the least RWU, followed by AttackRandom, and finally by AttackHubNeighbors. What this tells us is that regardless of which positions the malicious nodes take on in the network, they are able to have about an equivalent effect regardless of what IAS is being used. Due to the structure of Gnutella-1787, and its similarity to a power-law topology, every node, including malicious nodes, are almost always within just a few hops of highly connected nodes. Once queries arrive at the highly connected nodes, they are multiplicatively broadcasted, and in the case that the queries are issued by malicious nodes, the queries deny a significant amount of service to queries issued by good nodes.

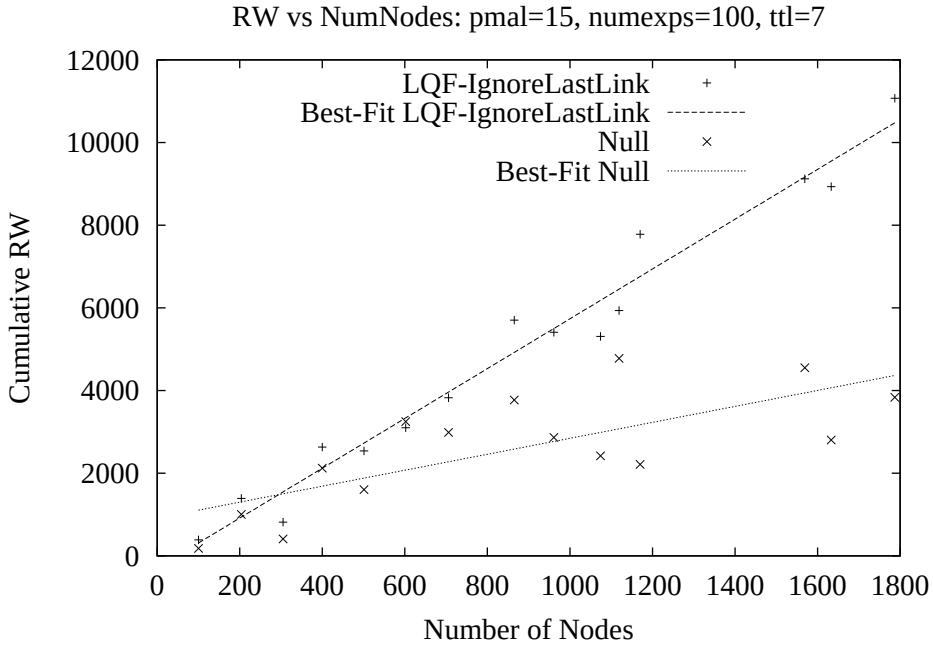

Figure 3.18: Cumulative RW vs. Number of Nodes

3.2.5 Scalability

In this chapter, we have run simulations on the largest Gnutella topology available from [178], but we wanted to have some level of confidence that our results in the above sections were not dependent upon the size of the topology, or the particular snapshot that we used. Since we are not aware of the availability of larger, fairly accurate topologies, we used smaller snapshots that were gathered by Sariou et al. to verify the performance of the policies we propose. In this section, we present one of the experiments we conducted to assess whether or not we can expect that our results scale with the size of the network.

Figure 3.18 plots the cumulative amount of RW after 2τ time-steps for topologies of different sizes. In each of the topologies, 15 percent of the most highly conencted nodes in the topology were selected to be malicious (AttackHubs PAM), and the malicious nodes use StandardFlood FS. The results of 100 simulations were averaged to obtain each of the data points. Topologies were chosen where the size of the

networks fell closest to increments of 100, but as can be seen from the figure, no topologies of sizes near 1300, 1400, and 1500 were available in the data set.

Simulations were run with both Null and LQF-IgnoreLastLink IASes, and lines were best-fit for each of the policies. From Figure 3.18, we can see that as the number of nodes in the topologies increases, the cumulative RW (for both IASes) also increases linearly. However, the slope of the best-fit line for the Null IAS line is much flatter than the best-fit line for the LQF-IgnoreLastLink IAS indicating that as the size of the topologies grow, LQF-IgnoreLastLink successfully mitigates flooding proportionally to the size of the network. The LQF-IgnoreLastLink best-fit line had an asymptotic standard error of under 5 percent. We expect that the linear trend would continue if simulations were to be run on larger topologies. In addition, while we present the scalability of LQF-IgnoreLastLink here, we ran the same scalability experiment on the other IASes we studied in this chapter, and found that the trends we describe in our results are observable in topologies of different sizes. As such, we expect that the policies we propose and evaluate in this chapter should be applicable to larger Gnutella topologies also.

3.3 TTLShaping

3.3.1 Choosing d_α

In Section 3.1.2, we described a TTLShaping policy in which good nodes limit the number of queries they accept with different TTLs. The intuition behind TTLShaping is that if all good nodes in the network admit at most $\hat{\rho}C$ queries each, then we can expect that the TTLs of queries that arrive on any particular link should have a geometric distribution.

We experimented with different settings of d_α for Gnutella-1787, and the results are shown in Figure 3.19. RWU is plotted for increasing percentages of malicious nodes in the network under a Null IAS and a TTLShaping DS. Malicious nodes are placed using the AttackHubs PAM, and they conduct a StandardFlood in which they set $\rho = 1$. From the figure, we can see that at 5 and 10 percent of the nodes being

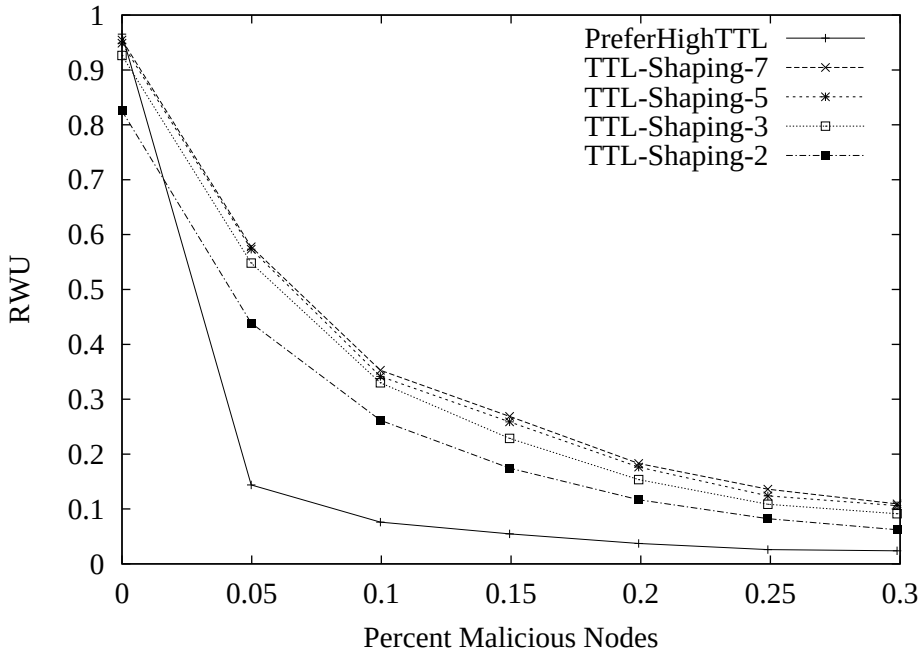

Figure 3.19: RWU vs. Fraction of Malicious Nodes for TTLShaping

malicious, TTLShaping produces a 50 and 30 percent increase in RWU, respectively, compared to PreferHighTTL when $d_\alpha = 3$. Higher settings of d_α only recover marginal RWU.

3.3.2 TTLShaping-Flood

In Section 3.3.1, we saw that when malicious nodes inject all their queries with TTL τ and $\rho = 1$, TTLShaping filters out a significant fraction of the malicious queries they inject. However, malicious nodes can inject their queries with a distribution of TTLs that is within the bounds imposed by the TTLShaping policy. Even when the malicious nodes do so, the effect of their query-flood will not be as significant as before, since most of their malicious queries will not travel as far in the network.

Figure 3.20 shows the results of the same simulations as those run to generate Figure 3.19 with $d_\alpha = 3$ and $d_\alpha = 5$ and two additional lines for those settings of d_α in which malicious nodes injected queries using a TTLShaping-Flood as described in

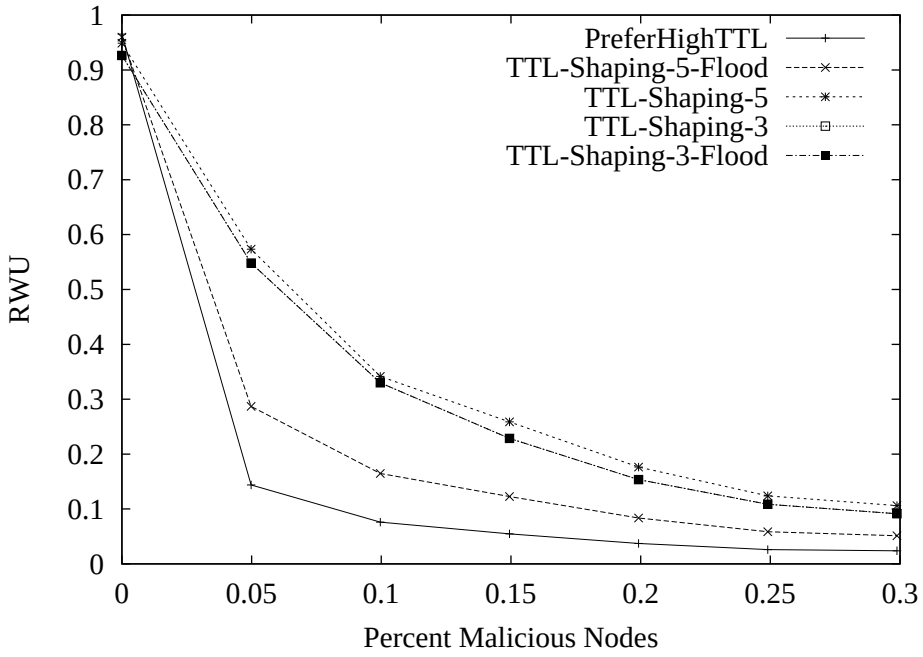

Figure 3.20: RWU vs. Fraction of Malicious Nodes for TTLShaping under TTLShaping-Flood

Section 3.1.3. We can see that with $d_\alpha = 3$, increased RWU can be gained (compared to $d_\alpha = 5$) even if malicious nodes use a TTLShaping-Flood. However, if $d_\alpha = 5$ is used, too many malicious queries persist in the network, and RWU is significantly lower than when $d_\alpha = 3$. It is also important to note that at $d_\alpha = 3$, RWU, on average, does not suffer at all due to the TTLShaping-Flood. This occurs because when $d_\alpha = 3$, malicious nodes are injecting queries with the almost the exact distribiton of TTLs that good nodes forward.

3.4 Complementary Approaches

Other approaches to deal with the problem of query-floods may involve preventing malicious nodes from injecting queries in the network or requiring malicious nodes to expend more effort before they are able to inject a query. For instance, one could require each node in the network to sucessfully authenticate itself before being able

to submit queries. Then, a malicious node would be required to subvert the authentication mechanism or take control of an already authenticated, good node. On the other hand, an authentication-based approach may limit the openness of the network, and/or may not be able to as easily allow anonymous nodes to participate.

Alternatlively, a node can be required to expend some effort before other nodes accept a query. A node can be required to, for example, solve a cryptographic puzzle, and attach the solution to a puzzle to each of its queries. Nodes in the network would then verify the answers to crypto puzzles before processing and forwarding queries.

The query selection approaches that we present in this chapter are complementary to these alternative approaches. Malicious nodes may subvert authentication mechanisms, or may be willing to expend the effort to solve crypto puzzles. In either case, the techniques we have proposed and evaluated in this chapter mitigate the effects of query-floods once the queries have been injected into the network.

3.5 Chapter Summary

In this chapter, we have studied flooding-based DoS attacks through simulations on real snapshots of Gnutella networks.

We found that:

- A TTL of 3 (as opposed to a TTL of 7 as is used in practice) is sufficient to maximize remote work when LQF and Fractional IASes are used. Remote work degrades as TTL increases when Fractional-Spillover and Null IASes are used.
- A LQF IAS is effective at recovering remote work (compared to Fractional and Null IASes) when many malicious nodes are present in the network. In the case that malicious nodes use LQF-IgnoreLastLink-Flood, LQF performs comparably to Fractional.
- A PA IAS is also effective at recovering remote work when malicious nodes are present, but performs worse than Fractional IAS if the malicious nodes use PA-Flood. On the other hand, PA IAS is better at localizing damage to fewer nodes, and acheives more remote work when high TTLs are used.

- A TTL-Shaping DS can be used instead of PreferHighTTL to increase RWU by between 30 to 50 percent when 5 to 10 percent of the nodes in the network are malicious.

Now that we have studied flooding-based DoS attacks in-depth in the unstructured Gnutella P2P network, we will study DoS attacks in DHT and non-forwarding P2P networks in subsequent chapters.

Chapter 4

Blasting in a DHT

In this chapter, we extend the model we developed in Chapter 2 to allow us to study performance and denial-of-service issues in Distributed Hash Tables (DHTs). We focus on performance and denial-of-service issues that result from nodes "blasting" queries. In a "blasting" attack, a malicious node spends all of its processing capacity generating queries in an attempt to overload the network. While superficially similar to malicious nodes that flood in unstructured P2P networks, blasting has different effects because queries are not repeatedly broadcasted at each node of the network. Also, while our extended model allows us to study blasting in many DHTs, we focus on Chord for concreteness.

4.1 Review of DHTs and Chord

A DHT (see Chapter 1) is a P2P system in which a set of data items $\{D_1, D_2, ..., D_k\}$ is distributed across the nodes $\{N_1, N_2, ..., N_n\}$, such that each data item can be predictably found at a particular node. Each of the data items and node ids are mapped to an address space $[0, A)$, and each node is responsible for storing data items that map to a subset of the address space. The subsets of the address space that different nodes store are typically not overlapping, such that, at any given time, there exists only one node responsible for storing a particular subset of the address

space [1].

Each node keeps a number of pointers to other nodes that handle "adjacent" partitions of the address space. The set of pointers is typically called a "finger" or "routing" table. As nodes join and leave the network, the address spaces that particular nodes are responsible for changes, as do the routing tables.

When a node N_i is interested in searching for a data item D_j, N_i computes (using a hash function) the address of the data item $h(D_j) \in [0, A)$. If N_i is the node that handles the corresponding part of address space, the data item can be found locally. If not, then N_i consults its routing table to determine which "adjacent" nodes are responsible for addresses that are closest to $h(D_j)$.

In this chapter, we focus on blasting attacks in the Chord [193] DHT, but our extended model and metrics can be applied to any DHT. We choose Chord because it is the DHT that is most "flexible" with respect to node failure. That is, Chord has the fewest constraints of all the DHTs studied in [79] as to which nodes can appear in particular entries of a routing table, and is thus most resilient to failures of particular nodes. Ratnaswamy et. al. present detailed simulation results and analysis of the advantages of Chord as opposed to other DHTs in [79].

We described how Chord works in Chapter 1, but we provide a brief review here. The reader is also encouraged to consult [193] for a detailed treatment of Chord.

In Chord, each node N_i hashes its IP address to determine its own address $h(N_i)$. Each node maintains a routing table with $\log n$ entries. The ith entry in a node's routing table points to another node whose hash is the smallest in the network that is larger than $h(N_i) + 2^{i-1}$. To search for a data item with id a, a node N_i determines which routing table entry j for which $h(N_i) + 2^{j-1} \leq a < h(N_i) + 2^j$, and forwards its query to node pointed to by the jth entry in the routing table. On average, a query will be forwarded $\frac{1}{2} \log n$ times before it reaches its destination [193].

To illustrate how Chord works concretely, we use a 3-node example in which the nodes have ids 0, 1, and 3. In this small example, we assume that the system can support a maximum of 8 nodes. In Table 4.1, we show which nodes are responsible

[1] The fault-tolerance of many DHTs can be improved by mapping data items to multiple nodes; if a data item cannot be found at one of the nodes that it is mapped to due to a node failure, it can be found at one of the others to which it is mapped.

Node Id	*Responsible for keys:*
0	0, 4-7
1	1
3	2, 3

Table 4.1: Key Distribution Across Nodes

Index	*Node Id*
1	1
2	3
3	0

Table 4.2: Routing Table for Node 0

for storing which keys.

Each of the nodes has a routing table with three entries as shown in Tables 4.2, 4.3, and 4.4.

To illustrate how queries work in Chord, consider what happens when node 1 issues a query for a key that hashes to 4. The query is sent to node 3 because $1 + 2^1 = 3$ which is the largest node id that is less than 4. Node 3 is responsible for storing data items with keys 3 through 7, so it responds to the query.

4.2 Extended Model

In this section, we describe a model that we use to capture the query flows in a DHT with N nodes such that we can study blasting attacks. As in Chapter 2, our model is a discrete-event-based model. In each time step, a node conducts three actions: 1) it admits new queries into the system, 2) it answers queries that have arrived for which it is responsible for storing the corresponding data items, and 3) it forwards any remaining queries to "adjacent" nodes. Each of these actions takes some amount of processing capacity, but for simplicity we assume that each such action requires one unit of processing capacity. (In Section 4.7.8, we study the case in which answering queries is more expensive than forwarding queries.) One unit of

Index	*Node Id*
1	3
2	3
3	0

Table 4.3: Routing Table for Node 1

Index	*Node Id*
1	0
2	0
3	0

Table 4.4: Routing Table for Node 3

processing capacity may involve some arbitrary number of CPU cycles, disk I/Os, and network bandwidth, but for the purposes of our study, we aggregate all these sub-component resources required to process a query into a single unit of normalized processing capacity. Also, we will say that a node has processed a query when it has either admitted, answered, or forwarded it. Any of these three actions constitutes processing a query. Finally, we assume that each node has some maximum processing capacity constraint, and that each node in the system can process C queries per time step. That is, each node can admit, answer, and/or forward a maximum of C queries per time step. Each node may execute some combination of these actions, but can execute no more than C of these actions.

4.2.1 Reservation Ratio (ρ)

Each node must decide what fraction of its processing capacity it should dedicate to each of the actions above.

In particular, nodes admit gC queries, answer aC queries, and forward fC queries where $g + a + f \leq 1$. We say that when a node answers a query it has done one unit of "work." In addition, if a node answers a query that was admitted at some other node, we say that it has done one unit of "remote work." (When a node answers a query that was admitted locally, we say that it has done one unit of "local work.")

A node must expend some of its bandwidth for injecting queries into the network, and we assume that a node reserves $\rho C = gC$ units of processing capacity at each time step for query injection / admission. A node's remaining query bandwidth can be used to either answer or forward queries.

We are interested in studying Chord networks when they are under stress, and, as in previous chapters, we make the assumption that nodes have a near infinite supply of queries that they could admit to the system. However, if all nodes spend their entire processing capacity admitting queries and forwarding them along their first hop, they will have no processing capacity left over to answer queries or route them to their destinations. Ideally, we do not want any queries to be dropped due to a lack of processing capacity at any node. Therefore, we want to determine the setting of ρ that will result in the highest throughput (remote work) and no dropped queries. We call the setting of ρ that maximizes remote work *optimal rho*, denoted by $\hat{\rho}$.

Due to the properties of Chord, we can estimate $\hat{\rho}$ analytically. We start with the capacity constraint that the total fractions of queries admitted, answered, and forwarded by a node in a given time step must sum up to a maximum of one:

$$a + g + f \leq 1$$

If all nodes have the same capacity, and admit the same number of queries ρC, we can expect that they will receive ρC queries to which they can provide answers. We therefore assume that $a = g$ if we would like to answer all the queries that are admitted in the system. We let $\rho = a = g$ be the ratio of processing capacity that nodes set aside for admitting and answering queries. In addition, because we do not want any processing capacity to be wasted, we use strict equality:

$$\rho + \rho + f = 1$$

In Chord, a query is forwarded, on average, through $\frac{1}{2} \log N$ nodes before it arrives at its destination [193]. If we assume that each node must spend a corresponding amount of its processing capacity forwarding queries, such that all queries can arrive at their destinations, we have:

$$\rho + \rho + \frac{1}{2}\rho \log N = 1$$

Solving for ρ:

$$\hat{\rho} = \frac{1}{2 + \frac{1}{2}\log N}$$

Nodes set $\rho = \hat{\rho}$ to maximize remote work.

In deriving our estimate for $\hat{\rho}$, we made the assumption that all nodes behave symmetrically in each round of operation. However, there are various sources of variability that cause nodes to have different loads. As a result, we may not be able to maximize remote work with our estimated $\hat{\rho}$ setting if some of the following sources of variability are present:

1. *Non-uniformly distributed node ids.* When nodes choose their ids at random, the ids may not be perfectly distributed around the Chord ring / address space. As a result, some nodes may be responsible for forwarding queries to a larger part of the address space than others. Consider, for instance, the small network of 3 nodes in Section 4.1 in which nodes had ids 0, 1, and 3. In that example, node 0 is responsible for the largest part of the keyspace, and node 3 must forward all queries it receives to node 0. It is therefore likely that if query keys are chosen at random from the key space, then node 3 will receive more queries to forward than if the node ids were uniformly distributed around the ring. We will shortly describe some approaches that have been proposed to deal with the issue of non-uniformly distributed node ids.

2. *Non-uniform query key distribution.* Query keys will not necessarily have a uniform random distribution. Some documents may be more "popular" than others, and keys that match such documents may appear more frequently than other keys found in queries. A non-uniform query key distribution will result in some nodes becoming "hot-spots."

3. *Variable hops to destination.* While queries will take $\frac{1}{2}\log N$ hops to arrive *on*

average, some queries may take more or less hops to arrive at their destinations, causing transient changes in load at nodes. These transient load changes can easily be evened out by requiring that nodes use finite length queues to temporarily store queries that cannot be forwarded or answered at a particular point in time. At a later point in time, if processing capacity is not fully utilized, the node can service queries from its queue.

For the reasons above, only a fraction of admitted queries result in RW. While the third source of variability above, variable hops to destination, can easily be addressed by having nodes use queues to smooth out their load over time, solutions for non-uniform node id and query key distributions are a topic of active research.

We note that non-uniformly distributed ids can be used as a proxy for other sources of load variability. A node that is responsible for a large part of the keyspace in a system with non-uniformly distributed ids is equivalent to a node that receives many queries for a "popular" key in a system in which nodes do have uniformly distributed ids. We do not model documents or file distributions across nodes in our work here, and we use non-uniformly distributed node ids to simulate the effect of non-uniform query key distribution. So, while the results we provide in later sections of this chapter may specify that non-uniform node ids were used, we expect the results will be similar for non-uniform query key distributions, and potentially other sources of load variability.

The importance of load balancing and eliminating the above sources of variability in Chord has received some attention in the literature. Basic solutions have been proposed to achieve uniformly distributed node ids. We briefly survey three basic solutions proposed to uniformly balance node ids, and point out their strengths and limitations. Stoica et. al. in [193] propose that each real node in a Chord system should host $\log N$ virtual nodes. While Stoica et. al. showed through simulation that virtual nodes can be used to balance load, this approach has the disadvantage that each real node will have to maintain $\log^2 N$ connections instead of $\log N$ connections. Alternatively, Karger and Ruhl in [92] provide a node id balancing scheme in which each real node is only required to maintain $\log N$ active connections for one of its virtual nodes at a time, but may require many nodes to change which virtual node

is active when some real node leaves the system. Manku [120] develops a node id balancing scheme in which nodes sample the id space upon joining, and choose an id that will lead to a nearly uniformly balanced spacing between nodes participating in the system.

While these approaches may achieve node id balancing at the expense of active connections and network stability, there is still much room for load variability due to non-uniform query key generation. Karger and Ruhl in [92] also provide an item balancing scheme in which nodes can collaborate to re-assign node ids to balance load based upon the run-time distribution of query keys. There are two key disadvantage(s) of this approach. Firstly, if some nodes are malicious, they could force address re-assignments that allow them to take control of particular data items or parts of the address space. Secondly, nodes are required to collaborate to balance load. Malicious nodes could create highly undistributed load by lying about their loads.

In studying the policies that we proposed in Section 4.3, and in contrast to the solutions proposed in [92] and [193], we are interested balancing load irrespective of the source of load variability. The policies we propose are complementary to those proposed in [92], [193], and [120]. Our policies will help balance any transitory or interim load variations that occur while the proposed node id balancing schemes are executing, and our policies will also help balance load in the case that "non-compliant" or malicious nodes do not follow the proposed id balancing schemes. In addition, our policies are practical, easy-to-implement, and when a Chord network is under stress, they provide increased system throughput (potentially at the cost of "fairness").

4.3 Policies

There are a number of different traffic management policies that nodes may attempt to use to maximize remote work, handle traffic "surges" caused by load variability, and contain the effects of malicious nodes. In this section, we describe a number of basic policies.

A node must decide what mix of its available query bandwidth it should use to

spend answering queries versus forwarding queries. When a query arrives at a node, we say that the query is *answerable* if the node is responsible for storing the *value* corresponding to the *key* specified in the query. A query is *forwardable* if it is not answerable. (Of course, all queries will eventually be answerable once they have been forwarded to the appropriate destination node.) Let A_{arrive} be the number of queries that arrive at a node that can be answered at that node, and F_{arrive} be the number of queries that arrive that can be forwarded by that node. Note that A_{arrive} can be greater or less than aC (and F_{arrive} can be greater or less than $(1-a-g)C$) even when $\rho = \hat{\rho}$ at any particular time due to variations in the number of hops that it takes for queries to travel from their sources to their destinations.

Nodes first use an incoming allocation strategy (IAS) to decide how many queries to answer and how many to forward. After decisions about how many queries to answer and forward have been made, nodes then use a drop strategy (DS) to decide which queries are to be dropped / ignored. In the following subsections, we describe some basic choices for IAS and DS policies.

4.3.1 IAS

In this subsection, we describe various IASes. Let A be the number of queries that are actually answered at the node and F be the number of queries that are actually forwarded by that node. For instance, if $A_{arrive} > aC$, then an IAS may choose to actually answer only $A = aC$ queries. Alternatively, if $A_{arrive} < aC$, then an IAS may choose to actually answer $A = A_{arrive}$ queries.

The goal of an IAS is to, given some set of $A_{arrive} + F_{arrive}$ queries that arrive at a node, decide how many answerable queries and how many forwardable queries should be processed. In illustrating various basic options for IASes, we use a running example in which a node has $C = 12$ units of capacity which it may use to admit, answer, and/or forward queries. In our examples, $\rho C = \frac{1}{6}12 = 2$ units of capacity are reserved to admit new queries. The remaining capacity is available for answering or forwarding queries.

We now describe various IAS policies. Note that in our descriptions, ρ may take

on values in the range $[0, \frac{1}{2}]$.

- *Null.* Null IAS answers up to ρC answerable queries, and up to $(1-2\rho)C$ forwardable queries. That is, Null IAS will answer $A = min(A_{arrive}, \rho C)$ queries, and forward $F = min(F_{arrive}, (1-2\rho)C)$ queries. Any unused capacity is "wasted." For instance, if $A_{arrive} = 1$ and $F_{arrive} = 9$, Null IAS will answer $A = min(1, \frac{1}{6}12) = 1$ query, and will forward $F = min(F_{arrive}, (1-2\rho)C) = min(1, (1-2\frac{1}{6})12) = min(12, 8) = 8$ queries. The extra 1 unit of answering capacity is not re-utilized for forwarding queries.

- *Answer First Priority (AFP).* AFP IAS first spends its available capacity processing any answerable queries that arrive. Only after answerable queries have been processed are forwardable queries processed. More precisely, AFP first processes $A = min(A_{arrive}, (1-\rho)C)$ answerable queries, and then processes $F = min(F_{arrive}, (1-\rho)C - A)$ forwardable queries. For example, in our running example, if $A_{arrive} = 4$, then $A = min(A_{arrive}, (1-\rho)C) = min(4, (1-\frac{1}{6})12)) = min(4, 10) = 4$ answerable queries are processed. If $F_{arrive} = 8$, then $F = min(F_{arrive}, (1-\rho)C - A) = min(8, (1-\frac{1}{6})12 - 4) = min(8, 6) = 6$ forwardable queries are processed.

- *Answer First Spillover (AFS).* AFS IAS works similarly to AFP IAS with the exception that some capacity is reserved for forwarding queries. In particular, if $F_{arrive} < (1-2\rho)C$ then let $F_{leftover} = (1-2\rho)C - F_{arrive}$, else $F_{leftover} = 0$. AFS will process $A = min(A_{arrive}, \rho C + F_{leftover})$ answerable queries and $F = min(F_{arrive}, (1-2\rho)C)$ forwardable queries. For instance, if $A_{arrive} = 4$ and $F_{arrive} = 7$, then $F_{leftover} = (1-2\rho)C - F_{arrive} = (1-2\frac{1}{6})12 - 7 = 1$. AFS IAS will answer $A = min(A_{arrive}, \rho C + F_{leftover}) = min(4, \frac{1}{6}12 + 1) = min(4, 2+1) = min(4, 3) = 3$ queries, and forward $F = min(F_{arrive}, (1-2\rho)C) = min(7, 8) = 7$ queries.

- *Forward First Priority (FFP).* FFP processes any forwardable queries that arrive first before considering answerable queries. Specifically, FFP first processes $F = min(F_{arrive}, (1-\rho)C)$ forwardable queries, and then processes

$A = min(A_{arrive}, (1-\rho)C - F)$ answerable queries. For instance, if $F_{arrive} = 12$, then $F = min(F_{arrive}, (1-\rho)C) = min(12, (1-\frac{1}{6})12)) = min(12, 10) = 10$ forwardable queries are processed. No answerable queries that arrive are processed.

- *Forward First Spillover (FFS).* FFS IAS works similarly to FFP IAS with the exception that some capacity is reserved for answering queries. In particular, if $A_{arrive} < \rho C$ then let $A_{leftover} = \rho C - A_{arrive}$, else $A_{leftover} = 0$. FFS will process $F = min(F_{arrive}, (1-2\rho)C + A_{leftover})$ forwardable queries and $A = min(A_{arrive}, \rho C)$ answerable queries. For instance, if $F_{arrive} = 10$ and $A_{arrive} = 1$, then $A_{leftover} = \rho C - A_{arrive} = \frac{1}{6}12 - 1 = 1$. FFS IAS will forward $F = min(F_{arrive}, (1-2\rho)C + A_{leftover}) = min(10, (1-2\frac{1}{6})12 + 1) = min(10, 8 + 1) = min(10, 9) = 9$ queries, and answer $A = min(A_{arrive}, \rho C) = min(1, 2) = 1$ query.

4.3.2 DS

Once an IAS has been used to determine how many queries to answer (A), and how many queries to forward (F), a drop strategy (DS) can then be used to determine exactly which queries to process. Specifically, if $A < A_{arrive}$ and/or $F < F_{arrive}$, then a DS is used to determine which $A_{arrive} - A$ and/or $F_{arrive} - F$ queries to drop.

In this subsection, we describe some DSes.

- *Drop Youngest (DY).* In this DS, we assume that query messages have a "hop count" field. When a query message is first created, it is given a hop count of 0. Each time that the message is forwarded from one node to another, the hop count is incremented by one. The hop count indicates the "age" of the query. Queries that have been forwarded just a few times are considered "young," while queries that have been forwarded many times are considered "old."

  In the DY DS, queries with the smallest hop counts are dropped. The rationale behind DY DS is that the least amount of effort has been expended on young queries. That is, young queries have been forwarded by a fewer number of nodes, compared to older queries, and by dropping younger queries, the amount of effort that is wasted is lower.

For example, if an IAS is used to decide that $F = 8$ queries can be forwarded out of $F_{arrive} = 12$ queries, then the 4 queries with the lowest hop counts will be dropped. The 8 oldest queries will be forwarded as per a node's finger table. Ties are broken arbitrarily.

- *Drop Farthest (DF).* In DF DS, queries that have the farthest to travel, as measured by the "clockwise distance" between the key specified in the query and the id of the current node, are dropped first. The clockwise distance between two ids id_1 and id_2 is defined as d where $id_1 + d \mod N_{max}$ and N_{max} is the maximum node id possible. For instance, if $N_{max} = 12$, then the clockwise distance between 9 and 2 is 5.

  To illustrate DF DS, consider an example in which a node with id 8 (N_8) uses an IAS to decide that $F = 2$ queries can be forwarded out of $F_{arrive} = 4$ queries. Also, assume that $N_{max} = 32$. If the ids of the 4 queries are $\{9, 12, 17, 1\}$, then the corresponding clockwise distances between N_8 and the query keys are 1, 4, 9, and 25, respectively. DF DS would drop the queries with keys 17 and 1 since they have the farthest clockwise distances to travel.

- *Drop Random (DR).* In this DS, queries are dropped at random to meet the quotas set by an IAS, irrespective of their age or clockwise distances to their destinations.

4.3.3 Retransmission Policy

In this subsection, we consider what a node might do when it does not receive an answer to its query. After issuing (admitting) a query, a node may not receive a answer because the query was dropped along the path to its destination.

- *No Retransmission (No-RTX).* Using this policy, nodes do not retransmit queries if they do not receive an answer. Of course, this might lead to unhappy users.

- *Standard Retransmission (S-RTX).* A node that uses S-RTX will retransmit its query after a fixed timeout period if it does not receive a answer. The retransmitted query will have the same key (destination) as the original query.

4.4 Threat Model

In this section, we explicitly state the expected behaviors for good and malicious nodes in our model.

As in Chapter 2, we assume that good nodes are altruistic in the sense that they would like to maximize the total number of queries that are answered in the network. That is, they would like to maximize the remote work. Good nodes, therefore, set $\rho = \hat{\rho}$.

Even though various mechanisms to prevent malicious nodes from joining a Chord network might be deployed, it may still be possible for a limited number of nodes to breach such defenses. For instance, even if nodes are required to have certified node ids (as suggested in [26]), a malicious adversary could compromise the security of existing hosts that have valid node ids.

There are many possible attacks that malicious nodes that have joined a Chord network can carry out. In this chapter, we consider malicious nodes that blast useless queries in an attempt to deny service to legitimate queries. Malicious nodes spend all of their processing capacity admitting queries into the system, and spend none of their capacity answering or forwarding queries on behalf of other nodes. We therefore model malicious nodes as nodes that set $g = 1$, and $a = f = 0$.

The malicious nodes that we study in this chapter are interested in simply introducing extra work into the system, and they are not interested in attacking particular victim nodes. The queries that they admit are for random keys, and are sent to destinations that are distributed at random around the Chord ring. Finally, malicious nodes choose node ids at random from the node id address space.

4.5 Traffic Limiting

To deal with malicious nodes that pose the threats outlined in the previous section, we propose traffic limiting countermeasures in this section.

As malicious nodes in our threat model simply admit more queries than good nodes, one approach to dealing with them is to limit the amount of traffic that nodes

will accept from each other. However, even basic traffic limiting policies for Chord are likely to be more complicated than, for instance, the *Fractional* policy that we discussed in Chapter 2 for Gnutella because the expected traffic patterns in Chord are more complicated. In Chord, nodes do not simply flood the network with their queries, but send them towards their destinations in a highly directed fashion based on their finger tables.

We will now describe two complementary traffic limiting mechanisms, an *admission limit* and a *forwarding limit*. Both mechanisms take advantage of observations that we make regarding expected traffic patterns of queries in a Chord network consisting of all good nodes. The admission limit applies to queries that have just been admitted and have a hop count of one, while the forwarding limit applies to queries that have been forwarded at least once and have a hop count greater than one.

4.5.1 Admission Limit

We start by describing the first of these two traffic limiting mechanisms, the *admission limit*. When a good node admits a query, the query key can be expected to randomly fall anywhere in the address space so long as a good hash function is used to map search terms to query keys. One-half of the queries admitted by a good node are expected to have keys that map to the half of the Chord ring that is farthest from it. Namely, a good node with id i, N_i, is expected to have half of the queries that it admits have keys in the range $[i+2^{\log N-1}, i-1)$. If a key is in the range $(pred(i), i-1]$, the query will be handled locally, but most of the query keys in the farthest half of the ring will be in the range $[i+2^{\log N-1}, pred(i)]$. Such queries will be routed to the node pointed to by the $\log Nth$ entry in its finger table. That is, such queries will be routed to the node with the smallest id that is larger than $i+2^{\log N-1}$.

Let node N_h be the node that has the smallest id that is larger than $i+2^{\log N-1}$. Note that since the node N_i is good, it generates a total of $\hat{\rho}C$ queries. Node N_h can therefore expect that one-half of these $\hat{\rho}C$ queries should be routed to it. If node N_h receives more than $\frac{1}{2}\hat{\rho}C$ queries from node N_i, on average, then node N_i may be admitting too many queries and/or might be malicious. A node N_h that uses an

admission limit and receives queries from N_i accepts no more than $\frac{1}{2}\hat{\rho}C$ queries from N_i.

We could further generalize the admission limit rule to be parametrized based upon the distance between the sending and receiving node. The farther the distance between the sending and receiving node, the more queries the receiving node should allow the sending node to admit. Consider a sending node N_i and a receiving node N_h. Let i and h be the respective node ids of N_i and N_h. Also, let D_C be the circumference, or total distance around the Chord ring. [2] The relative distance between the sending node N_i and N_h is $\frac{|i-h|}{D_C}$, and we can generalize our admission limit as follows: the maximum number of queries that N_h should accept from N_i per unit time is, on average, $\frac{|i-h|}{D_C}\hat{\rho}C$.

4.5.2 Forwarding Limit

In addition to the admission limit, a good node can use a forwarding limit. In this subsection, we derive such a *forwarding limit*.

Consider our node N_h (from Section 4.5.1) that sends queries to a node N_q where N_q is the node with the smallest id that is larger than $h + 2^{\log N-2}$. (N_q is one quarter of the way around the ring from N_h.) All queries that N_h sends to N_q with a hop count greater than one have already been admitted. Once these queries arrive at N_q, they are to make progress towards the forth-quarter of the ring with respect to N_i, the node that originally admitted the queries. (Queries that are to make progress towards the third-quarter of the ring are forwarded to nodes in between N_h and N_q.) Node N_h is responsible for forwarding approximately $(1 - 2\hat{\rho})C$ queries, as per our derivation of $\hat{\rho}$. Hence, one-half of these queries are to be forwarded to the third-quarter of the ring and one-half of these queries are to be forwarded to the forth-quarter of the ring. Therefore, node N_q can expect to receive approximately $\frac{1}{2}(1-2\hat{\rho})C$ from N_h. If node N_q receives more than $\frac{1}{2}(1 - 2\hat{\rho})C$ queries from node N_h, on average, then node N_h may be forwarding too many queries and/or might be malicious. A node N_q that uses a *forwarding limit* and receives queries from N_h, accepts no more than $\frac{1}{2}(1 - 2\hat{\rho})C$

[2] We also use D_C in our definition of "fairness" in Section 4.6.2.

queries from N_h.

In Section 4.7.9, we experiment with using the admission and forwarding traffic limits we described here, and we find that imposing such limits is able to mitigate the impact that malicious, blasting nodes are able to have on RW.

4.6 Metrics

4.6.1 Remote Work

Our notion of remote work in this chapter is similar to the remote work of the previous chapter with the exception that one unit of remote work is considered done only when a query is answered, not simply when it is forwarded.

When a query arrives at its destination and is answered, one unit of remote work has been done. Remote work is a measure of throughput– the more queries that arrive at their destinations and are answered in a given round, the more users will be happy with the system since their queries are being answered. Hence, if we want to maximize the happiness of our users, we want to maximize remote work. In the remainder of this chapter, we will focus on studying steady-state remote work, the remote work that takes place in one round once the system has achieved steady-state. We will abbreviate steady-state remote work as RW.

4.6.2 Fairness

In Section 4.7, we will find that although some of the policies we propose in Section 4.3 increase RW, they may do so at the expense of being "fair" to queries that have to travel varying distances through the network to arrive at their destinations. For instance, the DF DS favors dropping queries that have to travel far distances. To quantify how "unfair" the DF DS (or any given IAS or DS) might be to queries that are required to travel varying distances, we introduce a fairness metric.

We say that the distance, D, that a query is required to travel is the clockwise distance (as defined in Section 4.3) between its origin and its key. Let D_D be the average distance of queries that are dropped by any node, and D_A be the average

distance of queries that are answered by some node when a particular IAS and DS combination is in use by all the nodes in the network. Finally, we let D_C be the total distance around the Chord ring. For instance, if the address space from which node identifiers and keys are chosen from is $[0, A)$, then $D_C = A$.

We define F, the fairness during a given round, as follows:

$$F = \frac{D_A - D_D}{D_C}$$

If the average distance of queries that are dropped and the average distance of queries that are answered are the same ($D_D = D_A$), then $F = 0$. On the other hand, if there is a difference between these two averages, then F is non-zero. For instance, $F > 0$ then queries that only have to travel relatively short distances are favored.

The key idea here is that if queries that are being answered and queries that are being dropped are being treated fairly with respect to their distance, then F will be close to 0. Otherwise, if there is a significant difference in the distances of queries are that getting answered and those that are being dropped, then F will be considerably higher or lower than 0.

Note that fairness is only well-defined if at least one query in the system has been dropped and at least one query has been answered. In our simulations in Section 4.7.7, we measure F once the network has achieved steady-state.

4.7 Results

4.7.1 Simulation Setup

In this section, we describe the results of various simulations that we ran to determine which of the policies described in Section 4.3 perform best under a number of different scenarios.

The goal of our evaluations is to build a fundamental understanding of the issues and trade-offs involved in using the various policies we outlined in Section 4.3. Our evaluations are not designed to predict the performance of an actual system, but to gain an understanding of the trends and trade-offs involved in using the different

Parameter	*Value*
Number of Nodes (N)	256
Capacity (C)	1000
Reservation Ratio (ρ)	$\hat{\rho} = 0.1\bar{6}$
IAS	Null
DS	Null
RTX	No
Id Placement	Non-Uniform

Table 4.5: Baseline Simulation Parameters

policies. While we do not expect our simulations to predict actual query loads (as might be observed in a real network), we *do* expect them to tell us about relative performance that can be achieved by using the different policies that we described in Section 4.3.

In the evaluations described below, we simulated a Chord network using the baseline parameters in Table 4.5. In our evaluations, we vary some of these parameters, and we explicitly mention when we do so. In reporting results of simulations, if we do not mention a particular parameter, its value is set as per the baseline value specified in Table 4.5.

In our simulations, there are $N = 256$ nodes in the network at any instant, and nodes do not join and leave the network as we are interested in studying the system's steady-state behavior. The same trends that we see in our small 256-node simulations can be seen in larger Chord networks, and we clearly expect real Chord networks to have much larger numbers of nodes.

For simplicity, we assigned all of the nodes in the network the same capacity, $C = 10^4$. Hence, the maximum amount of total work, which includes local plus remote work, that can take place in an $N = 256$-node network in one time-step is $CN = 10^4(256) = 2,560,000$. To keep our work figures reasonable and meaningful, however, we normalize all our results by dividing by $C = 10^4$ such that the maximum total work is $N = 256$, but all simulations are carried out with a "precision" of $C = 10^4$. Therefore, for instance, the maximum total work that can be achieved in one time-step in a particular simulation is 256.

4.7.2 Optimal Rho

In Section 4.2.1, we analytically computed $\hat{\rho}$, the setting of ρ that maximizes RW. In this subsection, we verify that our estimate of $\hat{\rho}$ is accurate for our baseline parameters by simulation.

Figure 4.1 measures RW as a function of ρ. Each point in Figure 4.1 is the result of a simulation which was run with nodes using Null IAS and DR DS until steady-state. (We will present simulations on how many rounds are needed to achieve steady-state in Section 4.7.3.)

Figure 4.1 shows that our estimate of $\hat{\rho}$ is accurate under the baseline simulation parameters. First, note that if $\rho = 0$, then no queries are admitted– hence no queries are answered, and RW is 0. On the other hand if $\rho = \frac{1}{2}$ then nodes dedicate half of their processing capacity to admitting queries, and half of their processing capacity to answering queries. Unfortunately, at $\rho = \frac{1}{2}$, nodes do not dedicate any of their processing capacity to forwarding queries, and because queries are not forwarded to their destinations, RW is close to 0. (The 0.5 units of RW at $\rho = \frac{1}{2}$ occurs due to queries that are admitted whose first hop happens to be their destination.) As ρ varies from 0 to $\frac{1}{2}$, RW increases until its maximum, and then decreases.

Note that RW is maximum at $\rho = \hat{\rho} = 0.1\bar{6}$. If $\rho < \hat{\rho}$, not enough queries are admitted to saturate the answering and forwarding processing capacity of nodes. On the other hand, if $\rho > \hat{\rho}$, then more queries are being admitted than can be answered and/or forwarded.

When $\rho = \hat{\rho}$, we can see that RW is just under 18. Since there are 256 nodes in our simulation, and each of them dedicate ρ of their capacity to answering queries, we might expect that the maximum possible RW that can be achieved is $256\rho = 256(0.1\bar{6}) = 42.7$. Of course, in order to achieve a RW of 42.7, every query that is admitted must arrive at its destination. Unfortunately, some of the queries are dropped along the way to their destination, and some of the queries that arrive at their destination nodes cannot be processed due to a lack of answering capacity at the destination node. However, with a RW of 18, more than half of the queries that are admitted are dropped!

To understand why less than one-half of admitted queries were answered, consider

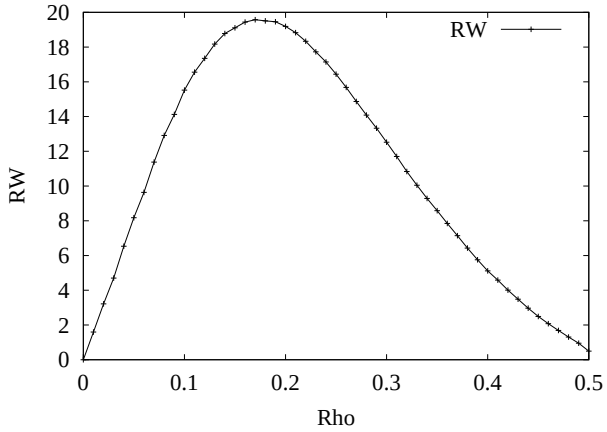

Figure 4.1: RW vs Rho

that when we derived the equation for $\hat{\rho}$ in Section 4.2.1, we assumed uniformly distributed node ids, equal load at each node, and each node to behave symmetrically. However, in simulations used to generate Figure 4.1, nodes chose their ids at random, and the ids are not uniformly distributed around the ring. The load variations caused by non-uniform node id distribution are quite significant, and result in lost RW.

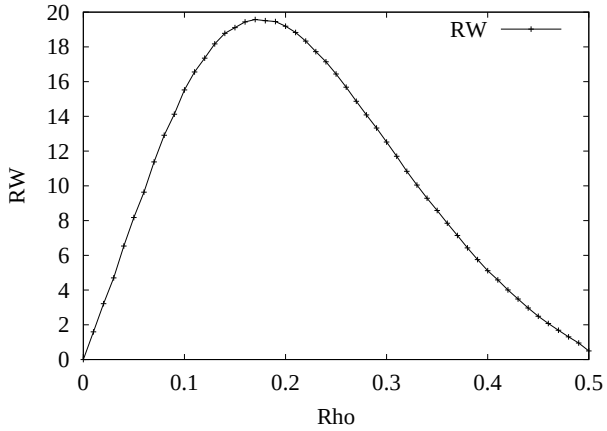

Figure 4.2: RW vs Rho: Uniformly Distributed Node Ids

To confirm that the source of the lost RW was indeed non-uniformly distributed ids, we ran a simulation in which node ids were chosen to be uniformly distributed

around the ring, and the results of that simulation are shown in Figure 4.2. As we can see from Figure 4.2, the RW achieved at $\hat{\rho}$ is 42.7, exactly the RW we expect.

4.7.3 Steady-State Performance

A Chord network with N nodes achieves steady-state within $k \log N$ rounds, where $k = 2$, for the IASes and DSes we consider in this chapter.

In this subsection, we study the number of rounds required to achieve steady-state for our various IAS, DS, and retransmission policies.

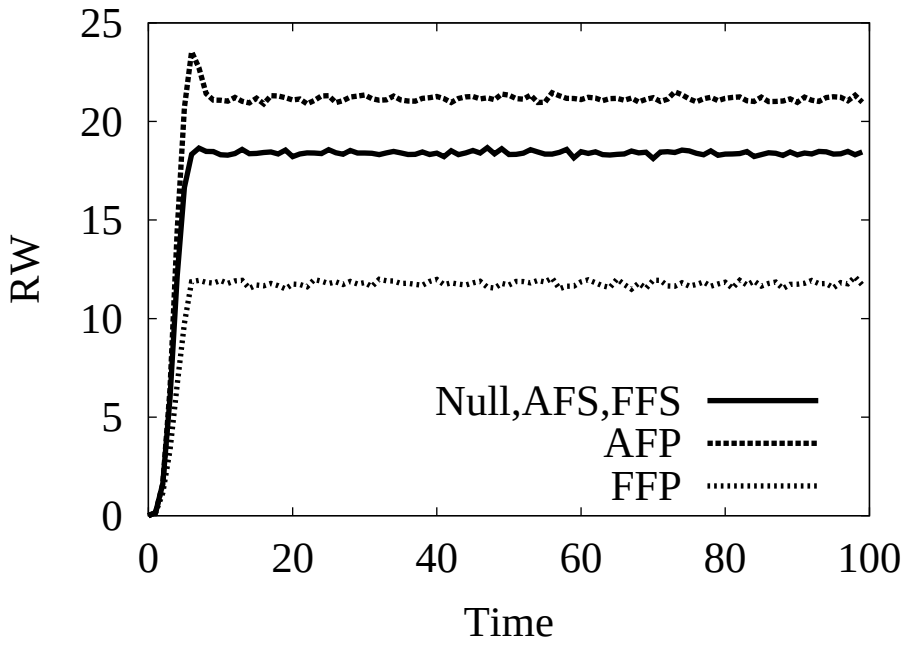

Figure 4.3: RW vs Time for Various IASes

Figure 4.3 measures RW over time for the various IASes. In a Chord network of N nodes, messages take an average $\frac{1}{2} \log N$ hops to arrive at their destinations. As such, it is reasonable to expect that if the system achieves steady-state, it should achieve it in $\Theta(\log N)$ rounds. In the case of Figure 4.3, steady-state is achieved in less than $2 \log N = 2 \log 256 = 16$ rounds.

While the RW measured for each IAS achieves steady-state in about 12 rounds,

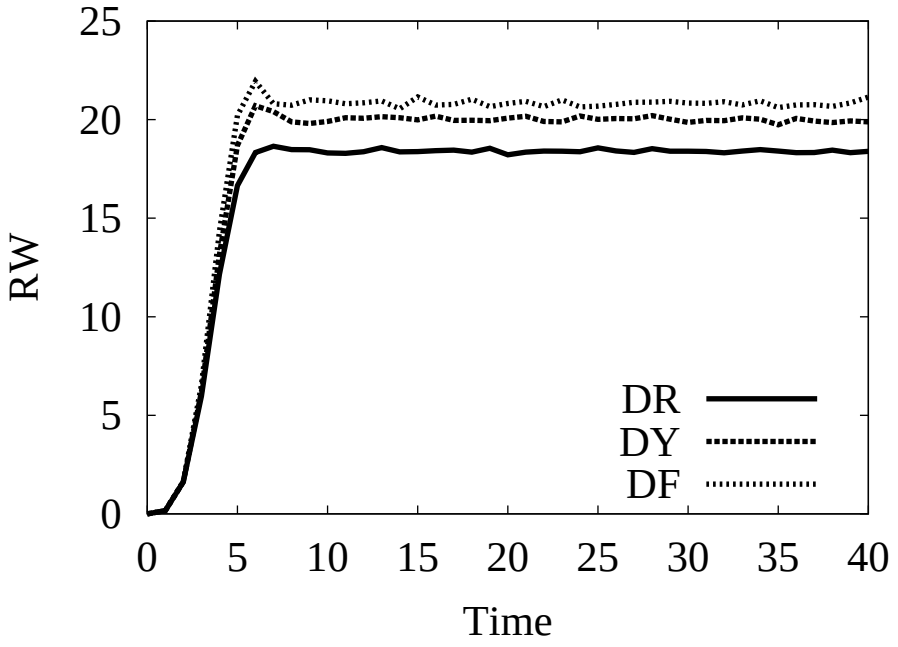

Figure 4.4: RW vs Time for Various DSes

note that the "hump" at time $t = 8$ for AFP is significant. When forwardable queries are dropped due to answerable queries being processed first at time $t = 8$, this causes a temporary "vacuum" of queries that can be answered shortly thereafter.

Figure 4.4 shows that the various DSes we consider also achieve steady-state within $2 \log N$ rounds. The DF and DY DSes experience a hump because they favor processing the closest and oldest queries first, respectively. Once the closest and oldest queries are answered at $t = 8$, there is a drop in RW because farther and younger queries, respectively, were sacrificed in forwarding the closest and older queries to their destinations.

All the remaining simulations in this chapter are run until steady-state.

4.7.4 IAS

AFP IAS maximizes RW.

Turning our attention back to Figure 4.3, we can see how RW varies with time

for various IASes. The DS and other parameters were set to their baseline values.

Initially, at $t = 0$ no queries have been admitted, forwarded or answered. During the first round of the simulation, nodes admit $\hat{\rho}C$ queries and forward them along their first hops. Approximately, $\frac{1}{2}\log N = 4$ rounds later, these queries arrive at their destinations, and RW increases from 0 at $t = 0$ to varying double-digit numbers for various IASes.

In the steady-state in Figure 4.3, we can see that AFP IAS maximizes RW. If a query has arrived at its destination, processing capacity should be given to answering it before admitting or forwarding other queries to maximize RW. If a query arrives at its destination, and it is not processed, the forwarding capacity that was used at other nodes along the path to the destination was needlessly wasted.

From Figure 4.3, we also see that the AFS, FFS, and Null IASes result in the same of amount of RW. In a given round, if a node using AFS, FFS, or Null IAS receives enough queries such that $A_{arrive} > \hat{\rho}C$ and $F_{arrive} > (1 - 2\rho)C$, then the node is overloaded and will answer $A = \rho C$ and forward $F = (1 - 2\rho)C$ queries. In the case that the node is underloaded ($A_{arrive} \leq \rho C$ and $F_{arrive} \leq (1 - 2\rho)C$), the node will answer $A = A_{arrive}$ and forward $F = F_{arrive}$ queries. So, in the case that a node is either underloaded or overloaded, it answers and forwards exactly the same number of queries under the AFS, FFS, and Null IASes.

There are two additional cases to consider. In the case that $A_{arrive} > \rho C$ and $F_{arrive} \leq (1 - 2\rho)C$, AFS, FFS, and Null IAS will all forward F_{arrive} queries. While Null IAS and FFS IAS will answer ρC queries, AFS IAS will answer $\rho C + ((1 - 2\rho)C - F_{arrive})$ queries since excess forwarding capacity will be used for answering queries. However, when $\rho = \hat{\rho}$ (as is the case in Figure 4.3), $(1 - 2\hat{\rho})C - F_{arrive}$ is 0, on average. Hence, in the average case, AFS IAS performs equivalently to FFS and Null IAS when $A_{arrive} > \rho C$ and $F_{arrive} \leq (1 - 2\rho)C$.

The final case is when $A_{arrive} \leq \rho C$ and $F_{arrive} > (1 - 2\rho)C$. AFS, FFS, and Null IAS will answer A_{arrive} queries. Null and AFS IAS will forward $(1 - 2\rho)C$ queries while FFS IAS will forward $(1 - 2\hat{\rho})C + (\rho C - A_{arrive})$. When $\rho = \hat{\rho}$, $\rho C - A_{arrive}$ is 0, on average. Therefore, FFS performs similarly to AFS and Null IAS, on average.

FFP IAS results in much less RW than the other IASes because even when queries

arrive at their destinations, they are then dropped in favor of forwardable queries.

While Figure 4.3 shows us how our IASes perform when $\rho = \hat{\rho}$ for non-uniformly distributed ids, Figure 4.5 shows us how they perform for other settings of ρ. The key observation that we make from Figure 4.5 is that AFP is the best-performing IAS irrespective of the rate at which nodes are admitting queries.

Figure 4.6 shows how various IAS policies fare at different ρ's when nodes have ids that are distributed uniformly. All of the IASes we consider perform equivalently when $\rho \leq \hat{\rho}$. When $\rho > \hat{\rho}$, an overabundance of queries are being admitted into the system and the AFP, AFS, and FFS IAS maximize RW.

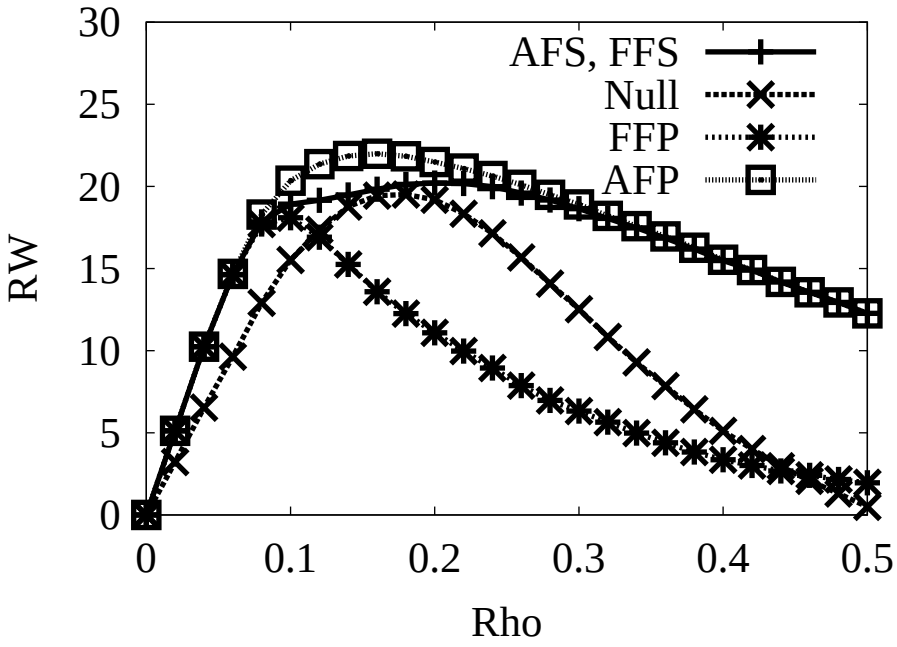

Figure 4.5: RW vs Rho for Various IASes: Non-Uniformly Distributed Ids

4.7.5 DS

DF DS maximizes RW both when ids are and are not uniformly distributed. DY DS approximates DF DS, but does not perform quite as well.

Figure 4.4 shows that the DF DS is the best of all the DSes we considered at

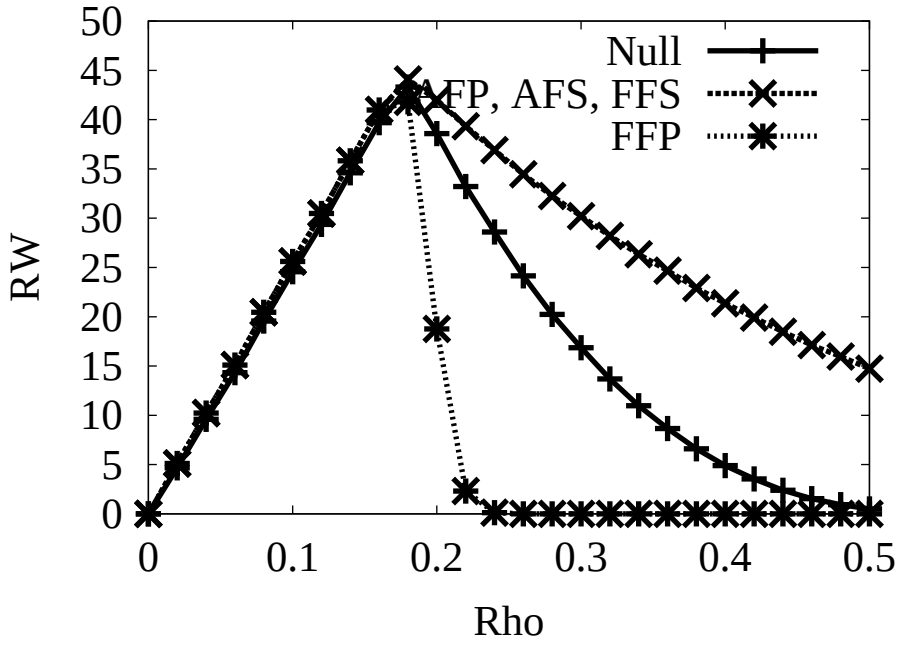

Figure 4.6: RW vs Rho for Various IASes: Uniformly Distributed Ids

maximizing remote work. The DF DS drops those queries that have the farthest clockwise distances to travel around the Chord ring. Those queries that have the longest distances to travel have the least probability of making it to their destinations due to competition for forwarding capacity at nodes between their current location and their destinations. Each node that a query needs to traverse to arrive at its destination is a potential location at which it might be dropped. By dropping those queries that have the least probability of arriving at their destinations, DF DS saves forwarding capacity that might be wasted on queries that might get dropped on the way to their destinations. The DF DS uses this saved capacity to forward queries whose destinations are the closest and thereby forwards queries that have the highest probabilities of not being dropped on the way to their destinations. The DF DS has the highest throughput, and results in the highest remote work of the DSes that we considered.

The downside of DF DS is that those queries that have to travel the farthest distances around the ring are not treated "fairly." Queries that have to travel far

IAS	$\hat{\rho}$	*Max RW*
Null	0.17	19.58
AFP	0.16	21.99
AFS	0.21	20.25
FFP	0.09	18.22
FFS	0.20	20.23

Table 4.6: Optimal Rhos and Max RWs for Various IASes: Non-Uniformly Distributed Ids

DS	$\hat{\rho}$	*Max RW*
Null	0.17	19.58
DY	0.22	22.48
DF	0.26	25.18

Table 4.7: Optimal Rhos and Max RWs for Various DSes: Non-Uniformly Distributed Ids

distances are naturally at a disadvantage because they need to travel more hops than queries that have to travel closer distances, and, in general, have a higher probability of not making it to their destinations. DF DS further exacerbates this problem by dropping such queries early in their life span. Queries that have to travel far distances, and are are dropped due to DF DS can be retransmitted, but still face a relatively high probability of being dropped upon retransmission. We study the fairness of the various policies that we consider in Section 4.7.7.

We note that DY DS achieves almost as much RW as DF DS in Figure 4.4. DY drops the youngest queries– those that have traveled the fewest hops towards their destinations, and favors the oldest queries that have taken the most hops towards their destinations. To an extent, the DY DS approximates the DF DS, and uses the hop count as an indication of how far a query has traveled. If a query is "old" and has a large hop count, it is probably very close to its destination, and will not be dropped as compared to a "young" query with a small hop count that is probably very far from its destination.

Figures 4.7 and 4.8 measure RW for different settings of ρ for the various DSes

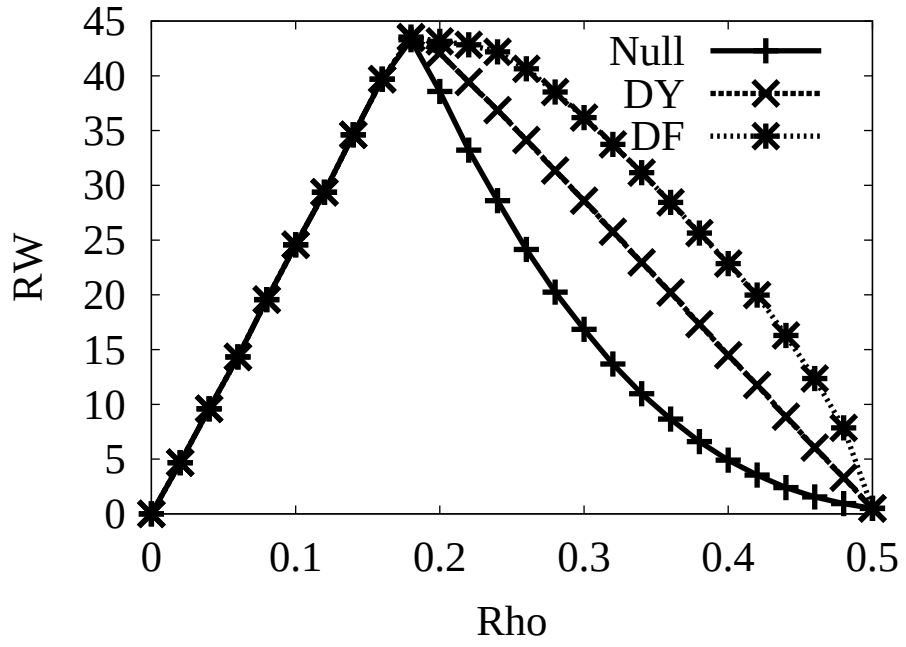

Figure 4.7: RW vs Rho for Various DSes: Uniformly Distributed Ids

in networks that do and not have uniformly distributed node ids, respectively. We find that the choice of DS is irrelevant when node ids are uniformly distributed and $\rho \leq \hat{\rho}$. We also find that DF DS results in more RW than DY DS regardless of the spacing of ids when $\rho > \hat{\rho}$. Finally, Table 4.7 shows the maximum RW attainable and $\hat{\rho}$ values for the DSes in a non-uniformly distributed network. We observe that DF DS is able to provide a relatively high maximum RW of 25.18 when $\rho = 0.26$, a setting of ρ that most other policies would not be able to perform well under.

4.7.6 Retransmissions

Retransmissions result in lower RW due to queries that must travel long distances.

Figure 4.9 graphs RW over time when retransmissions are and are not used. Retransmissions are critical in any real system, and we can see that in the steady-state, retransmissions result in a loss of RW. Retransmissions result in lower performance in the steady-state than No-RTX because queries that have to travel long paths get

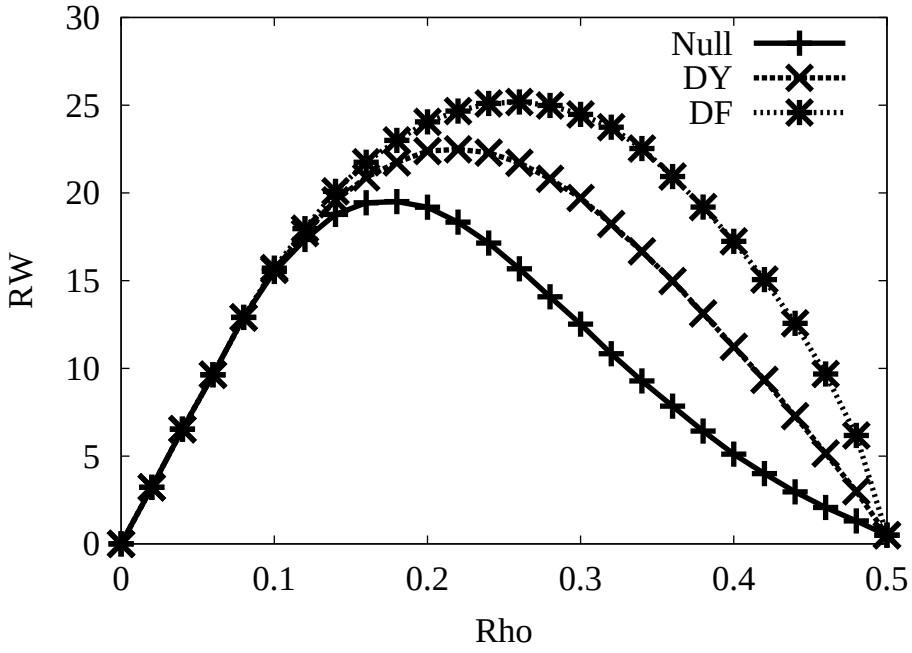

Figure 4.8: RW vs Rho for Various DSes

dropped over and over again. These queries eventually get answered, but remote work is lost because the queries that have to travel long path deny service to queries that would have to travel shorter paths.

Retransmitted queries steal forwarding capacity from queries that are transmitted for the first-time. After a query is admitted, it uses up forwarding capacity as it progresses from one node to another. If the query is dropped and retransmitted, additional forwarding capacity is used to move the query from its origin to its destination. In the steady-state shown in Figure 4.9, approximately $\frac{1}{6}$th of the total RW is lost compared to when retransmissions are not used. Of course, the users that issued the retransmissions are happy that they receive search results, at the expense of a longer latency.

In addition to resulting in decreased RW, retransmissions have a slight affect on the relative performance of our IAS policies. Figure 4.10 plots RW over time for the different IAS policies when $\rho = \hat{\rho}$ in a network with non-uniformly distributed ids, with retransmissions enabled. We can see that AFP achieves the highest performance,

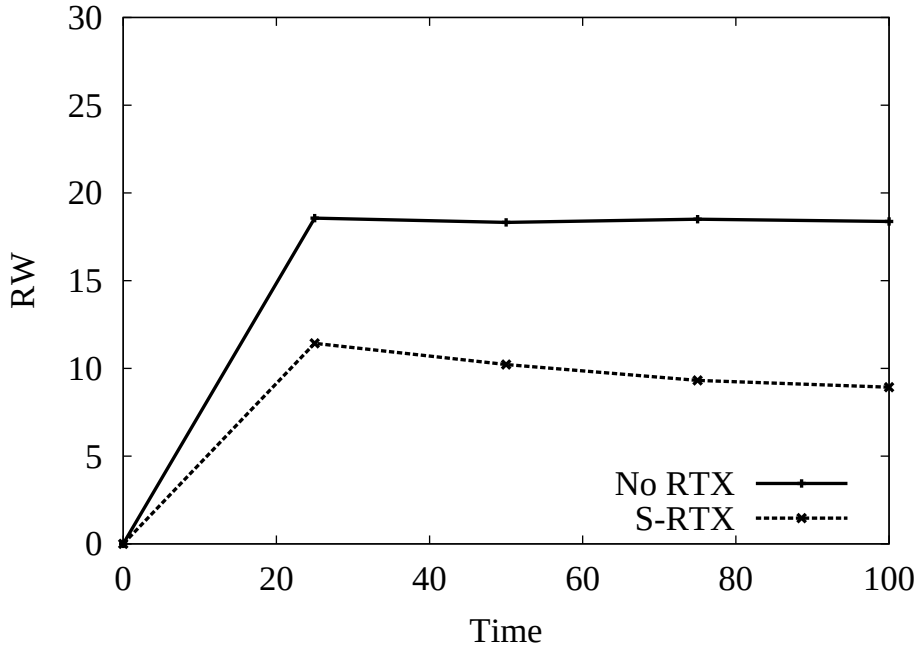

Figure 4.9: Impact of Retransmissions on RW

as before. While the AFS, FFS, and Null IASes performed about equivalently without retransmissions, AFS is the best performing, and FFS comes in next, followed by Null IAS when retransmissions are enabled. AFS performs better than FFS because it allocates excess capacity to answering queries, which results in saving unnecessary retransmissions. FFS performs better than Null because Null wastes unallocated capacity that could be used to forward queries closer to their destinations, and potentially avoid unnecessary retransmissions. Finally, we can see that FFP performs the worst and also oscillates instead of converging to a steady-state. FFP oscillates because when many answerable queries arrive at a node and they are all dropped, many retransmissions occur, causing the queries to be re-admitted, forwarded to their destinations, and dropped again. The cycle continues resulting in oscillation.

4.7.7 Fairness

DF DS provides higher RW than DR DS at the expense of decreased fairness.

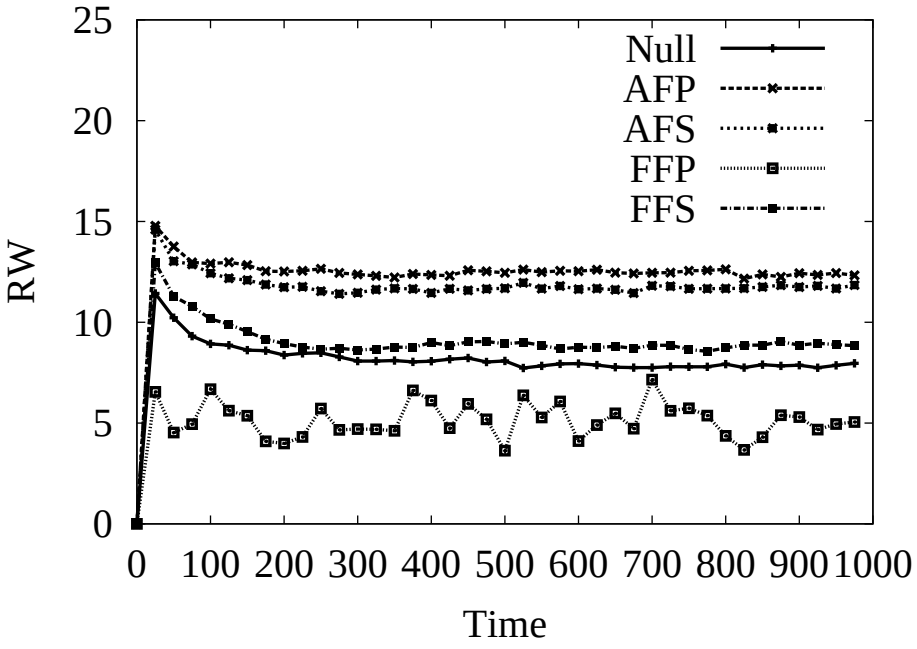

Figure 4.10: RW vs Time for Various IASes with Retransmissions

Figure 4.12 plots fairness with DR and DF DS under Null IAS.

We observe in Figure 4.12 that both DR and DF DS result are equally fair when ρ is in the range $[0, \hat{\rho}]$. In that range, few queries are dropped since there is enough answering and forwarding capacity to process most queries. Once $\rho > \hat{\rho}$, fairness suffers.

When DF DS is used, there is a drop in fairness, compared to DR DS, when ρ is greater than $\hat{\rho}$. DF DS heavily penalizes queries that have to travel long distances, and hence most queries that get answered are those that only have to travel short distances to arrive at their destinations.

4.7.8 Relative Costs

The trends that we report in earlier sections continue to hold even as the cost of answering queries increases compared to the cost of forwarding queries.

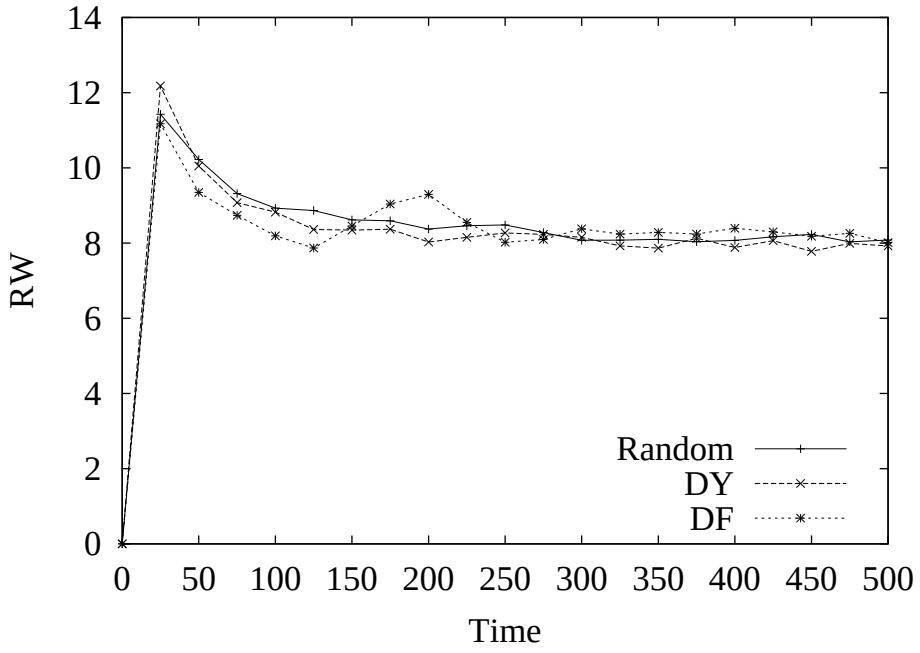

Figure 4.11: RW vs Time for Various DSes with Retransmissions

Until this point, we have run our simulations based on the assumption that forwarding queries and answering queries take the same amount of processing capacity. In a real implementation of Chord, however, it is likely that answering a query will require more processing resources than forwarding a query will require. Forwarding a query only requires a quick search through a routing table of $\log N$ entries. On the other hand, nodes might have large collections of files, and answering a query might require searching through large main memory or on-disk data structures.

In this subsection, we consider how the results of our evaluations in previous sections might differ if answering queries requires β times more processing capacity than forwarding queries. For the purposes of this subsection, we observe that admitting a query entails forwarding it along its first hop, and has approximately the same cost as forwarding a query. We can re-derive the $\hat{\rho}$ setting that nodes should use to maximize RW if answering queries is β times more expensive than forwarding (or admitting) queries. The derivation is similar to that of Section 4.2.1, except that we substitute $\beta\rho$ for ρ for the cost of answering queries. The capacity constraint is:

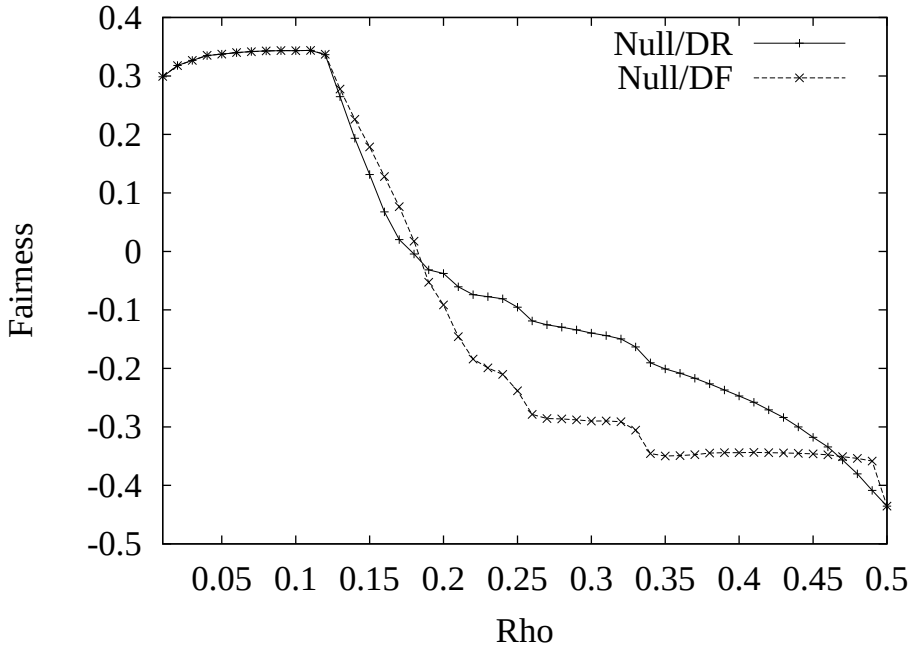

Figure 4.12: Fairness vs Rho for Various IAS/DS Combinations

$$\rho + \beta\rho + \frac{1}{2}\rho \log N = 1.$$

The solution for $\hat{\rho}$ is:

$$\hat{\rho}(\beta) = \frac{1}{1 + \beta + \frac{1}{2}\log N}.$$

Note that for $\beta = 1$, the case in which forwarding and answering requires the same processing capacity, this equation reduces to the equation we derived in Section 4.2.1.

Now that $\hat{\rho}$ can be determined as a function of β, we can run simulations comparing how our policies perform under $\hat{\rho}$ for different values of β. Figure 4.13 plots how RW varies with β for various IASes in a network that has non-uniformly distributed node ids. Even as answering queries becomes more expensive relative to forwarding queries and β increases, we can see that the basic trends that we saw in Section 4.7.4 continue to hold: when $\rho = \hat{\rho}$, AFP IAS performs the best out of all the IASes, followed by AFS,

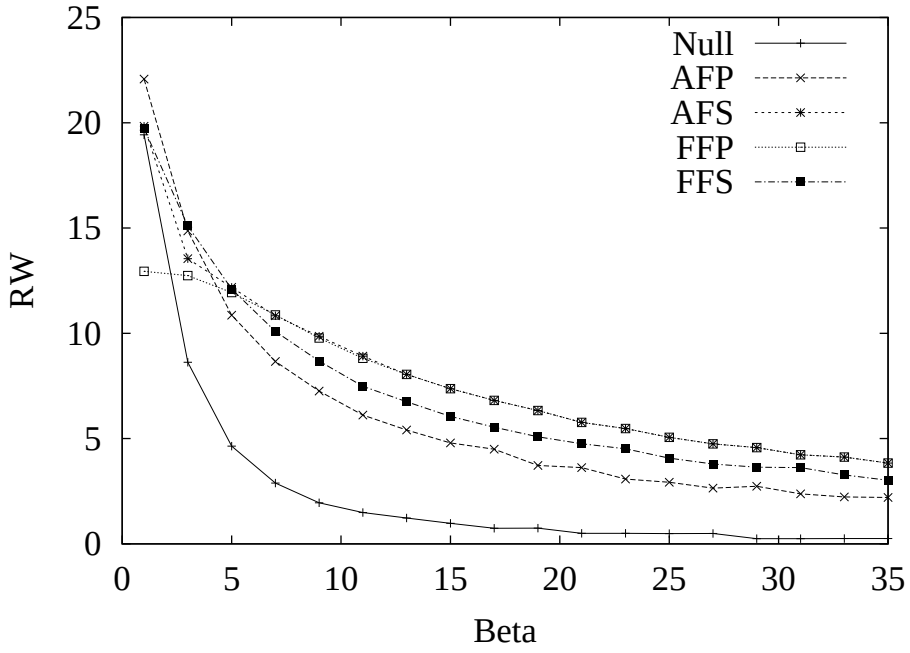

Figure 4.13: RW vs Beta for Various IASes

Null, FFS, and FFP IAS. The same trends continue to hold when ids are uniformly distributed as can be seen in Figure 4.14, although the difference between AFP and AFS IAS, and between a number of other IASes becomes marginal.

4.7.9 Malicious Nodes

IASes and DSes that improve performance also help deal with malicious nodes in Chord.

Figures 4.15 and 4.16 measure how RW varies for increasing numbers of malicious nodes for different IASes and DSes in a network with a total of 256 nodes. The figures show that the AFP IAS and the DY and DF DSes result in slightly more RW than our baseline policies when there are malicious nodes in the graph.

Note that this is in contrast to our results for flooding-based networks. In Gnutella, we saw that Null IAS with PreferHighTTL DS (our baseline policies) maximized RW when no malicious nodes were present but resulted in an abysmal effect when

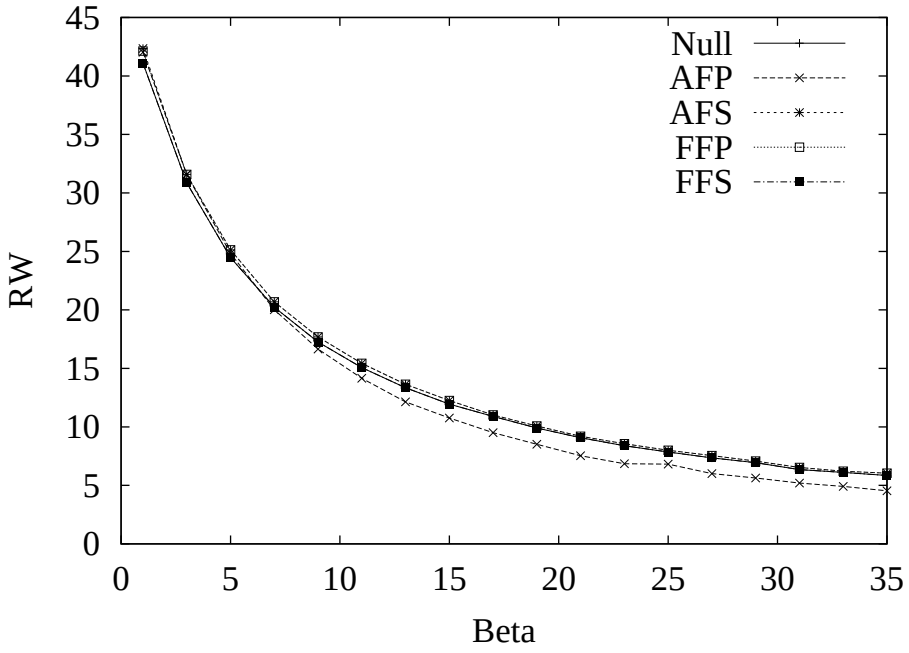

Figure 4.14: RW vs Beta for Various IASes: Uniformly Distributed Ids

malicious nodes were present in the network. The IASes that we used to contain attacks in Gnutella, on the other hand, performed well when malicious nodes were present, but resulted in less RW than the baseline policies when no malicious nodes were present. In Chord, the baseline policies' RW performance drops off sub-linearly with the number of malicious nodes, and the AFP IAS and non-random DS policies provide an incremental improvement in RW both *when malicious nodes are and are not present*. Hence, it makes sense to use them whether or not malicious nodes are present.

Traffic limits can increase RW significantly in the presence of malicious nodes, and can be used to virtually eliminate "flood" loss due to malicious nodes.

In Section 4.5, we developed traffic limits based on the expected query load in a Chord network to mitigate the effects of malicious nodes blasting queries. We evaluate the effectiveness of the traffic limiting techniques in this subsection.

Figure 4.17 was generated by running simulations in which we measured RW as the number of malicious nodes increased in a network with uniformly-balanced ids.

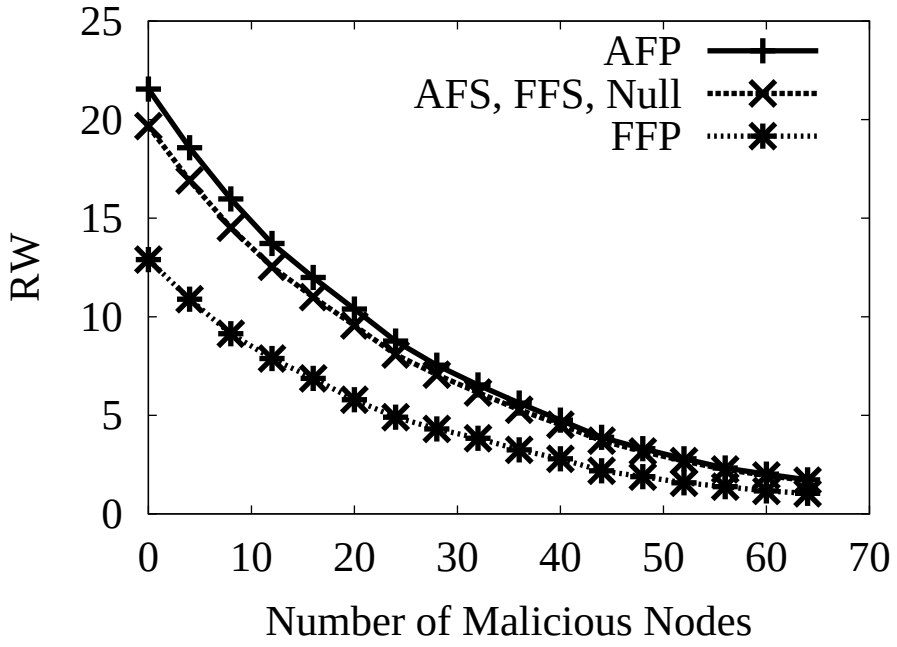

Figure 4.15: RW vs Number of Malicious Nodes for Various IASes

There are three curves plotted in the figure. The "No Limits" curve shows the RW in a system where nodes *do not* impose the admission or forwarding traffic limits. The "With Limits" curve plots RW when nodes use both the admission and forwarding traffic limits. Finally, the "Ideal" curve plots the RW that would result if good nodes could use an oracle to help them decide whether or not to process a query. That is, upon receiving a query, a good node can submit the query to an oracle, and decide to process it only if the oracle reveals that the query was admitted by another good node.

From the figure, we can see that imposing traffic limits can result in a 25 percent or more increase in RW. For instance, if there are 32 malicious nodes in the network, the RW is only 15 when no limits are used, whereas imposing limits results in a RW of 19.4, an increase of 29.3 percent. From Figure 4.17 we can also see that imposing traffic limits results in a RW that is very close to the "Ideal" RW that could be achieved if good nodes used an oracle. The distance between the "With Limits" and the "Ideal" curve is greatest at 12 to 16 malicious nodes, and even then so, imposing

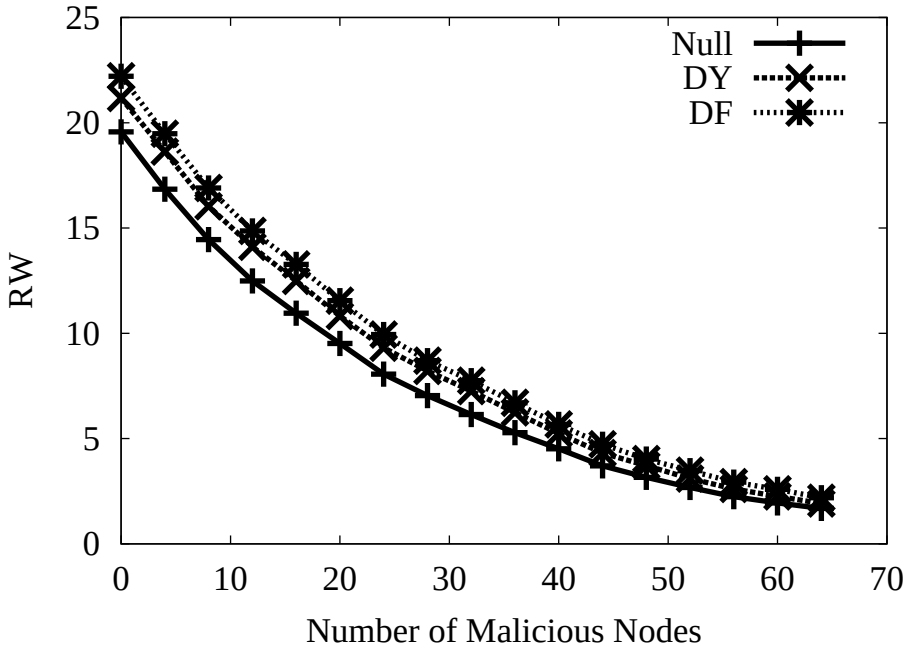

Figure 4.16: RW vs Number of Malicious Nodes for Various DSes

traffic limits results in 97 percent of the RW achievable of the ideal. Hence, traffic limits are effective at screening out excessive queries sent by malicious nodes.

However, the mere presence of the malicious nodes in the network has an impact on RW even if most of their excessive queries can be filtered out. The loss in RW due to the malicious nodes has two major causes: 1) queries that happen to be forwarded to malicious nodes as they attempt to traverse the path from their source to their destination are dropped at the malicious nodes, and 2) legitimate queries that arrive at good nodes may be dropped in favor of processing useless queries that were admitted by malicious nodes. The loss in RW that occurs as a result of these causes is similar to structural and flood damage defined in Chapter 2. We will analogously refer to causes (1) and (2) as *structural loss* and *flood loss*, respectively.

In the simulation that was used to generate Figure 4.17, most of the loss in RW is structural loss. When there are no malicious nodes in a network with uniformly-balanced ids, the RW that can be achieved is 42.7. However, even in the ideal case that good nodes can use an oracle to completely eliminate flood loss, a significant

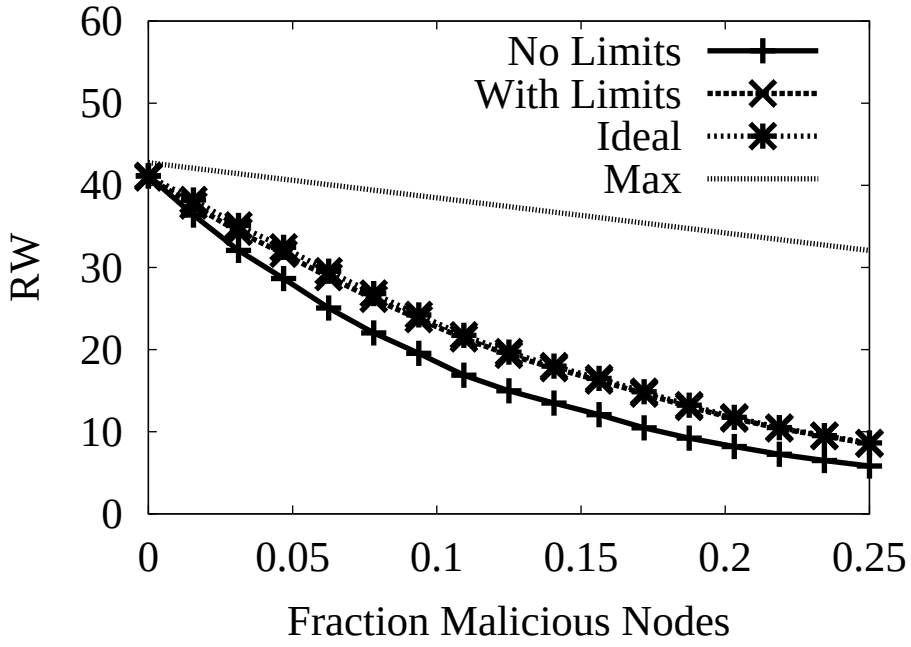

Figure 4.17: RW vs Number of Malicious Nodes Using Traffic Limits

amount of RW is lost simply due to the presence of malicious nodes that drop all queries that are forwarded to them. For instance, if even just under 10 percent of the nodes in the network are malicious (24 malicious nodes), the maximum RW that can be achieved with an oracle is 24.2. Over 45 percent of the RW lost is structural. While imposing traffic limits can recover approximately 4.7 units of RW due to flood loss, 18.5 units of RW of structural loss can be recovered by eliminating malicious nodes from the network. In this particular case, flood loss only accounts for 20 percent of the total loss; the remaining 80 percent of the loss is structural. Hence, further work must be done to detect malicious nodes and eliminate them. Nevertheless, the traffic limits that we propose are a significant first step that contains the effects of malicious nodes while the good nodes are in the process of detecting and eliminating malicious nodes.

4.8 Chapter Summary

In this chapter, we extended the model and metrics that we used in previous chapters to study performance and denial-of-service issues in DHTs, and we specifically focused on the Chord DHT. We developed a number of IASes and DSes for Chord, and evaluated them using our model and metrics. We modeled malicious nodes that blast queries into the network with the intent of denying service to legitimate queries, and developed traffic limits to mitigate such a blasting attack.

Specifically, we found that:

- AFP IAS maximizes RW.
- DF DS maximizes RW both when ids are and are not uniformly distributed. DY DS approximates DF DS, but does not perform quite as well.
- Retransmissions result in lower RW due to queries that must travel long distances.
- The trends that we report in earlier sections continue to hold even as the cost of answering queries increases compared to the cost of forwarding queries.
- IASes and DSes that improve performance also help deal with malicious nodes in Chord.
- Traffic limits can increase RW significantly in the presence of malicious nodes, and can be used to virtually eliminate "flood" damage due to malicious nodes.

In the past three chapters, we have studied application-layer denial-of-service and performance issues in the unstructured Gnutella network and the structured Chord P2P network. In the next chapter, we will complete our investigation of application-layer denial-of-service in P2P networks by studying similar issues in a non-forwarding P2P network called GUESS.

Chapter 5

Pong-Cache Poisoning in GUESS

In the previous chapter, we studied application-layer DoS attacks in DHTs. In both DHTs and unstructured P2P systems (as studied in Chapter 2 and 3), nodes maintained persistent connections with other nodes, and routed queries over them. In this chapter, we study application-layer DoS attacks in non-forwarding P2P systems in which nodes do not maintain persistent connections to other nodes. In particular, attackers can manipulate the resource discovery mechanisms in non-forwarding protocols in order to conduct DoS attacks.

In this chapter, we propose policies that make the discovery of resources (or nodes) resilient to coordinated attacks by malicious nodes. For concreteness, we focus on a novel P2P protocol called GUESS (Gnutella UDP Extension for Scalable Searches) [44] that is under consideration by the Gnutella Development Forum (GDF) [73], a grouping of independent Gnutella software vendors, for use in the Gnutella P2P network. The Gnutella file-sharing network is one of the most widely used P2P networks with over 100,000 concurrent users on the network at any one time offering 5 to 10 terabytes of files. In addition, over 10 distinct vendors have deployed Gnutella application software and over 35 million copies of these applications have been downloaded by users. We also note that other extremely popular P2P protocols such as FastTrack and eDonkey already use protocols similar to GUESS for resource discovery, and hence could also benefit from the techniques we develop in this paper.

As opposed to traditional Gnutella networks in which flooding is used to propagate

queries through a software overlay topology, nodes in a GUESS network explicitly query other nodes one at a time. The protocol design was partially motivated by results on random walks in unstructured P2P networks [115]. In GUESS, each node keeps a cache of other nodes that are available to accept queries, and sends its queries to one of the nodes in its cache at random. Nodes are required to manage the cache by deleting nodes that are no longer available, and adding new nodes that are available. Nodes exchange information about which nodes are available by exchanging "ping" and "pong" messages, and, as a result, the caches that nodes use to store node ids of other nodes are called "pong caches."

Initial evaluations [15] suggest that GUESS can be a significantly more efficient search mechanism, compared to basic Gnutella, essentially because flooding is replaced by directed querying that can be throttled by the originating node. Furthermore, queries are not routed through untrusted third parties, so GUESS nodes are less susceptible to attacks by such intermediaries.

However, GUESS does have a new potential security weakness: the pong caches. For reasons that we describe in Section 5.2, malicious nodes will be interested in having their node ids appear in the pong caches of good nodes as a pre-cursor to many forms of attack. When ids of malicious nodes appear in the pong cache of a good node, we say that the good node's pong cache has been *poisoned*. And once a cache is poisoned, it becomes very hard for a good node to find other good nodes that may provide useful results. In this chapter we focus on this pong-cache poisoning problem, and suggest techniques for making nodes more resilient to this attack.

The techniques that we propose are *complementary* to existing security mechanisms, and help us implement defense-in-depth within the context of a GUESS network. In particular, as discussed in the FLI model of Chapter 1, a number of categories of defenses are required:

- *Prevention.* These defenses make it hard for malicious nodes to participate in the system. For instance, we can require that nodes obtain certified node ids, as proposed in [26], to limit the absolute number of malicious nodes, and/or require nodes to solve crypto-puzzles to obtain node ids with the intent of slowing down the rate at which malicious nodes can join.

- *Detection.* These defenses identify malicious nodes that have joined and attempt to remove them from the network and/or revoke their privileges. For instance, one can use a collaborative trust system to identify nodes that are providing bad results or misbehaving in other ways.

- *Containment.* Finally, these defenses attempt to minimize damage from nodes that have entered the system and have not yet been detected. Our techniques to deal with pong-cache poisoning are in this category.

We believe that for best results all three types of defenses must be employed in GUESS, or in any large scale distributed system. Since prevention and detection cannot be expected to work perfectly in practice, it is safer to design the GUESS cache management policies under the assumption that a few malicious nodes have gotten through and may poison caches. Indeed, as we will see in Section 5.4, even just a few malicious nodes can poison a large fraction of all the caches in the network, if good cache management policies are not used. Thus, it is critical to have strong containment policies in place. Of course, our policies will only reduce pong-cache poisoning instead of eliminating it altogether. As such, the policies we propose and evaluate provide practical security; they do not necessarily provide "provable" security.

Note that we are conducting this work early in the evolution of GUESS instead of waiting after its initial deployment, such that the protocol will not have to be retrofitted with security features only after the fact. Some of the decisions that nodes need to make are currently unspecified in the existing protocol specification, and we fill in these gaps with recommendations on how nodes should make these decisions to deal with attacks.

Also note that while we focus on the ping-pong resource discovery protocol used in GUESS, the need to maintain caches of available peers also arises in other contexts, such as wireless ad-hoc networks, grid computing, and autonomic computing. Thus, we believe that our work can be generalized in a straightforward fashion to be applied in these other areas.

Our specific contributions in this chapter are:

- We define a model that captures how ping and pong messages affect pong cache

behavior, and we describe the key decisions that nodes must make as they interact with other nodes (Section 5.1).

- We define expected behaviors for good and malicious nodes, and we outline the key research goals that need to be addressed to deal with pong-cache poisoning in GUESS (Section 5.2).
- We propose new mechanisms and improvements to existing mechanisms in the protocol to mitigate attacks by malicious nodes. These mechanisms control how nodes should initialize, insert, delete, and replace entries in their caches (Section 5.3).
- We evaluate the impact that malicious nodes have by poisoning pong caches of good nodes, and how the various policies that we propose mitigate poisoning (Section 5.4).

5.1 Basic Model

Our model considers a set of n nodes in a peer-to-peer network. Initially, the set of nodes participating in the network is $N = \{N_1, N_2, ..., N_n\}$. Some subset of the nodes in N are *good*, while others are *malicious*. Good nodes, in general, follow the protocols we describe, while malicious nodes may or may not. We assume that G denotes the set of good nodes, and that M denotes the set of malicious nodes. Of course, at any instant $G \cup M = N$ and $G \cap M = \emptyset$.

Not all of the nodes in the system are expected to be available at the same time. Indeed, we expect some percentage of the nodes to be unavailable most of the time, although we expect the specific set of nodes that are unavailable at any given time to vary. Some typical reasons for unavailability could be node failure, overload due to a denial-of-service attack, or (if the node is a mobile device) it may be simply out of range of some of the other nodes in the network. We say that a node that becomes unavailable has "died." This "death" may not be permanent, but from the standpoint of resource discovery, the node will need to be re-discovered by other nodes once it becomes available again.

At some instant in time, a set of nodes $\Delta \in G$ may die. At the same time that the set of nodes in Δ die, a set of nodes, B, can be born. The new set of nodes in the system is $N' = (N - \Delta) \cup B$.

A node N_i may rely on another node N_j from time-to-time to service its queries or provide some other network service. However, node N_j may die without explicitly informing N_i of its death. We assume that node N_i keeps a list of nodes that it typically relies on. In a GUESS network, this list is called a *pong cache*. Each node N_i has a fixed-size pong cache, $P(N_i)$, associated with it where $P(N_i)$ is a set of node ids. The set $P(N_i)$ may change over time.

A node N_i may decide to *ping* another node N_j to determine if it is still available. A ping is simply a "no-operation" query to which a node is expected to respond if it is available. Since message passing incurs network and computational overhead in real systems, a ping may contain a request to process a query instead of just being a "no-op." When node N_j receives the ping, it may respond to N_i with a *pong* that tells N_i that N_j is "live."

In our model, a node N_i may choose to ping any node $N_j \in P(N_i)$ in its pong cache. Node N_i chooses which N_j according to a *ping probe policy*. If node N_j temporarily or permanently dies, N_i may not receive a response to its ping after some timeout period, and N_i may decide to remove N_j from its cache. Node N_i could also decide to keep the cache entry around in the hope that N_j may become available again at some time later.

As part of the pong that N_i receives from N_j, N_i receives a set of node ids $S = \{N_{k1}, N_{k2}, ...N_{k|S|}\}$ where $N_{ki} \in N, 1 \leq i \leq |S|$. The nodes in set S are chosen by N_j from its pong cache ($S \subseteq P(N_j)$) using a *pong choice policy*.

Upon receiving S, N_i may replace any subset of $P(N_i)$ with any subset of S to create $P'(N_i)$, node N_i's updated pong cache. Let X be any subset of $P(N_i)$, and Y be any subset of S. Then, $P'(N_i) \leftarrow (P(N_i) - X) \cup Y$. For example, since the empty set is a subset of $P(N_i)$, and Y could be exactly equal to S, N_i could simply add all of the entries received from N_j to its pong cache. Of course, pong caches in real systems typically have a fixed-size, and require different choices for X and Y. We will refer to the exact method that node N_i uses to determine the sets X and Y

as the *cache replacement policy*.

When a node N_k is newly born, its pong cache, $P(N_k)$, needs to be initialized. We say that N_k's pong cache needs to be "seeded" when the node is born, and we explore various *seeding policies* in Section 5.3.1.

It is important not only for new nodes to have their caches seeded with existing node ids when they are born, but we believe that it is equally as important for existing nodes to find out about new nodes that have joined the network. As we will see in Section 5.4, after some amount of time a majority of the entries in a node's pong cache can become invalid due to the deaths of the corresponding nodes. To eliminate this problem, we propose adding an *introduction protocol (IP)* to GUESS in which a new node N_k may want to make itself available to existing nodes by having its id added to their pong caches. In Section 5.3.2, we describe an IP that can be used to accomplish this.

Figure 5.1 shows an example of how resource discovery takes place in a GUESS network. The large, rounded squares represent nodes N_1 and N_2, and the box within each circle represents that node's pong cache. The top part of the figure shows the state of the nodes before any interaction between them takes place. For instance, initially node N_1 has the node ids of nodes N_2 and N_3 in its pong cache. The figure shows what happens when node N_1 pings N_2. In particular, N_1 chooses N_2 from its pong cache, and sends N_2 a ping message. N_1 could have alternatively chosen to ping N_3 since N_3 is also in its pong cache. In response to the ping, N_2 chooses a set, say $S = \{N_4\}$, from its pong cache. N_2 could have also potentially chosen any of the other three possible subsets of $P(N_2)$ as its response to the ping. In the example in Figure 5.1, N_1 choses to update its pong cache by replacing the N_3 entry ($X = \{N_3\}$) with N_4 ($Y = \{N_4\}$). The bottom part of the figure shows the resulting state: Node N_1 computes $P'(N_1) = (P(N_1) - X) \cup Y = \{N_2, N_4\}$ as its updated pong cache.

We may view the model above in one of two (equivalent) ways. In one view, such a network has no overlay topology, and queries are sent to nodes whose ids are chosen from pong caches in an ad-hoc fashion. This view of the system is similar to that which we would find in an ad-hoc wireless network. Certain nodes may join and leave the system based on the mobility of nodes in the wireless network. When

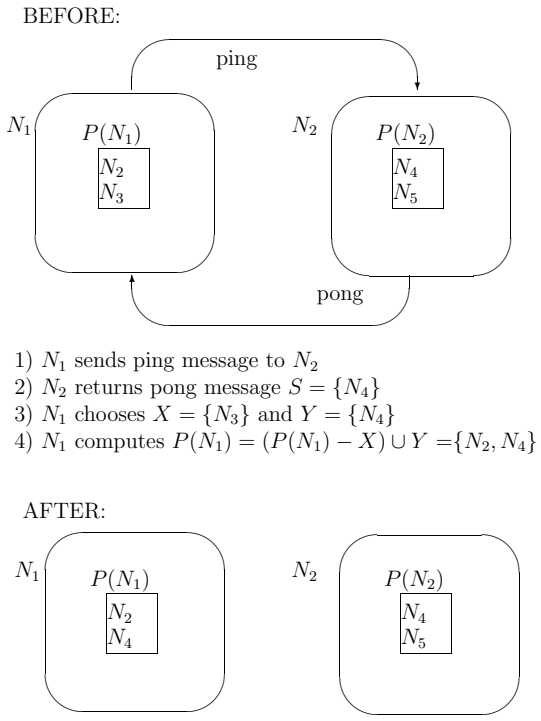

Figure 5.1: An example of a ping and pong between two good nodes in a GUESS network.

a node comes within range of other nodes, we model this as a "birth," and when a node leaves the range of other nodes, we model this as a "death." In this type of an application, cache size and communication costs may be limiting factors due to the constraints of mobile devices.

Another way of thinking about a GUESS network is that it does have an overlay topology which evolves quite dynamically. For instance, one could consider the node ids in a given node's pong cache to be the nodes to which it is "connected." If node N_1 has N_2 in its pong cache ($N_2 \in P(N_1)$), we might say that node N_1 is connected to node N_2. When N_1, say, replaces N_2 with N_3 in its pong cache, we consider this equivalent to N_1 breaking a connection with N_2 and establishing a connection with N_3. In this view, the set of neighbors that a node has is larger and more dynamic than in traditional Gnutella, with the exception that we do not flood, and queries are only processed by direct neighbors. If nodes in such a system are desktop PCs or high-end servers on a LAN, they may be less transient than nodes in the wireless scenario above, and connections between them may be longer-lived. In addition, such nodes may be able to have relatively large cache sizes and numbers of connections, and can run more complicated algorithms to decide when to add or remove cache entries and make or break connections.

5.2 Threat Model

While there might exist various prevention and detection mechanisms in a GUESS system, some number of malicious nodes may still be able to join. For instance, we might require that nodes obtain certified node ids as propsed in [26]. However, Sybil attacks might still be possible. A Sybil attack [59] is one in which a malicious adversary generates or obtains many node ids (identities). A malicious adversary could do so by paying for node ids, or compromising the security of existing hosts that have valid node ids. In our threat model, there is some non-negligible cost (i.e. time and/or money) for obtaining a node id, and malicious nodes can acquire some limited number of node ids.

While there are many possible attacks that can be conducted by malicious nodes

that have taken control of node ids, we focus on one key class of attacks in this paper. In particular, we will be most concerned with malicious nodes that may collude to conduct "pong-cache poisoning" attacks in which they attempt to propagate malicious node ids in as many pong caches as possible. We believe that the poisoning of pong caches will be the first step in many forms of distributed attacks in a GUESS system.

In a pong-cache poisoning attack, malicious nodes respond to pings with node ids of other malicious nodes. In addition, all malicious nodes in the system have the ability to collude as a single large conspiracy. Doing so gives them a maximal advantage in poisoning pong-caches with as many distinct malicious node ids as possible.

Each malicious node is aware of the node ids of all other malicious nodes in the system. When a malicious node receives a ping, it responds with a pong that contains malicious node ids chosen from the entire set of all possible malicious node ids.

Finally, in our model, we assume that malicious nodes send the same number of ping messages that good nodes do. If an IP is used by good nodes, malicious nodes have an incentive to issue pings as doing so can result in malicious node ids being introduced into the pong-caches of good nodes. We can account for attackers that send more pings than good nodes as multiple malicious nodes. For instance, if "average" good nodes issue C pings per second, and an attacker issues $2C$ pings per second, we can model the attacker as two malicious nodes. We also note that attackers that send out too many pings make it easier for them to be identified and ignored. Hence, malicious nodes have an incentive to issue a number of pings that is comparable to the number issued by good nodes.

Once good nodes rely on the services of malicious nodes, the malicious nodes will have many options available to them as to what type of damage they would like to incur. For instance, the malicious nodes could conduct a distributed denial-of-service attack in which they all decide to "die" at the same time, thereby creating a situation in which good nodes have a large number of "dead" cache entries and are unable to service queries from their clients due to the resulting network fragmentation and/or partition. Alternatively, the malicious nodes could continue to offer service to the good nodes, but all respond with inauthentic copies of documents. We do not consider these post-poisoning attacks here. We only cite them to motivate why

containment of poisoning is so critical and why techniques that specifically address containment should be deployed as an important line of defense in a GUESS system.

Our first goal is to maximize the number of live node ids in pong caches in the steady-state. We say that a GUESS system has achieved "p percent liveness" when over p percent of the node ids in all of the pong caches of good nodes in the system are not dead.

Our second goal is to mitigate pong-cache poisoning, for the reasons described above. We have two sub-goals: 1) limit (upper bound) the number of cache entries containing malicious node ids in the steady-state, and 2) reduce the rate at which poisoning occurs.

Note that we are interested in the steady-state behavior of the protocol because if poisoning cannot be controlled in the steady-state in our idealized model of the protocol, there may be little hope of controlling poisoning in an oscillating, real-world system that implements the protocol. However, if we can find policies that work well in the steady-state in our idealized model, such policies may be good candidates for experimentation with in real-world systems.

5.3 Protocol Policies

In this section, we describe in more detail the various protocol policies required in GUESS. Table 5.1 summarizes the policies and the options that we explore in this paper.

5.3.1 Seeding Policy (SP)

When a new node wants to join the network (e.g. when it is born), it must initialize, or *seed* its pong cache.

We study three seeding policies (SPs), out of many possible such seeding policies, in this paper.

The first SP we study is a *random-friend (RF) policy*. In this SP, we assume that each newly born node N_b knows of the existence of one other live, "friend" node

N_j ($N_j \in N - \Delta$). Each newly born node may know of the existence of a different friend node. The newly born node copies the pong cache of this live, friend node: $P(N_b) = P(N_j)$. The node N_j could, of course, be malicious. We choose to study this policy in this study due to its ease of implementation in real systems.

The second SP we study is a *popular-node (PN) policy*, which is similar to the RF policy, except that all nodes have the same "friend." That is, each newly born node knows of the existence of the same node N_1. We assume that N_1 is not a malicious node, else all of the newly born nodes would have all of their entries poisoned immediately. However, some of N_1's entries may become poisoned as it issues pings and processes pongs itself. Finally, we assume that N_1 never dies.

The third SP we study is a *trusted directory (TD)* policy. In particular, we assume the existence of a node that serves as a trusted directory for N_b. Each newly born node N_b may have access to a different trusted directory, although we imagine that there will be a small, finite number of trusted directories.

The directory is trusted with the task of returning k live (but some or all of which may be malicious) node ids. The parameter k may be equal to the cache size of the newly born node. The directory is trusted to not return dead node ids. When nodes are born, they "register" with the trusted directory, letting the directory know that they are available to service queries. Nodes are not required to inform the trusted directory when they leave the network. Instead, before a trusted directory returns a node id to a newly born node, the trusted directory pings the nodes whose ids it intends to return, and only returns ids of nodes that are live.

In a practical system, a trusted directory continuously pings nodes on a periodic basis, and returns node ids that have been successfully pinged recently instead of pinging nodes each time before their ids are returned to newly born nodes.

In an alternative approach, a trusted directory could provide a newly born node with a random assortment of node ids, and transfer the responsibility of testing the node ids returned to the newly born node. However, we assume that newly born node ids have small memories compared to trusted directories, and we would not want to fill the memories of newly born nodes with potentially dead ids. We assume that trusted directories have significantly more memory and CPU resources than regular

nodes, and provide the service of returning node ids live with very high probability to newly born nodes. (It is possible that a node whose id is returned by a trusted directory could die in between the time that the trusted directory transmits a pong and the time that it is received. However, we assume that pong message transmission time is relatively low compared to the rate at which nodes die, so we are not overly concerned with this case.)

While we are not convinced of the viability of such a trusted directory in a peer-to-peer network, we include these relatively powerful nodes as one potential seeding option in our study as a point of comparison.

5.3.2 Introduction Protocol (IP)

Nodes that are newly born do not have their node ids appear in the pong caches of any of the existing nodes. While newly born nodes can query existing nodes because they are seeded appropriately upon birth, GUESS also needs a mechanism by which existing nodes can be given the opportunity to query newly born nodes. In this section, we propose a mechanism by which GUESS may "introduce" the node ids of newly born nodes into the pong caches of existing nodes. An introduction protocol (IP) is not part of the original GUESS protocol, but we show that an IP is essential to achieve a steady-state in which most cache entries are live.

The GUESS specification envisions deploying GUESS in a hybrid fashion together with the traditional Gnutella flooding protocol. In such a scenario, existing nodes can resort to broadcasting ping messages to allow them to discover newly born nodes. However, doing so requires that the traditional protocol (and all of the performance and security problems associated with it) be carried forward as the network evolves to incorporate GUESS. Hence, we feel it is important to study how to use introductions such that node discovery could take place without any reliance on the traditional protocol.

A natural opportunity at which to introduce a newly born node to an existing node is when a newly born node N_i pings an existing node N_j. After responding to the ping, the node N_j enters N_i into its cache with some probability p, $0 \leq p \leq 1$. N_j

uses its cache replacement policy (see Section 5.3.4) to decide which existing entry in its cache to replace. Note that $p = 0$ is equivalent to using no introduction policy at all. If $p = 1$, an introduction always occurs. By choosing a value of p between 0 and 1, we can vary how "aggressive" we would like to be with introducing new node ids into the system.

If an IP is not used, malicious nodes have no incentive to issue pings since it will not help them propagate their node ids into the caches of good nodes. If an IP is used, however, then malicious nodes *do* have an incentive to issue pings, as it can help them propagate their node ids into the caches of nodes pinged.

5.3.3 Ping Probe and Pong Choice Policies (PPP and PCP)

A node N_i must decide which node N_j to ping. The algorithm that a node uses to decide which node N_j to ping is its *ping probe policy (PPP)*.

All of the entries in N_i's pong cache are "marked" as either *live* or *dead*. If N_i has never pinged a particular entry N_j, it marks the entry live. If N_i pings a particular entry N_j and does not receive a pong within some timeout period, it marks the entry dead.

There are many possible policies that may be used to choose a server to ping. For instance, a client may choose to ping a server at random, or it may choose the server that produced the most search results to its last query.

When N_i chooses a node id N_j from its pong cache to ping, it might be the case that node id N_j has died in between the current time and the last time that N_j was pinged. Such a choice for N_j is allowed since N_j is not marked dead yet. Once N_i discovers that N_j has died, N_j is marked dead, and will not be chosen a second time.

All nodes respond with up to $|S|$ node ids that are not marked dead when they receive a ping. The policy that nodes use to determine exactly which $|S|$ node ids to respond with is called the *pong choice policy (PCP)*.

There are many alternatives for a pong choice policy that one may choose from including choosing node ids randomly, choosing the most recently pinged node ids, or choosing the node ids that have returned the most number of results. In this

paper, we experiment with LRU, MRU, Youngest, Oldest, and Random PCPs. (We also experiment with the same policies as cache replacement policies, and we describe how they work in the next section.)

Note that different choices for PPPs and PCPs may have an affect on search performance, and various options for PPPs and PCPs are studied in [15].

While we study the PCP policies above, in real GUESS systems, it might be useful to consider other policies as well. For instance, in a real system, pings and pongs may "piggyback" on top of query and response messages. If N_j does not have any search results in response to a query from N_i, N_j may include node ids that might have the requested results in a pong message. Node N_j might choose which node ids to include in the pong message based, for example, on which nodes have responded with the most results to N_j's own queries. Node N_j might select such node id based on a routing index as proposed in [36]. However, the types of routing indicies we may want to use in GUESS networks will differ from those used in traditional Gnutella networks. In particular, since nodes do not have queries and responses routed *through* them, GUESS routing indicies may amount to being a local cache of results received by node N_j.

5.3.4 Cache Replacement Policy (CRP)

Once N_i receives up to $|S|$ node ids from N_j, it chooses to replace $r \leq |S|$ entries in $P(N_i)$ to form $P'(N_i)$. In other words, N_i chooses $Y \subseteq S$ and $|X| = r$. Node N_i first replaces any node ids in its pong cache that are marked dead. Node N_i first replaces dead cache entries with those from the set Y. N_i then chooses the contents of the set X using a CRP. We consider six CRPs in this paper:

Random. N_j randomly chooses up to $|S|$ entries from its pong cache $P(N_j)$. Node N_j excludes any entries that are marked dead from its choice.

Most Recently Used (MRU). N_j maintains a *last-time-pinged* timestamp associated with each cache entry, and returns the $|S|$ non-dead entries in $P(N_j)$ that have the *highest* last-time-pinged timestamps. The last-time-pinged timestamp is set to the current time when a cache entry is chosen to be pinged.

Least Recently Used (LRU). N_j maintains a *last-time-pinged* timestamp associated with each cache entry, and returns the $|S|$ non-dead entries in $P(N_j)$ that have the *lowest* last-time-pinged timestamps. The timestamp is updated similarly.

Oldest (FIFO). N_j maintains a *time-inserted* timestamp that records the timestamp at which each cache entry was inserted into the cache, and returns the $|S|$ non-dead entries in $P(N_j)$ that have the *lowest* time-inserted timestamps. The time-inserted timestamp is never updated.

Youngest (LIFO). N_j maintains a *time-inserted* timestamp, and returns the $|S|$ non-dead entries in $P(N_j)$ that have the *highest* time-inserted timestamps. As before, the time-inserted timestamp is never updated.

5.3.5 ID Smearing Algorithm (IDSA)

In addition to the simple CRPs just described, we develop a more sophisticated cache management policy to help deal with malicious nodes. In particular, if node N_i receives a node id N_k as a response in pongs from many nodes, then it may be the case that either 1) N_k is a malicious node and is working to have its id propagated to as many good node caches as possible as a pre-cursor to an attack, or 2) N_k is a good node that is likely to become overloaded with too many queries because other nodes happen to be including N_k's id in pongs. In either case, it is in the best interest of the good nodes to balance out the number of times that any particular node id appears in the pong caches of other good nodes. That is, good nodes want to replace entries in their pong cache such that for any of the n node ids in the system, each of the ids appears in the pong caches of good nodes with a frequency proportional to $1/n$.

To help accomplish this, we use an "ID smearing" algorithm (IDSA). The algorithm evenly "smears" all of the available node ids across all of the pong caches by doing the following locally at each node. When N_i receives N_k in a pong message from N_j, it checks whether or not N_k already appears in its cache, $P(N_i)$. If $N_k \in P(N_i)$, then there may be too many copies of N_k in the network, and we set $P'(N_i) = P(N_i) - \{N_k\}$. For example, if $P(N_1) = \{N_2, N_3\}$, and N_1 receives N_3 in a pong message from N_2, N_1 updates its pong cache such that $P'(N_1) = P(N_1) - \{N_3\} = \{N_2\}$.

We call this "light-state" ID smearing for reasons that will become clear shortly.

A node may or may not have the ability to store extra state in addition to that which is used for its pong cache. We study one IDSA ("light-state") that does not require any additional state beyond the pong cache, and two policies that *do* use additional state. We will refer to the two policies that do use additional state as "heavy-state" IDSA. In particular, in "heavy-state" IDSA, each node N_i maintains a list $L(N_i)$ of all node ids that have ever appeared in its pong cache (whether or not the entries currently appear in the pong cache).

Note that pong cache entries might contain additional state such as last-time-pinged timestamps and time-inserted timestamps. Also, additional pong cache state can be used to optimize search. For instance, pong caches could keep track of the number of search results received from a particular node. As a result, pong cache entries can be expensive in the way of memory. On the other hand, entires in the list L need not be very expensive at all. L can be implemented quite efficiently as a bit vector, or can even be approximated using a hash function or a Bloom filter. As such, L only requires a single-bit (or less) of state per node id.

The following two "heavy-state" IDSAs take advantage of L:

Conservative. If N_i receives N_k in response to a query, N_i will replace one of its pong cache entries in $P(N_i)$ with N_k as per its CRP if and only if $N_k \notin L(N_i)$. In addition, $L'(N_i) = L(N_i) \cup \{N_k\}$.

Aggressive. Just as with conservative ID balancing, if N_i receives N_k in response to a query, N_i will replace one of its pong cache entries $P(N_i)$ with N_k as per its CRP if and only if $N_k \notin L(N_i)$. However, if $N_k \in L(N_i)$ and $N_k \in P(N_i)$, then $P'(N_i) = P(N_i) - \{N_k\}$. As before, $L'(N_i) = L(N_i) \cup \{N_k\}$.

In Section 5.4, we study the effectiveness of these IDSAs.

5.3.6 Dynamic Network Partitioning (DNP)

In the results we will see in the next section, we will find that "light-state" IDSA is able to significantly mitigate poisoning so long as the number of malicious nodes does not exceed cache size. In this subsection, we propose a *dynamic network partitioning*

Table 5.1: GUESS Protocol Policies

Policy	*Options*
Seeding Policy (SP)	Random-Friend (RF), Popular-Node (PN), Trusted Directory (TD)
Introduction Protocol (IP)	Enabled ($0 < p \leq 1$) or Disabled ($p = 0$)
Ping Probe Policy (PPP)	Random
Pong Choice Policy (PCP)	Random, MRU, LRU, FIFO, LIFO
Cache Replacement Policy (CRP)	Random, MRU, LRU, FIFO, LIFO
ID Smearing Algorithm (IDSA)	Light-State, Conservative, Aggressive

(DNP) technique whose goal is to keep the number of malicious nodes that nodes interact with, on average, below their cache size.

The key idea that we employ in DNP is that nodes do not necessarily need to search the entire network at any particular instant, and if we can limit the subset of nodes that are searched at any given time, then we can also limit the number of malicious nodes that can impact the search. As search proceeds, the subset of the network that is searched (the "active" subset) can change, but we would like to keep the number of malicious nodes, on average, in the active subset of the network below a node's pong cache size.

At any given time, a node divides the entire network into a number of partitions and selects one of those partitions to be the active subset. A node only accepts node ids into its pong cache from the active subset. By only accepting node ids from the active subset, malicious nodes that would like to have their node ids in a victim's pong cache will be required to seize node ids in many different partitions.

The method that we use to partition the network works as follows. We assume that IP addresses are j-length bit strings. Each partition is of size 2^p, where a node has the freedom to choose p such that $0 \leq p \leq j$. There are 2^{j-p} partitions. (The case in which $p = 0$ corresponds to a scenario in which the node relies on a single central server. If $p = j$ then DNP is not used as the size of the partition is the size of the entire network.)

Each node chooses a constant, ϕ ($0 < \phi < 2^{j-p}$), and only accepts an IP address a into its pong cache if a falls into partition ϕ. In addition, each node chooses a random

key, κ. The nodes that make up partition ϕ are those nodes that have ids such that the value of the $j - p$ least significant bits of $h(a||\kappa) = \phi$.

Our DNP scheme allows good nodes to dynamically change how they partition the network much more easily than the amount of effort required by malicious nodes to acquire node ids in different partitions. The parameter κ allows each node to partition the network in a way that is dynamic and unpredictable.

By taking this approach, each good node can use a different key κ such that malicious parties are not able to predict which partitions they need to acquire node ids in to poison the caches of particular good nodes, and good nodes can periodically change the key that they use to thwart any incoming attack mounted by malicious nodes that have already acquired a particular set of node ids.

We assume that acquiring node ids involves some reasonable (but not necessarily prohibitive) cost for malicious nodes. For instance, in order to receive pings and send pongs with malicious ids, a malicious node needs to be able to receive traffic at an IP address that it either owns or has acquired by compromising an Internet host. Malicious nodes need to expend some effort to acquire IP addresses, and, once acquired, they will not be able to easily spoof a different set of IP addresses immediately.

With high probability, it is the case that if m random malicious node ids are in the network, then on average, at most $\frac{m}{2^{j-p}}$ will appear in a node's pong cache. If DNP is not used, then as long as $c > m$, IDSAs significantly mitigate poisoning, whereas when this DNP is used, then as long as $c > \frac{m}{2^{j-p}}$, IDSAs are able to significantly mitigate poisoning. The use of DNP allows our clients to be able to tolerate more malicious nodes in the network.

5.3.7 Malicious Node Detection (MND)

The policies that we have described thus far attempt to use various cache management strategies to mitigate poisoning. In addition to studying how cache management can help us achieve our goals, we also study malicious node detectors (MNDs). Our goal in this paper is to understand how well our cache management mechanisms perform

compared to MNDs, and to understand under what conditions we will be required to use MNDs in addition to cache management techniques.

A MND takes a node id as input, and attempts nto determine if the corresponding node is malicious. A MND tries to make this determination based on past experience interacting with the node, or using information from other more "trusted" nodes. Various algorithms based on reputation systems and other mechanisms have been proposed (i.e., [91]), and can serve as MNDs.

We model the accuracy of a MND using a single parameter, p, the probability that the MND will tell us that a node is malicious given a malicious node id. That is, a MND is a black-box that takes a node id as input, and outputs "yes" or "no" if it believes the corresponding node is or is not malicious, respectively. We do not consider the actual implementation of MNDs in this paper, but if the output of the MND is correct p percent of the time, we say that the MND has an "accuracy" of p percent.

We assume that if a MND is used, it is used as follows: when a node N_i receives a pong from N_j, it submits all of the ids received to the MND before considering them for insertion into its pong cache. In addition, if N_i uses an IP and receives a query from N_k, then N_k's node id is submitted to the MND before its introduction into N_i's pong cache is considered.

5.4 Simulation and Results

In this section, we present the results of various evaluations that we conducted using a discrete-event simulation of the model described above to address the research goals presented in Section 5.2.

To facilitate simulation, we extended the various sets we described in the basic model of Section 5.1 to be functions of time. The set of nodes N, is a function of time, $N(t)$, in our simulation. The pong caches $P(N_i), \forall i$ are also functions of time, $P(N_i, t), \forall i$, *etc.* Time advances in discrete-intervals $t = 0, 1, 2, etc.$ Our discrete-event model can approximate the continuous behavior of a real system as the physical time between intervals decreases.

Table 5.2: Baseline Simulation Parameters

Parameter	*Value*
Number of Nodes (at any one time) (n)	100
Pong Cache Size (c)	5
Births / Deaths Per Round (b)	1
Number of Node Ids in Pongs (r)	1
Number of Experiments	100
Ping Probe Policy (PPP)	Random
Pong Choice Policy (PCP)	Random

We say that all of the events that occur in one time step occur in a "round." If node N_i pings N_j, receives N_k in a pong, and updates its pong cache at time t to additionally contain node N_k, then we say that $P(N_i, t+1) = P(N_i, t) \cup \{N_k\}$. The ping, pong, and cache updates for all of the nodes in the system at a given time t happen sequentially within the same "round."

5.4.1 Simulation Setup

Our goal in this work is to build a fundamental understanding of the issues and trade-offs involved in using the various policies we outlined in Section 5.3 to mitigate pong-cache poisoning. Our evaluations are not designed to predict the performance of an actual system, but to gain an understanding of the trends and trade-offs involved in using the different policies. In the evaluations described below, we simulated a GUESS network using the baseline parameters in Table 5.2.

We estimate that nodes in a GUESS network will have enough memory to store a substantial number of node ids in their pong caches, even for large networks. For instance, if each pong cache entry requires 10-bytes of storage (4 bytes for an IP address, and 6 bytes of additional storage for timestamps and other data), then storing pong cache entries for, say five percent, of the nodes in a 2,000,000 node GUESS system, only 1 megabyte of main memory is required. Such a memory requirement is even becoming acceptable for traditionally memory-constrained wireless devices such as cell phones and PDAs. In our simulations, we conservatively assume that pong

Table 5.3: GUESS Simulation Parameters (V=Variable, PN=Popular Node, R=Random, D=Disabled, A=Aggressive)

Parameter/Simulation	A1	A2	A3	A4	B	C	D	E1	E2
SP	V	V	PN	V	PN	PN	PN	PN	PN
IP (p)	0	1	V	1	1	1	1	1	1
PCP	R	R	R	R	V	R	R	R	R
CRP	R	R	R	R	R	V	R	R	R
IDSA	D	D	D	D	D	D	V	D	A
Num Rounds (thousands)	1	0.5	1	2.5	10	75	1	5	5
Mal Nodes (m)	0	0	0	0	5	5	V	V	V

caches are able to store node ids of five percent of the nodes in the network. Larger pong caches will result in higher numbers of live ids and a smaller ratio of poisoned ids to live ids than those that we present in the simulations in this section.

In our simulations, there are $n = 100$ nodes in the network at any instant, and they have pong caches that can store $c = 5$ node ids. The same trends that we see in our small 100-node simulations can be seen in larger GUESS networks, and we clearly expect real GUESS networks to have much larger numbers of nodes. We confirmed that trends that we see scale to larger networks in Section 5.4.7 where we describe the results of a simulation that varies the total number of nodes and shows that we can expect our observations to be valid for larger systems.

The settings for all of the policies used in each simulation that we present later in this section are shown in Table 5.3. Each column of Table 5.3 corresponds to one of our simulations, and we refer to them by the column names in the results below.

We assume that each of the good nodes issues a ping in every round. While this need not be the case in a real network at all times, it allows us to model the effect of pong-cache poisoning when the network is at its "busiest." Also, when an IP is used, we assume malicious nodes issue one ping every round.

For accounting purposes, we assume that malicious nodes have pong caches that are of the same size as good nodes, but that their entries are always poisoned. Furthermore, all good nodes use the same SP, IP, PPP, PCP, CRP, and IDSA settings.

To produce the various graphs we present in this section, the corresponding simulations were run 100 times, and the results were averaged. The y-axis on the graphs typically measure the total number of live or poisoned pong cache entires in the system. Live cache entries are ones that contain the node ids of good nodes that are not dead. Malicious node ids are not considered live ids. Instead, a pong cache entry that has a malicious node id is considered poisoned. Therefore, the sum of the number of live, poisoned, and dead cache entries equals the total number of pong cache entries in the system.

After describing some simplifying assumptions that we made in our simulations, our first order of business is to analyze which protocol options achieve our first goal of maximizing the number of live node ids in pong caches, without the presence of malicious nodes. We then evaluate which policies minimize susceptibility to attacks.

Good Nodes

Each good node N_i issues a ping at each time step t, and chooses which node N_j to ping from its pong cache, $P(N_i, t)$. We assume that N_i receives one node id from N_j in response to its ping, and N_i replaces at most one of its cache entries with the node id in the pong if necessary as per its CRP. More formally, when node N_j receives a ping, it always responds with $r = |S| = 1$ node id marked live in its cache, if possible. Once N_i receives the pong, it chooses at most one cache entry as the set X to replace according to its CRP, and chooses $Y = S$.

Deaths

After all nodes issue their pings and receive pongs in a time step, one node is randomly chosen to die, and a new node is born is place of it. That is, $\delta = |\Delta(t)| = |B(t)| = 1$, $G(t+1) = (G(t) - \Delta(t)) \cup B(t)$, and $N(t+1) = G(t+1) \cup M(t+1), \forall t$. Furthermore, we assume that nodes that are born are newly born, and have never previously participated in the system. Nodes that die are not re-born later. In other words, $B(t) \cap (G(0) \cup G(1) \cup ... \cup G(t)) = \emptyset, \forall t$.

Malicious Nodes

In our simulations, we assume that malicious nodes are reasonably strong adversaries, with respect to their lifetimes, and the amount of state they can keep track of.

While good nodes may be susceptible to death, we assume that the set of malicious nodes is constant: $M(0) = M(1) = ... = M(t), \forall t$. The set of live good nodes $G(t)$, on the other hand, changes over time, but its cardinality is constant, $|G(0)| = |G(1)| = ... = |G(t)|$. Therefore, we assume that good nodes are born and die at a constant and equal rate. While this may not be true for a system in which the number of nodes is growing or shrinking, we roughly expect this to be true when the system achieves steady-state. While we do not vary the number of malicious nodes in the system in the midst of a simulation, we do study how different numbers of malicious nodes in the system impacts our ability to mitigate attacks. In evaluations in which we studied malicious nodes, we measured how the number of poisoned cache entries increases as many nodes become malicious.

We assume that malicious nodes have larger memories than good nodes, and can keep track of many more node ids. Firstly, malicious nodes keep track of the node ids of all other malicious nodes in the system. Secondly, we assume that malicious nodes have enough memory to keep track of the node ids of all of the good nodes in the system.

If an IP is not used in a GUESS system, then malicious nodes have no way of "actively" working to insert their node ids into the caches of good nodes. Instead, they must wait until good nodes ping them, at which point they can send a pong with a malicious node id. However, if an introduction protocol is used, then malicious nodes can actively work to poison caches by pinging good nodes. As a result, in our simulations, we have malicious nodes issue pings when an IP is used. We assume that malicious nodes are "omniscient" in that they know the node ids of all of the live, good nodes in the system. Malicious nodes issue pings only to live, good nodes when an IP is used, and they do not waste their resources pinging other malicious nodes or dead nodes.

Malicious nodes have the incentive to issue pings at an increased frequency when an introduction protocol is used because the more pings they issue, the more good

nodes are likely to insert malicious ids into their caches. However, we assume that malicious nodes have the same processing capacity as good nodes, and are able to issue one ping per time step just as the other nodes may do so when the system is maximally loaded. To model a scenario in which malicious nodes have more processing power than good nodes, we can simply increase the number of malicious nodes appropriately.

Now that we have described our simplifying assumptions and our simulation scenario, we describe what we learned from the results of our simulations.

5.4.2 Seeding and Introductions

R1. A seeding policy must be used in tandem with an introduction protocol if it is to be successful at all in achieving liveness. When seeding policies are used in combination with introductions, a high percentage liveness can be achieved (e.g., a liveness of 95 percent was achieved using our baseline simulation parameters).

Figure 5.2 shows the results of simulation A1 in which we measured the number of live entries for the three SPs described in Section 5.3, with no malicious nodes in the system. Simulation A1 was run with no IP, a random CRP, and the IDSA disabled.

At time $t = 0$, seeding from the trusted directory results in 500 live node ids since newly born nodes are given $c = 5$ guaranteed-to-be-live entries from the directory. As time advances, the number of live node ids in the TD case drops because some nodes die, and the ids pointing to them become invalid.

However, when the RF or PN policies are used, we initialize all of the pong caches at $t = 0$ to contain just one live node id. The system starts out with a total of 100 live ids, and as pings and pongs are exchanged, the number of live ids starts to increase.

We first explain the system's steady-state behavior when *no queries* take place and a trusted directory is used for seeding. From Figure 5.2, we can see that there are 250 live entries in the steady-state. We can derive this number theoretically. Let x be the number of live entries in all of the caches in the system. Every round, some node dies and a new node is born in place of it. The node that died has $x/100$ live ids, on average. In addition, as a result of the death, any cache entries at other nodes that pointed to the dead node are no longer live. On average, $x/100$ cache entries die

at other nodes as a result since we assume that each node id is replicated an equal number of times. At the same time, the new node that is born is given 5 live ids by the trusted directory. The system achieves steady-state when the gains (5) equal the losses ($2x/100$), or when $x = 250$.

We now consider the case in which TD SP is used and pings and pongs *do* take place. An interesting effect occurs when pings and pongs are exchanged under TD SP. A few of the node ids become highly replicated throughout pong caches in the system while many of the node ids appear only a few times (or die out completely). To see this, assume that the system is initially seeded randomly with live node ids, and consider the state of the system after some N_i sends a ping message to N_j, and receives N_k in a pong message. While there is a relatively small probability that N_k is already in N_i's cache, it is more probable that some other node id N_r ($r \neq k$) in N_i's cache will be replaced. The total number of times that N_k appears in the pong caches of the system increases by one, and the total number of times N_r appears decreases by one. Since N_k now appears more frequently, the probability that it will be chosen to appear in a pong message and be replicated once more is increased. On the other hand, since N_r appears less frequently, the probability that it will be replicated is decreased. In further rounds, it is more likely that N_k will become highly replicated and that only a few copies of N_r will exist (if any). After many rounds, very few node ids become highly replicated, many node ids appear in just a few pong caches, and many node ids die out.

Due to the existence of some highly replicated node ids, the average live node id appears in more than 2.5 caches. Figure 5.2 shows that when pings and pongs are exchanged, just under 200 node ids are live in the steady-state. Hence, if a real system were to be able to use a TD SP, it would not be worthwhile to exchange ping and pong messages!

Finally, we explain what happens when RF and PN SPs are used. In the case of the RF and PN policies, the number of live ids in the system starts increasing as nodes exchange pings and pongs. Unfortunately, after some number of rounds, the RF and PN policies suffer from the same problem. In both policies, when new nodes are born, they seed their caches by copying the caches of some other node. Over time,

as nodes die, all of the node ids contained in the pong caches of the initial set of nodes in the system die. Since newly born nodes are simply copying the cache entries that originated from the initial set of the nodes in the system, all of these caches entries are bound to die as well. In other words, in the current scenario, newly born nodes only learn about nodes that existed before them, and never learn about nodes that are born after they are.

Nodes must use an IP if the "current generation" of nodes is to learn about "future generations" of nodes. Figure 5.3 shows the outcome of simulation A2 in which we employed an IP. The figure shows that any of the SPs we considered can be used to achieve over 475 live node ids out of a possible 500 in steady-state. In other words, we achieved a liveness of 95 percent when SPs were used in combination with an IP.

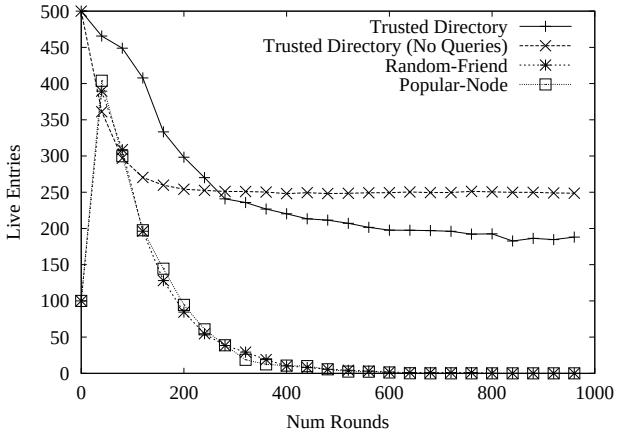

Figure 5.2: A1: Live Entries vs. Number of Rounds for Various Seeding Policies (No Introductions)

R2. Using small introduction probabilities do not mitigate poisoning in the steady-state.

Figure 5.4 shows an the results of a simulation in which we (for various numbers of malicious nodes) measured the number of poisoned steady-state entries when all of the good nodes used introduction probabilities between 0 and 1. When introductions are used in the network, malicious nodes have the incentive to issue queries, and can poison the caches of other nodes by pinging them. As such, we wanted to determine if

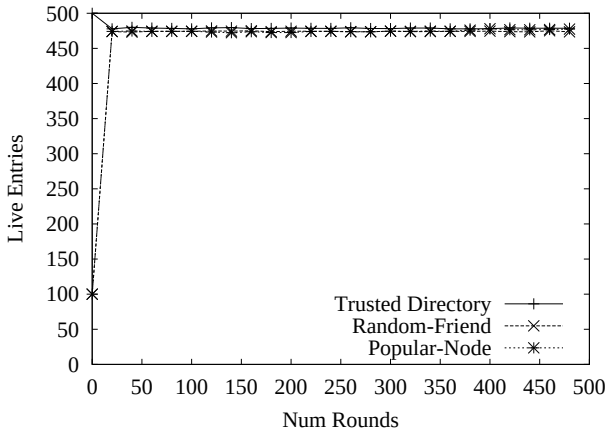

Figure 5.3: A2: Live Entries vs. Number of Rounds for Various Seeding Policies (With Introductions)

using introduction probabilities smaller than 1 might help decrease poisoning. However, due to the many pings and pongs that are exchanged even in a few number of rounds, we find that using even small introduction probabilities do not mitigate poisoning. For example, when there are 2 or 5 malicious nodes in the network, we find that approximately 20 and 50 cache entries are poisoned, respectively, as long as a non-zero introduction probability is used. When there are 10 malicious nodes in the network, we can see that using an introduction probability just above 0 results in approximately 100 poisoned steady-state cache entries. As the introduction probability is increased to 1, up to approximately 150 entries are poisoned in the steady-state. As the number of malicious nodes increases, higher introduction probabilities do result in more poisoning, but the key lesson that we learn here is that small introduction probabilities cannot be used to mitigate steady-state poisoning.

R3. Poisoning occurs more slowly with the popular-node (PN) than with the random-friend (RF) seeding policy (SP).

One might intuitively expect that the PN SP is more susceptible to poisoning because if the popular node's cache becomes poisoned, then all newly born nodes' caches become poisoned since their caches are seeded from the popular node. However, while PN SP is more susceptible than RF in this regard, we find that the rate at which

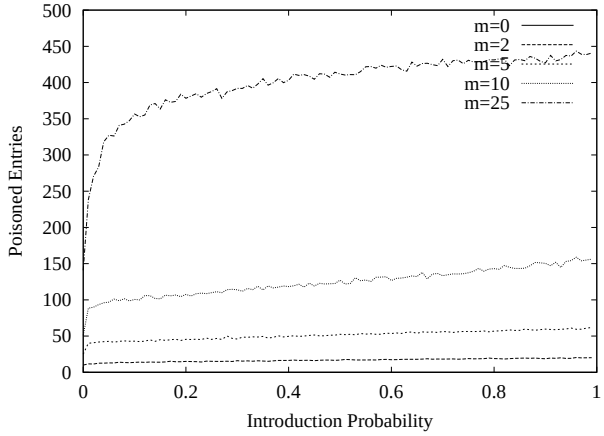

Figure 5.4: A3: Poisoned Ids vs. Introduction Probability for Various Numbers of Malicious Nodes

poisoning occurs with PN SP is, on average, slower because the popular node must be poisoned before other nodes can be poisoned.

Figure 5.5 shows the outcome of simulation A4 in which the number of poisoned cache entries is measured over time in a system with $m = c = 5$. From Figure 5.5, we can indeed see that once the popular node's pong cache is completely poisoned (after about 1250 rounds), all of the entries in the system become poisoned. However, the other interesting effect that we notice is that before the point at which the popular node's pong cache becomes completely poisoned, the rate at which entries in the system become poisoned is slower with the PN SP than with the RF SP.

Poisoning occurs more slowly with PN SP because the popular node's cache must become poisoned in order for all new node's caches to become poisoned. The set of malicious node ids that may appear in the caches of newly born nodes is restricted to the set of malicious node ids that appear in the cache of the popular node. When RF SP is used, each newly born node may have a different friend from which their cache is seeded, and each of those friends' caches may be poisoned with different malicious node ids. The pool of malicious node ids which may start poisoning the system is not constrained in any way with the RF SP. As nodes start exchanging ping and pong messages, the full set of malicious node ids are distributed to new nodes, and poisoning occurs more quickly with RF than with PN SP.

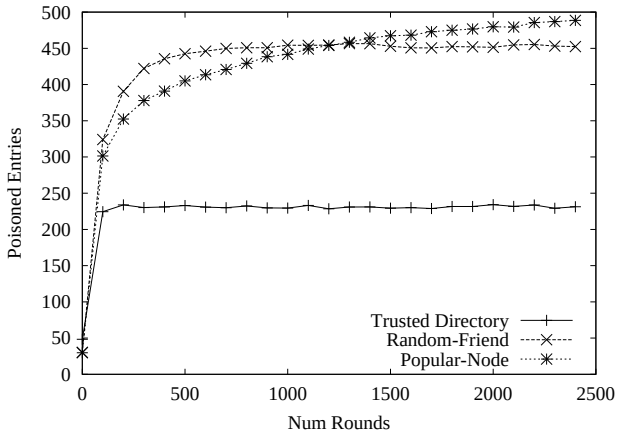

Figure 5.5: A4: Poisoned Entries vs. Number of Rounds for Various Seeding Policies (With Introductions)

We believe that using PN SP is a better choice than RF SP. One might consider using RF SP instead of PN SP because in the steady-state, there may still be some live entries in the system with RF SP. However, there are so few that little useful work will get done in the system. For this reason, and because PN SP results in slower poisoning than RF SP, we believe that using PN SP is a better choice than RF SP.

We also note that in steady-state, almost *all* of the cache entries in the system are poisoned (or dead). In Section 5.4.5 we explain why this happens, and we then go on to investigate how to control cache poisoning.

5.4.3 Pong Choice Policy (PCP)

R4. PCP does not affect poisoning in the steady-state.

In Figure 5.6 we show the results of a simulation that varies PCP in a system with five malicious nodes. The figure illustrates that pong choice does not significantly affect the number of live node ids in the system. This is true because pong choice does not impact which cache entries get replaced.

Malicious nodes respond to pings with other malicious node ids that are always live (since we assume that malicious nodes never die). Good nodes respond to pings

with node ids that are highly likely to be live, or unknowingly respond to pings with malicious node ids. The pong choice policy has no way of distinguishing good live nodes from malicious nodes, and hence does not help us mitigate the poisoning of pong caches.

As the number of live ids in the system is more or less indifferent to the PCP used, we use Random PCP as the default PCP in evaluations described in the following sections due to its relative ease of implementation and runtime efficiency.

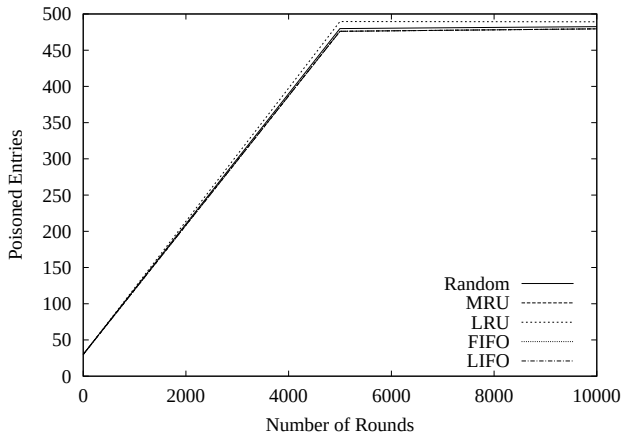

Figure 5.6: B: Poisoned Ids vs. Number of Rounds for Various PCPs

5.4.4 Cache Replacement Policy (CRP)

R5. While all the CRPs that we study result in inevitable cache poisoning, MRU CRP slows the rate of poisoning the most.

Figure 5.7 is the result of simulation C in which we held all simulation parameters constant and varied the CRP. The simulation measures the increase in the number of poisoned node ids over an extended number of rounds in a system with $m = 5$ malicious nodes for various CRPs. The figure shows that almost all of the cache entries become poisoned at 60,000 rounds or earlier. The posioning of almost all of the entries is "inevitable." In Section 5.4.5, we present an informal analysis explaining why this is the case.

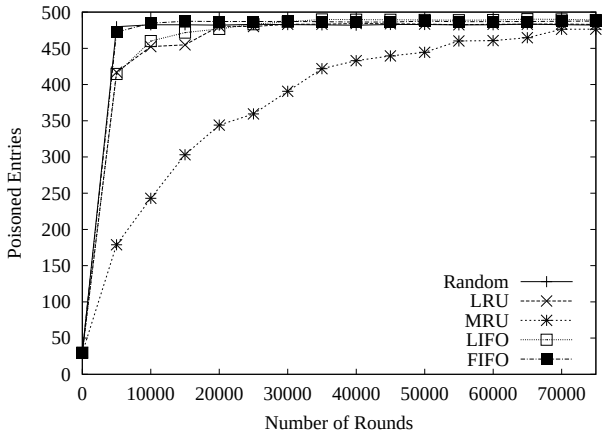

Figure 5.7: C: Poisoned Ids vs. Number of Rounds for Various CRPs

However, we can also see that using non-random CRPs can *slow* the rate of poisoning. When LRU CRP is used, cache entries are discarded (replaced) in the order in which they are pinged. When a particular node is pinged, its id is "pinned" in the cache, and becomes a candidate for being pinged again for up to $c-1$ rounds. Poisoning occurs slightly slower with LRU CRP than random CRP because a live, good node that is pinged cannot not be "randomly" replaced. A malicious node id cannot take the place of a pinged, good node id for $c-1$ rounds. The good node may even be pinged again, and so long as it is live, can continue to exist in the cache and prevent a malicious node id from taking its place.

MRU CRP slows poisoning more effectively than LRU or random CRP. When a malicious node is pinged under a MRU CRP, its id is repaced by the malicious id from the pong. The number of malicious node ids does not increase. Furthermore, when the next introduction is received, the malicious node id is replaced (since it is the most recently used). If the node introducing itself is not malicious, then the number of poisoned ids decreases. Otherwise, the number of poisoned ids in the cache stays the same. The only case in which the number of malicious nodes in a good node's cache increases is when the good node pings another good node that died since it was either inserted in the cache or pinged last, and it immediately receives an introduction from a malicious node. In this case, the malicious node id takes the place of the good

node that is marked dead. Hence, while inevitable cache poisoning does occur with MRU CRP, it occurs slower than with LRU or random CRP.

The "catch" with MRU CRP is that it leads to bad search performance [15] since many cache entries are not updated and die before their next use. Furthermore, live node ids that are introduced into the pong cache via an IP are subsequently replaced with ids that are more likely to be stale from pong messages. To deal with this problem, we suggest dividing the pong cache into two caches– a small pong cache and a large introduction cache. Node ids received in pongs are placed in the pong cache, and the pong cache is managed using a MRU CRP to slow poisoning. Node ids received from introductions are placed in the introduction cache. Node ids in the introduction cache are highly likely to be live since these nodes recently issued pings, and the introduction cache is managed using a FIFO (first-in-first-out) CRP. If many of the entries in the pong cache are dead, ids from the introduction cache can be used to issue queries and provide reasonable search performance. (This dual-cache scheme is similar to that described in [15]. The pong cache is equivalent to the link cache, and the introduction cache is analogous to the query cache, except that the source of the ids are introductions and not pongs.)

Now that we have an understanding of how to slow poisoning, in the next section we explain why cache poisoning inevitably occurs. In the section following the next, we show that by taking advantage of an IDSA, we can not only slow the rate of poisoning, but we can limit it in the steady-state as well.

5.4.5 Inevitable Cache Poisoning

We first explain why random CRP results in inevitable cache poisoning. To simplify our explanation let us assume, for the moment, that we are using the TD SP, and that when new nodes are born their pong caches are seeded with node ids uniformly at random from the set of live nodes in the system.

We recall from Figure 5.5 that poisoning occurs more slowly with TD SP than with PN or RF SP. Figure 5.5 shows the result of a simulation in which we measured how many cache entries were poisoned as time elapsed in a system with $m = 5$ malicious

nodes. If we can demonstrate that cache poisoning is inevitable using a TD SP, we can expect that poisoning will only occur faster (and inevitably) with the PN or RF SPs based on our observations from Figure 5.5.

Assume there exist n nodes, m of which are malicious in a GUESS system. The pong caches of all of the good nodes in the system have been initialized (or seeded) uniformly at random from the population of n node ids. On the average, $x = m/n$ percent of the entries in the caches of good nodes are poisoned. Each node uses a random PCP, and a random CRP. In addition, each node N_j that receives a ping from N_i responds with one node id in its pong, and N_i always chooses to replace some entry in its pong cache with the node id in the pong received.

Let us consider what happens in the first round of the protocol. The first node to issue a ping may end up (a) increasing the number of malicious node ids in its pong cache by replacing one of its good pong cache entries with a malicious one, (b) decreasing the number of malicious node ids in its pong cache by replacing one of its malicious pong cache entries with a good one, or (c) the number of malicious node ids in its pong cache will stay constant by having a good node id replace another good node id or a malicious node id replace another malicious node id.

Figure 5.8 shows a tree that enumerates the possible outcomes. The nodes of the tree indicate intermediate states that a GUESS node might be in as it issues a ping, receives a pong, and processes the pong. The edges of the tree are state transitions, and they are labeled with the probability that a particular state transition occurs. The left edge leaving a node indicates the case in which a good node id is chosen, and the right edge leaving a node indicates the case in which a malicious node id is chosen. The leaves of the tree are labeled with the case that they contribute to; for example, if a node chooses to ping another good node (left branch from state $s0$), receives a malicious node id in a pong (right branch from state $s1$), and replaces a good node id (left branch from state $s4$), the corresponding leaf is labeled (a) as the count of malicious node ids increases by one.

From the figure, we can see that the probability that the number of malicious node ids will increase after the first node N_i issues its first ping is $P_1 = (1-x)^2x + (1-x)x$. P_1 is simply the sum of the probabilities that 1) N_i pings a good node N_j, N_j returns

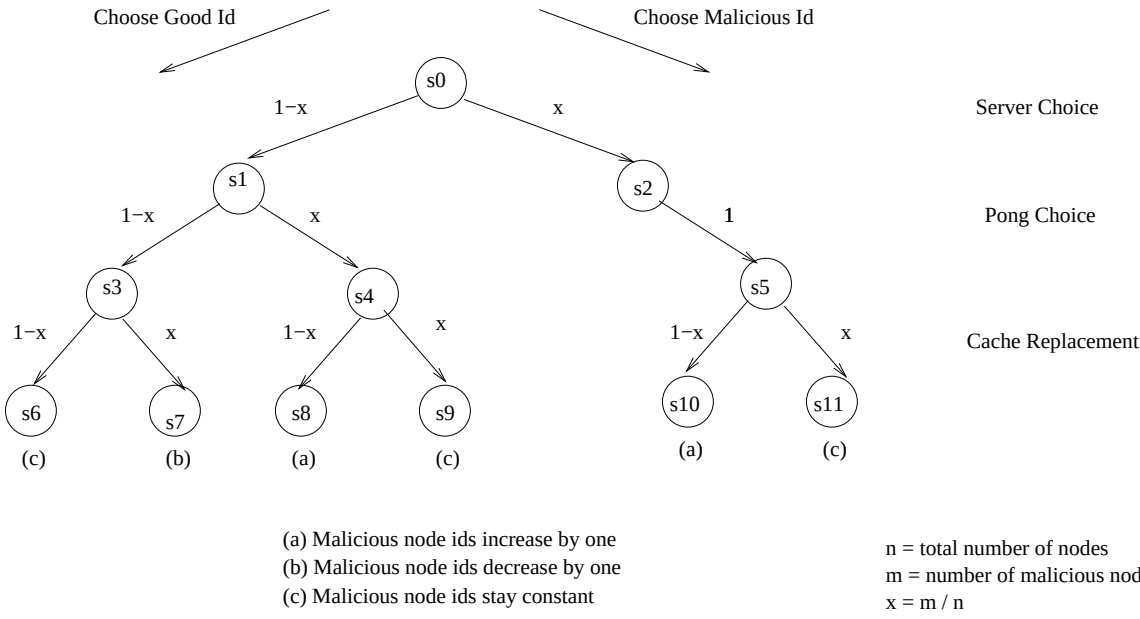

Figure 5.8: Probability of Propagated Poisoning

a malicious node id N_m, and N_m replaces a good node in N_i's cache, and 2) N_i pings a malicious node id N_m, and N_i replaces a good entry in its cache with the malicious node id from N_m's pong.

However, the probability that the number of malicious nodes in N_i's cache will decrease is only $P_2 = (1-x)^2 x$ since the only relevant corresponding path is the one through states $s0$, $s1$, $s3$, and $s7$.

Since $P_1 > P_2$, it is more likely that the first node that issues a ping will have its count of poisoned cache entries increased by one. In addition, if this first node has an additional entry poisoned, the problem is slightly magnified for the second node that issues its ping. Specifically, if the second node happens to choose the first node as the server to ping, then the probability that the first's nodes pong choice is malicious is slightly greater than x.

These probabilities explain why we can expect that when random CRP an a TD SP is used, the malicious nodes will have their node ids propagated into the caches of every good node in the system in the long-run. The situation will not be any better with PN or RF SP used together with introductions because the ids that newly born nodes are seeded with are more likely to be poisoned in the long-run than with the TD SP and no introductions.

When a PN or RF SP is used in combination with introductions, the introductions do not help the good nodes mitigate poisoned entries. If an IP is used in our example above, each node receives, on average, one introduction (ping) in the first round. The probability that a good node receives a ping (an introduction) from a malicious node is x. The probability that a good node replaces a good entry with the malicious node id from the pong is $1-x$. Hence, the probability that the number of malicious node ids in a good node's cache increases by one due to introductions is $x(1-x)$. However, the probability that the number of malicious node ids decreases by one is also $x(1-x)$ since another good node could introduce itself (with probability $1-x$), and a malicious id could be replaced (with probability x) with the good id. Hence, the IP does not help mitigate poisoning when PN or RF SP is used, and we can expect inevitable poisoning to occur when random CRP is used with a non-TD SP as well.

LRU and MRU CRP (like random CRP) also lead to inevitable cache poisoning. If LRU CRP is used, then when a good node N_g pings a malicious node N_{m1}, N_g receives a pong with a malicious node id N_{m2}. Node id N_{m2} is inserted into the N_g's cache as long as $m1 \neq m2$. (If $m1 = m2$, then a duplicate entry is not made in N_g's cache.) As a result, the number of poisoned entries in N_g's cache either stays the same or increases in each round when LRU CRP is used. In addition, if an introduction is received from a malicious node, yet another good cache entry could be replaced. As a result, LRU CRP will clearly lead to inevitable cache poisoning just as random CRP does. As can be seen from Figure 5.7, MRU CRP also leads to inevitable cache poisoning, although the rate of poisoning is slower than that of LRU CRP.

5.4.6 ID Smearing Algorithm (IDSA)

R6. The use of an IDSA limits the number of poisoned entries in the steady-state, and IDSAs mitigate poisoning as the number of malicious nodes increases.

Figure 5.9 shows the results of simulation D, and demonstrates that IDSAs can limit the number of poisoned cache entries *in the steady state*. For instance, when $m = 5$ the light-state IDSA results in 120 poisoned entries whereas over 400 entries are poisoned if no IDSA is enabled. IDSAs are successful in mitigating poisoning because the use of an IDSA causes malicious nodes to "limit their own success" in poisoning cache entries. If a good node N_g has two poisoned cache entries, N_{m1} and N_{m2}, then when one of the node ids in these cache entries is pinged, say N_{m1}, there is a non-negligible probability that N_{m1} will return N_{m2}'s id, causing N_g to remove N_{m2} from its cache as required by the IDSA. (In the case of conservative IDSA, it causes N_g to never re-insert N_{m2} into its pong cache again.) The more malicious entries that a good node's cache contains, the higher the probability that, when pinged, one of these malicious nodes will return an id of one of the other malicious nodes in the cache, and end up causing the good node to remove a malicious id from its cache (or never consider it for re-insertion). A similar effect that occurs (but is less likely) takes place when N_g's cache contains N_m, and N_g receives an introduction (a ping) from N_m itself. Use of the IDSA results in removing N_m from the good node's cache

(or not re-inserting it). For these reasons, the success of malicious nodes' poisoning efforts is self-regulated.

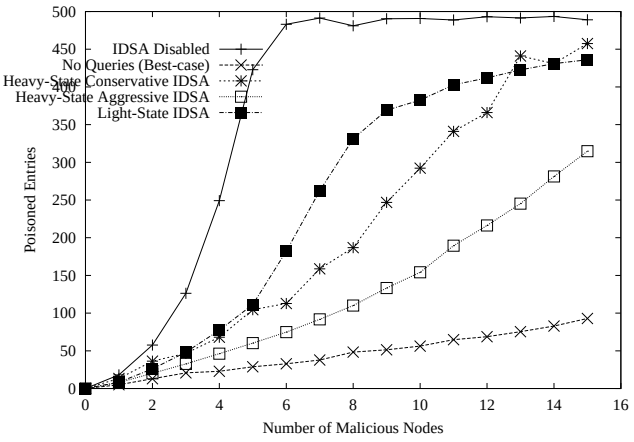

Figure 5.9: D: Poisoned Ids vs. Num Malicious Nodes with IDSA

We assume that a malicious node returns the id of another malicious node at random. Malicious nodes could try to work harder at poisoning by keeping track of exactly which good nodes have received which malicious node ids, such that malicious nodes can collude to ensure that the same malicious node id is never provided to the same good node twice. This could be done in an attempt to thwart the good node's use of the IDSA. However, good nodes could decide to drop ids out of their caches from time to time to easily thwart this attack, at the expense of application performance. (In Section 5.3.6, we describe a dynamic network partitioning scheme in which good nodes drop ids as such to thwart attacks.) Malicious parties would then need to have to obtain more malicious node ids to poison the good nodes because at any time they would be unsure as to whether or not sending an already used malicious node id would end up causing that malicious node id to be dropped out of the good node's cache or re-inserted.

R7. "Light-state" IDSA mitigates poisoning so long as the number of malicious nodes is less than pong cache size. Aggressive IDSA can mitigate poisoning even if the number of malicious nodes exceeds cache size, at the expense of memory and search performance. Conservative IDSA can be used as a middle ground to mitigate

poisoning while achieving better search performance than Aggressive IDSA.

If no IDSA is used at all, then as the number of malicious nodes approaches and slightly exceeds the cache size, Figure 5.9 shows that almost all of the cache entries in the system are poisoned. When $m = 4$, for instance, half of all of the cache entries are poisoned, and when $m = 6$, over 96 percent of the cache entries are poisoned.

Using a light-state IDSA significantly mitigates poisoning when the number of malicious nodes approaches cache size. At $m = 5$, only one quarter of the cache entries are poisoned, compared to over 80 percent poisoning if no IDSA is used. However, as m increases beyond c, light-state IDSA's performance worsens quickly. The number of poisoned entries when conservative IDSA is used does not worsen as quickly as that of the light-state IDSA, but it only eliminates about half the poisoned entries that aggressive IDSA is able to eliminate.

We can see from Figure 5.9 that even when the number of malicious nodes is significantly greater than the good nodes' cache size, the use of an aggressive IDSA limits poisoning to levels that might be tolerable. As the number of malicious nodes increases, the aggressive IDSA becomes less and less effective. Since node ids are eliminated from a cache when a duplicate id is received, the more unique ids the malicious nodes have, the fewer malicious entries the IDSA is able to eliminate.

5.4.7 Scaling-Up

In this section, we show that the observations that we report on small GUESS networks are indicative of the behavior of larger networks.

Figure 5.10 shows the results of simulations on networks that ranged in size from 100 to 1500 nodes. In each network, nodes have a cache size of five percent of the total nodes in the network. The "Total Entries" line plots the total number of cache entries that exist in the system. For instance, in a network of 1000 nodes, each node has a cache size of 1000*0.05 = 50 entries, and and a total of 50*1000 = 50,000 cache entries exist in the system.

In addition, five percent of the network is made up of malicious nodes, and each simulation is run until steady state; that is, each simulation is run until the number of

poisoned entries reached its maximum. The "Poisoned Entries: IDSA Disabled" line plots the total number of poisoned entries in the steady state when good nodes did not employ an IDSA, and the "Poisoned Entries: Aggressive IDSA" plots the same when an aggressive IDSA is used. From Figure 5.10, we can see that the IDSA plays a significant role in limiting poisoning even as the number of nodes in the network increases. However, from this figure, we cannot easily tell if the success of the IDSA will continue proportionally as the number of nodes in the network increases.

Since each of the networks we simulated had different cache sizes and different numbers of nodes, we need to normalize the y-axis of Figure 5.10 to compare the level of steady-state poisoning across networks of different sizes. To make this comparison, we look at the average fraction of a node's pong cache that has been poisoned in the steady-state. To determine this fraction, we divide the y-component of each of the points in Figure 5.10 by the product of the cache size and number of nodes to obtain Figure 5.11. (We leave out the line corresponding to "Total Entries" in the resulting figure.)

Figure 5.11 shows that the when IDSA is disabled, approximately 40 percent of the average node's pong cache is poisoned, regardless of the number of nodes in the network and the cache size of the nodes. On the other hand, when the aggressive IDSA is enable, only approximately 11 percent of the cache is poisoned, once again, irrespective of the number of nodes in the network and the cache size. From this figure we can conclude that enabling the IDSA limits the level of poisoning in a manner that is proportional to the number of nodes, and that the results of our 100-node simulations are likely to scale to networks of larger sizes.

5.4.8 Malicious Node Detection

IDSAs limit poisoning by attempting to distribute node ids equally across all of the pong caches. While we are able to eliminate a significant amount of poisoning with IDSAs, if we would like to further eliminate poisoning, we may need to employ a malicious node detector (MND). Result R8 tells us how well a MND used on its own (without an IDSA) mitigates poisoning, and we compare the performance of a MND

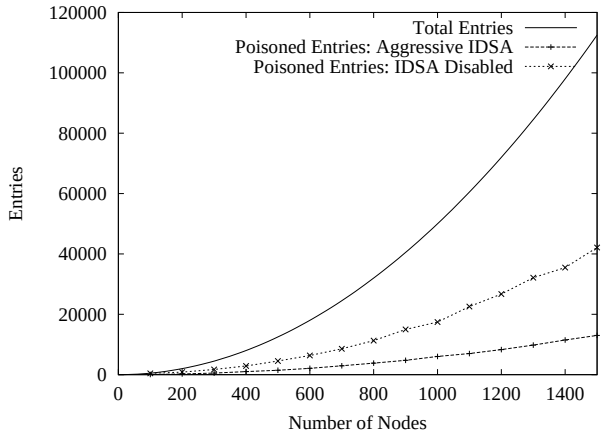

Figure 5.10: Poisoned Entries and Total Entries vs. Number of Nodes

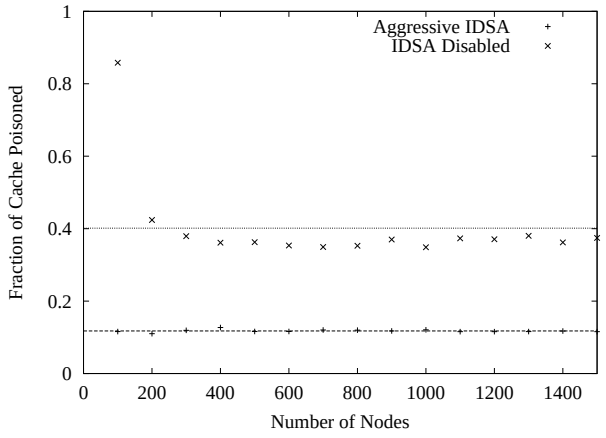

Figure 5.11: Normalized Poisoning vs. Number of Nodes

to the performance of the aggressive IDSA. Result R9 considers using a MND and the IDSA together.

R8. A MND needs to have an accuracy of at least 60 percent to achieve better results than the aggressive IDSA when $c = m$. Higher accuracies are needed for comparable performance with the IDSA as m increases.

Figure 5.12 shows how the number of poisoned entries in the steady state decreases as the accuracy of a MND increases. If the number of malicious nodes in the system is equal to the cache size ($m = 5$), we can see that a MND with an accuracy of 60 percent results in about 60 poisoned entries in the steady state. Figure 5.9 discussed in the previous section shows that enabling the aggressive IDSA (without using a MND) also results in about 60 poisoned entries (when $c = m = 5$). As the number of malicious nodes increases, the MND needs to become more and more accurate to achieve a level of poisoning comparable to that which can be achieved with the aggressive IDSA. The trade-off that needs to be considered is that the aggressive IDSA is (relatively) simple to implement, while MNDs of increasing accuracy may be harder to implement and may consume increasingly more system resources.

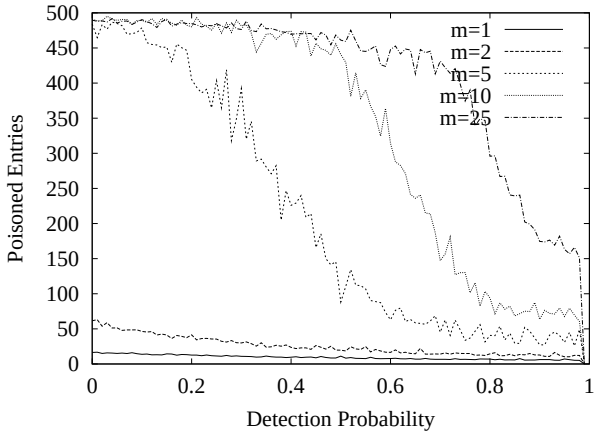

Figure 5.12: E1: Malicious Node Detection with IDSA Disabled

R9. A MND may be required in systems in which the number of malicious nodes is greater than the pong cache size. Else, the aggressive IDSA will be sufficient to mitigate poisoning.

Figure 5.13 shows the level of steady state poisoning that can be achieved when both the aggressive IDSA and a MND are employed. If the number of expected malicious nodes in the system is on the order of the cache size ($m = 5$), we can see that employing a MND results in a marginal benefit even if the detector is over 90 (but not 100) percent accurate. On the other hand, as the number of malicious nodes increases significantly beyond the cache size, detectors of increasing accuracies can significantly reduce poisoning.

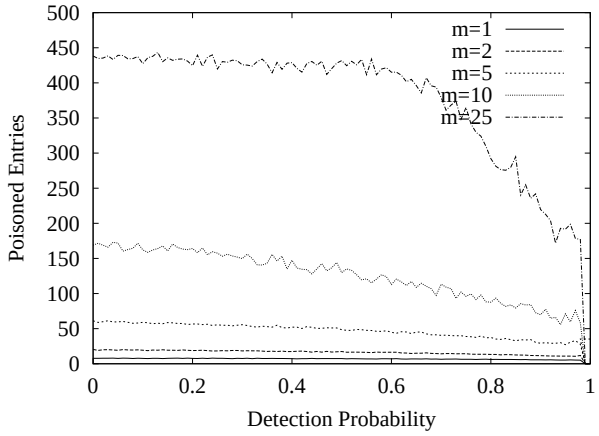

Figure 5.13: E2: Malicious Node Detection with Aggressive IDSA

5.5 Discussion

In the results we have obtained so far, we can see that with MRU CRP, IDSAs, and DNP, the effects of poisoning could still be significant and that GUESS, as a protocol, is highly vulnerable to pong-cache poisoning.

If no additional mechanisms are in place in the network to detect malicious nodes and remove them from the network, then these policies limit damage to the network due to undetected malicious nodes. While the simulations we conducted here assumed that the pong cache size of good nodes can accommodate five percent of the node ids in the network, we recommend that nodes use as large caches as possible to mitigate pong-cache poisoning. To further limit poisoning due to malicious nodes, it

is imperative that additional mechanisms such as MNDs that detect malicious nodes and eliminate them from the network be used.

It may be possible for MNDs to function without using any additional application-level mechanisms, but application-level mechanisms may greatly assist detection. For instance, in our model, we assumed that malicious nodes always returned other malicious node ids in pongs. In a sense, all the entries in the caches of malicious nodes "point" to each other, and form a "clique." As an example of a malicious node detection method that does not use any additional application-level knowledge, good nodes could attempt to detect such "cliques." That is, when good nodes receive node ids in pongs, they could attempt to piece together information about which nodes are pointing to which other nodes to detect such cliques in the network. Of course, such a mechanism might be hard to design because malicious nodes might not *always* return other malicious node ids in a real system, and a good node might have to exchange many pings and pongs to piece together this type of information.

Alternatively, nodes may take application-level information into account to decide which nodes are good and which nodes are malicious. If the protocol is to be used to support file-sharing, for instance, we could use the number of authentic search results obtained to help make an assessment of whether a node is or is not malicious. Of course, a file authenticity or reputation (i.e. [56]) system might need to be employed to come to a quality determination as to whether or not a file is authentic.

In our ongoing future work, we are exploring various application-level mechanisms that could be used to construct MNDs, extending our model to take these mechanisms into account, and evaluating their impact on limiting pong-cache poisoning using additional metrics.

5.6 Related Work

As discussed in Chapter 1, much research on security in P2P systems has worked to mitigate attacks against four distinct, but related, system properties: availability, authenticity, anonymity, and access control. As explained in Secton 5.2, our work here focuses on addressing attacks against the first two of these system properties,

availability and authenticity, in a GUESS P2P network.

While GUESS is just beginning to be studied by the academic research community [199], some research does exist on security in traditional Gnutella systems. In particular, [39] and [41] study availability and authenticity issues in traditional Gnutella. Work has also been done that studies how to use a P2P network to prevent DoS attacks on the Internet [94]. Papers such as [30] and [166] study how to use P2P networks to provide anonymity to users. Current P2P networks are plagued by digital rights management and access control issues, and [16] outlines some of the problems in this area. While the papers that we have mentioned so far focus mainly on unstructured P2P networks, [188] and [26] outline how security issues in DHTs might begin to be addressed.

We refer the reader to [139] for more complete coverage of related work in the area of secure P2P systems.

5.7 Chapter Summary

In this chapter, we defined a simple model of a GUESS network. We outlined the key decisions that nodes need to make to discover new nodes and manage their pong caches to mitigate pong-cache poisoning.

We ran simulations based on our model, and evaluated different options for the key decisions. Based on the results of our simulations, we found that:

- Introductions are essential to achieving liveness. We propose adding introductions as a basic mechanism in GUESS.
- We suggest using MRU CRP to slow down the rate of poisoning. However, we observe that MRU CRP does not limit poisoning in the steady-state.
- An IDSA can be used to limit poisoning in the steady-state. We recommend adding support for an IDSA in GUESS to mitigate pong-cache poisoning.
- If the number of malicious nodes is less than pong cache size, the IDSA is sufficient to limit poisoning, else a MND needs to be used.

- A DNP scheme can be used to reduce the number of malicious node ids that can poison a pong cache at any one time.

This chapter closes our study of application-layer DoS in P2P networks. In Chapters 2 and 3, we studied flooding-based DoS in the unstructured Gnutella P2P network. In Chapter 4, we studied DoS in the Chord DHT. Having studied DoS in the non-forwarding GUESS protocol in this chapter, we have completed our study of DoS in each of the major types of P2P networks.

In the next two chapters, now that we have developed some techniques to contain attacks in P2P networks, we will explore a commerce achitecture that can be used in a P2P network (Chapter 6), and an implementation of that architecture (Chapter 7).

Chapter 6

Digital Wallet Architecture

In the previous chapters, we studied application-layer DoS in P2P networks. Now that we have developed and tested various policies that can be used to contain such attacks, we turn our attention to how we might support e-commerce in P2P systems. We propose that nodes in P2P systems use "digital wallets" to conduct e-commerce transactions with other nodes. The nodes in such a P2P system could be owned and run by users, vendors, and/or banks. In this chapter, we specifically factor out the similarity between the interfaces used by user, vendor, and bank nodes such that the three types of nodes can be architected to interact in a P2P fashion.

A digital wallet is a software application that guides a user through a commerce transaction by helping him or her choose a *payment instrument* (such as a credit or debit card) that is acceptable to both the user and the vendor, and carries out the execution of the payment transaction. We define the term *payment instrument*, as well as several other terms that we will use in this chapter in Section 6.1. A number of wallet designs have recently been proposed, but we will argue they are typically targeted for particular payment instruments and operating environments. In this chapter, we describe a wallet architecture that generalizes the functionality of existing wallets, and provides simple and crisp interfaces for each of its components.

Before proceeding to describe the main features of our wallet architecture, we first introduce some terminology that we will use in this chapter and the next.

6.1 Terminology

In this section, we briefly define some terminology necessary for understanding our wallet architecture. The wallet architecture is described in the next section.

6.1.1 Instrument Instance and Instrument Class

An instrument instance (or, instrument, for simplicity) is a collection of state information representing economic value that a protocol can operate on as part of an electronic commerce transaction. For example, "Gary's Citibank Mastercard" is an instrument whose state is made up of his full name, credit card number, and expiration date. The instrument may also store other information such as his billing address.

It is important to note that an instrument, in the context of this paper, is a digital proxy for an instrument in the "physical" world. "Gary's Citibank Mastercard" is, in reality, an agreement or contract between Gary and Citibank which may be made up of a signed credit card application in addition to other documents such as a contract stating Gary's credit limit and the terms of the agreement. The digital representation of this instrument, however, only contains state parameters that are relevant for conducting commerce transactions with that instrument, and the instrument's digital representation need not contain the actual contract. In the case of "Gary's Citibank Mastercard", for example, the digital instrument may only contain those state parameters necessary for the SET protocol [183] to execute an online payment operation.

Each instrument belongs to an instrument class. An instrument class defines the structure of the state information necessary to store an instance of a given instrument, as well as behavior that is common to all instruments of that class. Examples of instrument classes are CyberCoin [37], ecash [54], or Mastercard. "Gary's Citibank Mastercard" is an instance of the Mastercard instrument class.

6.1.2 Protocol

A protocol defines a sequence of steps that accomplish a particular operation using a specified instrument. In each step, the protocol may send information to a peer, or process information locally. For example, in one step the protocol may create a certificate containing the user's account number and payment amount. In the next step, it may send the certificate to the peer.

The work of a protocol is to execute a correct sequence of steps to accomplish a requested operation; the sequence is not necessarily static and pre-determined, but may vary dynamically depending upon requests and responses sent between the peers executing the protocol. In general, a payment protocol is one that supports a PAY operation and whose sequence of steps results in a transfer of economic value between two or more peers. SET is a payment protocol that may be used to transfer monetary value from a bank to a vendor's account, while concurrently (and atomically) debiting the user's credit card account, under the condition that the resulting balance does not exceed the user's credit limit.

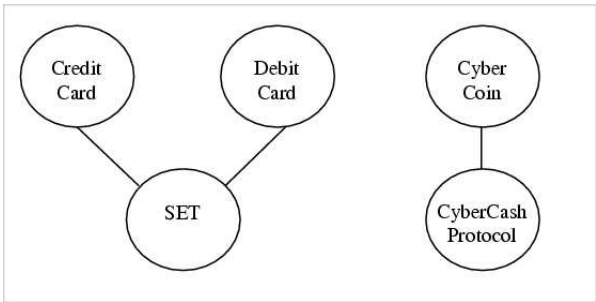

Figure 6.1: Protocol and Instrument Class Compatibility

A protocol may conduct operations using one or more instrument classes. A protocol that can conduct an operation with an instrument class is said to be compatible with that instrument class. For instance, the SET protocol is compatible with both credit card and debit card instrument classes, and may be used to execute payment operations with both credit cards and debit cards (see Figure 6.1). The CyberCash Protocol, on the other hand, may only be compatible with the CyberCoin instrument

class.

6.1.3 Client

A client is a human user or a software agent. A software agent may allow the user to make online purchases or participate in online auctions, for instance.

6.1.4 Digital Wallet

A digital wallet is a software component that provides a client with instrument management and protocol management services. Instrument management and protocol management are defined in Section 3, but, in brief, are services that allow the wallet to 1) install and uninstall instrument classes and protocols, 2) create, update, and delete instruments and protocols, and 3) execute protocols. Digital wallets are capable of executing an operation using an instrument according to a protocol. A digital wallet presents its client with a standard interface of functions; in the case that the client is a human user, this standard interface of functions may be accessed through a graphical user interface (GUI).

A digital wallet can be implemented as a software library that can be statically-linked (at compile-time) or dynamically-linked (at run-time) into an end-user, bank, or vendor application. A digital wallet software library provides the application that it is linked into with instrument management and protocol management services. The digital wallets that are linked into vendor and bank applications provide these management services in the same way that end-user digital wallets do. A vendor's digital wallet, however, may be part of a much larger software application that communicates with order and fulfillment systems. Similarly, a bank's digital wallet may be part of a larger application that is integrated with general ledger, profit & loss, and reconciliation systems.

Furthermore, a wallet is not limited to being a plug-in or applet or some other extension of a web browser. A digital wallet with a graphical user interface may also run as an application on its own, or even as part of a P2P file sharing application that allows users to pay for files they download from others. A digital wallet may

also run on computers that are not connected to the Internet such as smart cards or personal digital assistants. The user interface to the digital wallet may vary in such cases. In the case of a PDA, for example, the digital wallet may have a pen-based user interface. In the case of a smart card, the digital wallet may have no user interface at all. Nevertheless, in each case, the set of functions that the digital wallet's interface presents to its client should be the same.

6.1.5 Peer

A peer is a software thread or process that may be an end-user, vendor, or bank wallet that is capable of performing operations on instruments according to a protocol.

6.1.6 Session

A session is an interaction between two peers, and state information that may be built up over time as a result of interaction between the two peers may be stored in a Session object. One peer is said to initiate a session, while the receiving peer is said to service the session. The Session object keeps track of which peer is the initiator and which is the servicer.

6.2 SWAPEROO Architecture

The architecture we propose in this chapter has four key features. Existing proposals have some of these features, but we believe that none provides all of them in a comprehensive way. We first describe the key features of the architecture before describing the architecture itself.

6.2.1 Key Features

- *Extensible.* A wallet should be able to accommodate all of the user's different payment instruments, and inter- operate with multiple payment protocols. For example, a digital wallet should be able to "hold" a user's credit cards and

digital coins, and be able to make payments with either of them, perhaps using SET [183] in the case of the credit card, and by using a digital coin payment protocol in the latter case. As banks and vendors develop new payment instruments, a digital wallet should be capable of holding new payment instruments and making payments with these instruments. For instance, vendors should be able to develop electronic coupons that offer discounts on products without requiring that users install a new wallet to hold these coupons and make payments with them. Similarly, airlines should be able to develop frequent-flyer-mile instruments so that users may pay for airline tickets with them.

Many existing commercial digital wallet implementations are not extensible. They support limited, fixed sets of payment instruments and protocols, or require extra coding effort to support each instrument and protocol combination. In this scenario, end-users may need to use different wallets depending upon the payment instrument they want to use, and may even need to use different wallets to make purchases from different vendors. The CyberCash wallet, for example, only supports payments using certain credit cards and "CyberCash Coins." [37] Similarly, DigiCash's ecash Purse only supports ecash issued by a set of issuer banks [54]. The Millicent wallet only supports scrip used to make micro-payments [118]. Furthermore, while most existing wallets support at least one protocol for issuing payments, few support protocols for other types of financial transactions such as refunds or exchanges.

There do exist efforts to build digital wallets that support multiple payment instruments and payment protocols such as the Java Wallet [89] and the Microsoft Wallet [130]. In addition, some of these efforts are beginning to gain support as evidenced by initiatives such as CyberCash's development of a CyberCoin Client Payment Component (CPC) for the Microsoft Wallet, allowing users to make payments with CyberCash coins using Microsoft's Wallet. Unfortunately, these wallet architectures do not provide all of the features we describe next.

- *Symmetric.* Vendors and banks run software analogous to wallets, which manages their end of the financial operations. Since the functionality is so similar, it

makes sense to re-use, whenever possible, the same infrastructure and interfaces within end-user, vendor, and bank wallets. For example, the component that manages payment instruments (recording account balances, authorized uses, etc.) can be shared across these different participants in the financial operations. If the wallet components that are re-used are extensible, then we automatically get extensibility at the bank or vendor. So, for instance, an extensible instrument manager will allow the bank or vendor to easily use new instruments as they become available.

Current wallet implementations are often not symmetric. For instance, the components that make up the Microsoft Wallet are client-side objects. Seemingly little infrastructure is shared between the server-side CGI-scripts that process electronic commerce transactions, and the client-side Active/X controls that make up the wallet.

- *Non-web-centric.* Interfaces should be similar regardless of what type of device or computer the user, bank, or vendor application is running on. A digital wallet running on an "alternative" device, such as a personal digital assistant (PDA) or a smart card, for example, has substantial functionality in common with a digital wallet built as an extension to a web browser. Thus, a digital wallet in these environments should re- use the same instrument and protocol management interfaces.

  Many existing wallet architectures such as the Microsoft Wallet and the Java Wallet are heavily web- centric (as they are implemented as Active/X controls or plug-ins, respectively). With the exception of the JECF's (Java Electronic Commerce Framework's) [89] recent inclusion of a smart card API, these wallets do not even begin to address issues surrounding digital wallets running on "alternative" devices (such as PDAs), or in non-web environments.

- *Client-Driven.* The interaction between the wallet and the vendor, we believe, should be driven by the client (i.e., the customer). Vendors should not be capable of invoking the client's digital wallet to do anything that the end-user may resent or consider an annoyance. For example, a vendor should not be able

> to automatically launch a client's digital wallet application every time the user visits a web page that offers the opportunity to buy a product. Imagine what life would be like if, simply by walking into someone's store, the store owner had the right to reach into your pocket, pull out your wallet, hold it in front of you, and ask you if you wanted to buy something! A client-driven approach for building a digital wallet is important because software which customers consider "intrusive" will hinder the success of electronic commerce for all participants involved.

Some commercial wallets are not purely client-driven, since some of them can allow vendors to invoke a user's wallet simply by either: 1) having the user visit the vendor's web site, or 2) having the vendor send the user email. (See Section 6 for details.) When a vendor invokes the Java Wallet, for example, the splash page screen of the wallet applet is brought up and the user is prompted to enter her wallet password.

The wallet architecture we propose here has the features we have described. Specifically, 1) it can inter-operate with multiple existing and newly developed instruments and protocols; 2) it defines standard APIs (Application Programming Interfaces) that can be used across commerce applications for instrument and protocol management; 3) it builds a foundation general enough to implement digital wallets on "alternative" devices in addition to wallets as extensions to web browsers; and 4) it ensures that electronic commerce operations, including wallet invocation, are initiated by the client. Our contribution here is not a set of "new" services for wallets, but rather a flexible architecture that incorporates the best of existing ideas in a clean and extensible way. To verify some of our functionality claims, we have implemented this architecture in Java and C++. The Java version supports most of the features described in the body of the paper. In addition, while the C++ version implements only a subset of the features described here, it does provide support for digital wallets to run on non-traditional devices such as PDAs. These implementations run on the Windows and PalmOS platforms, and implementation details are described in the body of the paper.

6.2.2 Design

In the following, we present an extensible, symmetric, non-web-centric, and client-driven architecture for digital wallets.

In the SWAPEROO architecture, the interaction between a client wallet and a peer wallet roughly works as follows: Once a session is initiated by the client and the peer wallet prepares to service the client, the client can determine what instrument classes are available on the peer wallet, and then select an instrument class that is common to both peers. After an instrument class is selected, protocol management functions are called to determine what available protocols may be used to conduct operations on an instrument of the selected class. Depending upon what protocols are shared, a protocol is selected. The protocol supports certain operations for the selected instrument class, and the client may invoke those operations on an instrument instance. This interaction is described in detail in Section 5.

A digital wallet is an object that has four required key architectural component objects: a Profile Manager, an Instrument Manager, a Protocol Manager, and a Wallet Controller (see Figure 6.2).

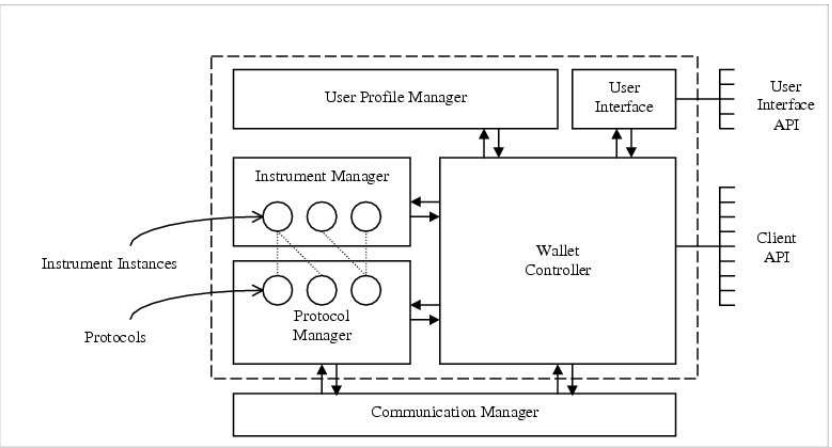

Figure 6.2: The SWAPEROO Digital Wallet Architecture

In Figure 6.2, objects within the dotted lines are the core components of the wallet object . We assume that communication between the core components of the wallet

object is secure such that sensitive data structures containing private information about users and their instruments may be passed between objects within the wallet. In a real wallet implementation, this "boundary" around the secure components of a wallet may be supplied by the operating system by having the core components reside within the address space of a process. This approach, of course, assumes that the operating system is trusted and safe. A trusted operating system will not itself attempt to compromise the security of the wallet, and a safe operating system allegedly has no loopholes which would allow other malicious software it is running to compromise the security of the wallet by reaching into its address space.

Since code modules that implement instruments and protocols may be obtained from different sources, they cannot all be mutually trusted. In particular, it should not be possible for instruments or protocols developed by a software vendor to compromise the privacy, security, or functionality of instruments or protocols developed by another software vendor. At the same time, it is beneficial to have instruments and protocols run within the same address space such that they can exchange private data efficiently.

Two approaches should both be taken to accomplish these conflicting goals. Code modules that implement instruments and protocols should, first of all, be digitally signed by their source (i.e. the software vendor that developed the module). As part of the installation process of such a code module into a wallet, the signature on the code module should be checked to determine if the module can be trusted. Secondly, after installation of the module, a capabilities-based security model needs to be employed to protect modules of code from each other. In such a model, a called object would be able to authenticate the object calling it by verifying that object's digital signature. The security model may also provide support to allow an object calling another object to verify the called object's digital signature. Under such a security model, objects that implement various instruments and protocols may authenticate each other at run-time to prevent potentially "malicious" calls from taking place. Implementing such a security model was beyond the scope of our work. A Java-version of such a security model is addressed by [76], and could be re-used within our work.

Objects that are outside the dotted lines reside in a different address space. However, under a capabilities- based security model in which these external objects are

only granted the appropriate capabilities to access privileged data, they may optionally reside in the address space of the wallet process itself, or can be dynamically-linked into the wallet process.

All components of the wallet are briefly described below except for the wallet's Cryptographic Engine (which has been excluded from Figure 6.2, since all components of the architecture within the wallet may use the Cryptographic Engine to encrypt sensitive data). The Cryptographic Engine resides within the wallet's address space.

We now describe the key components of a wallet:

1. The Instrument Manager manages all of the instrument instances contained in the wallet, and may be queried to determine which instrument classes and instances are available to execute a given payment or other operation.

2. The Protocol Manager manages all of the protocols that the wallet may use to accomplish various operations, and invokes protocols to carry out the interaction between the digital wallet and the vendors and banks. The Protocol Manager relies on the Communication Manager to process low-level communications requests with other peers representing banks and vendors.

3. The Wallet Controller presents a consolidated interface for the entire wallet to the client by coordinating the series of interactions between the Profile Manager, Instrument Manager, and Protocol Manager necessary to carry out high-level requests received from the client, such as "purchase a product." The Wallet Controller hides the complexity of the other components of the wallet, and provides a high-level interface to the client. A non-human client, or software agent, can make method calls on the Wallet Controller's interface through the Client API. A human client may use a graphical user interface (GUI) which may make method calls on the Wallet Controller. The Wallet Controller also handles end-user authentication and access control for operations in the wallet.

4. The User Profile Manager manages information about clients and groups of clients of the wallet including their user names, passwords, ship-to and bill-to addresses, and potentially other profile information as well. In addition, the

Profile Manager keeps access control information about what financial instruments each user has the authority to access, and the types of operations specific users have the privilege to execute with them.

5. The Communication Manager provides the wallet with an interface to send and receive messages between a wallet and a peer by setting up a "connection" with a remote Communication Manager. The Protocol Manager builds on top of the "connection" abstraction to support the concept of a session. A "connection" is typically asynchronous, while communications between peers in a session occur in (message,response) pairs where one peer sends a message, the other peer receives the message, executes some action, and returns a response. Depending upon the implementation of the Communication Manager, the messages may be sent over different types of networks using different communication protocols.

   For example, one implementation of a Communication Manager may send and receive messages over the Internet using HTTP requests and responses over a TCP/IP ethernet network. In this case, a session may be made up of a sequence of several HTTP GET messages and their corresponding responses. Another implementation of a Communication Manager may understand the details of how to talk to another peer using a particular P2P protocol. For instance, when Gnutella nodes communicate with each other, they send each other XML-based messages over HTTP. Yet another implementation of a Communication Manager may send and receive messages over an RS232 port. Note that the Protocol Manager is responsible for making calls to the Cryptographic Engine to encrypt any data that is passed to the Communication Manager, such that the data can be securely transmitted over the communications medium. The Communication Manager cannot be responsible for encryption of sensitive data from the wallet because it is not a core component, and can be replaced by another Communication Manager to run the wallet on another device, or over a different kind of network. If the Communication Manager is relied upon to encrypt sensitive data, then the Communication Manager might be replaced with a malicious Communication Manager that sends all sensitive data to an

adversary.

6. The Client API is an interface provided by the Wallet Controller that may be used by a software agent acting on behalf of an end-user, vendor, or bank.

7. The User Interface provides a graphical interface to the services offered by the Wallet Controller's interface. The User Interface is an optional component of the wallet. Some devices, such as most smart cards, do not have the ability to display a graphical user interface, and hence the Wallet Controller interface must be accessed through the Client API. Note that the User Interface is a core component within the wallet because certain parts of the user interface have access to sensitive user data. For example, the edit box object into which a user enters the password to "unlock" the wallet should run within the wallet's protected address space. On the other hand, customization of the wallet's user interface presents an important branding opportunity for banks and vendors that distribute wallets.

8. The wallet's user interface exports parts of its interface as the User Interface API to satisfy both the privacy and customization requirements. Methods in the User Interface API may be overloaded by software vendors to render customized parts of the interface. The User Interface API also decouples the GUI so that the GUI can be run on a thin client, such as a network computer, while the core components of the wallet can be run on a server.

We will now describe each of the required core components of the digital wallet.

6.2.3 Instrument Management

The Instrument Manager is responsible for managing instrument instances, as well as information about classes of instruments. Instrument Management is made up of the following services: instrument capability determination, instrument installation, instrument storage and retrieval, and instrument negotiation. Before describing the Instrument Manager interface in detail, we will first briefly describe the instrument objects that the Instrument Manager is responsible for storing and retrieving.

An instrument may be a financial instrument that can be used to make a payment, such as a credit card, debit card, or electronic coin. More generally, however, an instrument is made up of state information representing economic value that a protocol can operate on. For example, a digital cash instrument's state can be made up of its dollar (or other currency) value digitally signed by its issuing bank. The protocol used between an end-user wallet and a vendor wallet supports a VERIFY operation which verifies that the cash is authentic by applying the issuing bank's public key to the coin.

While end-user, vendor, and bank wallets share many code modules, some specialization is appropriate. Instruments may have to be managed slightly differently in end-user, vendor, and bank wallets. To illustrate this, we consider a trivially simple digital cash instrument example; please note that in real systems such a naive digital cash scheme is not viable because of real-world security, efficiency, and performance considerations. In this "toy" digital cash example, every time that a vendor receives a digital coin signed by a client's private key, the vendor needs to keep track of that signature in addition to its dollar value in its digital cash instrument. On the other hand, the client may want to keep track of the vendor's signatures on coins she signed for purposes of non-repudiation in her digital cash instrument. Although this is a simple case, we can begin to see that the state information for the digital cash instrument may differ depending upon whether or not the digital cash is being stored in the end-user's wallet or in the vendor's wallet.

Consider a digital cash instrument whose instrument class is Digital-Cash-Instrument. We derive two subclasses from our Digital-Cash- Instrument to manage the different implementations of the digital cash instrument on the different peers. A Vendor-Digital-Cash-Instrument is stored on the vendor, and is able to store a list of client's digital coin signatures. A Client-Digital- Cash-Instrument is able to store the vendor's signature on the client-signed coins. Although the Vendor-Digital-Cash-Instrument and the Client-Digital-Cash-Instrument, for the most part, present similar interfaces since they both derive from Digital-Cash-Instrument, the subclasses present specialized interfaces to their respective callers to access client or vendor-specific instrument information. Note once again that this is a trivial example, and is provided simply

to illustrate that the representation of the digital cash instrument may need to be specialized depending upon whether the peer is a user, vendor, or bank.

In addition to providing access to instrument information in the digital wallet's memory, the Instrument Manager provides interfaces to store and retrieve instruments to and from persistent storage. Note that the Instrument Manager may make calls, if necessary, to the wallet's Cryptographic Engine to encrypt instrument state information in preparation for writing this information to persistent storage, and for decrypting instrument state information when reading this information back from persistent storage.

Upon initialization, the Instrument Manager determines what instrument classes the wallet is capable of using by consulting a configuration file, dynamically determining this through introspection, or by accessing a Capabilities Management service [7]. Alternatively, the Instrument Manager can dynamically download an instrument class from a trusted third-party, and install it. The Instrument Manager may call the Cryptographic Engine to verify that the instrument class code is signed by the trusted third-party. Once the code supporting the appropriate instrument classes is loaded, instrument instances can be created by the user, but more often are loaded from encrypted files on the user's local hard disk, or even potentially from a file server on a network. Finally, the Instrument Manager supports methods to create, modify, commit changes to, and delete instrument instances under transactional semantics.

The Instrument Manager supports methods that query for available instrument classes to conduct instrument negotiation. Note that in our client-driven approach there is no way for a vendor to "Offer" instrument capabilities as there might be in [9], unless the vendor is explicitly queried.

6.2.4 Protocol Management

The Protocol Manager is responsible for managing protocol objects. Protocol Management is made up of the following services: protocol capability determination, protocol installation, and protocol negotiation.

Upon initialization, the Protocol Manager determines what protocols the wallet

is capable of using. As with Instrument Managers, this information can typically be read from a configuration file, determined dynamically through introspection, or can be accessed through a Capabilities Management service [7]. Once this information is determined, a class representing each protocol is loaded into memory, and one instance of each protocol class is instantiated. The instantiation of each protocol instance can be delayed until the Protocol Manager needs to use that protocol.

A protocol is capable of performing operations with an instrument. The exact operations that the protocol can support during a particular session may depend upon the type of peer the wallet is connected to. Consider, for example, a digital coin instrument, and associated digital coin protocols. During a session in which the wallet is connected to a bank, the digital coin protocol may provide deposit and withdrawal operations. On the other hand, while connected to a vendor, the digital coin protocol may provide purchase and refund operations. Protocol objects are responsible for ensuring that such operations take place under transactional semantics.

Furthermore, the Protocol Manager is capable of conducting protocol negotiation with peers for a specified instrument class. That is, the Protocol Manager is capable of determining which protocols the local and remote peers share in common for the specified instrument class. The Protocol Manager accomplishes protocol negotiation with a peer through a Protocol Negotiation Protocol (PNP). Protocol Negotiation Protocols are subclasses of Protocol in our architecture, and are managed in the same way as other protocols are managed, with the exception that the Protocol Manager must successfully load a PNP upon initialization. At least one PNP must load successfully such that it may engage in protocol negotiations with peers. Additionally, although a wallet may load more than one PNP upon initialization, the particular PNP that will be used to conduct protocol negotiation must be fixed by both peers before a session is initiated, otherwise a PNP to decide which PNP to use becomes necessary, and so forth.

Protocols may be compatible with one or more instrument classes, and a protocol object is capable of determining whether or not it is compatible with an instrument class. Once an instrument class is chosen by the client, the Protocol Manager may query the protocol objects it manages to determine which ones are compatible with

the chosen instrument class. If more than one protocol is compatible with the chosen instrument class, the Protocol Negotiation Protocol may pass the list of compatible protocols to the instrument class to allow the instrument class to determine which protocol is optimal for that instrument class.

To illustrate with an example, consider two protocols, Clear-Text-HTTP-Post and SET-Protocol, that are both compatible with a VISA-Credit-Card instrument. When the Protocol Manager calls the Clear-Text-HTTP-Post protocol's compatible-With method passing the VISA- Credit-Card instrument class as an argument, the return value will be true. Similarly, when the Protocol Manager calls the SET-Protocol protocol's compatible-With method passing the VISA- Credit-Card instrument class as an argument, the return value will also be true. Both protocols may not be available on both the client and the peer, but in the case that they are, the PNP may call the instrument class's get-Preferred-Protocol method passing both protocols as parameters. The instrument class is responsible for determining which protocol is optimal (using its own definition of optimal), and the VISA- Credit-Card instrument class may return SET- Protocol in favor of Clear-Text-HTTP-Post to obtain the best possible security for subsequent operations.

In summary, a protocol object is responsible for determining compatibility with an instrument class, the PNP is responsible for determining the availability of protocols between peers, and an instrument class can be called upon by a PNP for determining optimality amongst several compatible protocols available to both peers.

6.2.5 User Profile Management

The User Profile Manager stores information about clients and groups of clients of the wallet. The model we assume is one in which a client is a person or software agent that has authorization to use one or more financial instruments. It is important to note that most existing wallet implementations only allow for clients that are human users and not software agents. However, for the sake of discussion in this section, we will use the terms client and user interchangeably. A group is a set of clients that have access to a set of shared financial instruments, and each client within a group is authorized

to conduct financial transactions with the set of shared financial instruments. When financial instruments may be shared between clients in a group, the instruments may or may not be used concurrently by two or more users in the group depending upon the type of the instrument. Although the User Profile Manager provides access to information about which clients are authorized to use which instruments, and who "owns" or may access which instruments, the Instrument Manager (described in Section 3.2) provides synchronized, concurrent access to use instruments, such that conflicting operations are prevented. For example, a wallet should ensure that digital cash owned by a group of clients does not get doubly spent because two clients attempt to concurrently make a payment using the same digital cash instrument instance.

An example of a group might be a corporate department. Each employee in the department may have access to a set of shared financial instruments, but depending upon an employee's position within the company, the employee may be authorized to only conduct transactions under a certain amount.

6.2.6 Wallet Controller

The Wallet Controller provides an interface to all of the services that the wallet may offer to external objects. The "outside world" cannot see, and does not have direct access to any of the components internal to the wallet such as the Instrument or Protocol Managers. In our implementation, the components of the wallet are all private data members of the Wallet- Controller class.

Once a Wallet Controller method is invoked, the Wallet Controller coordinates the steps that need to be carried out among the User Interface, Profile Manager, Instrument Manager, Protocol Manager, and Cryptographic Engine to execute a payment or other operation. Before accessing or carrying out an operation with instrument instance data, the Wallet Controller makes the appropriate calls to the User Profile Manager to ensure that the user involved has the appropriate privileges to carry out the operation.

6.3 Vendor and Bank Digital Wallets

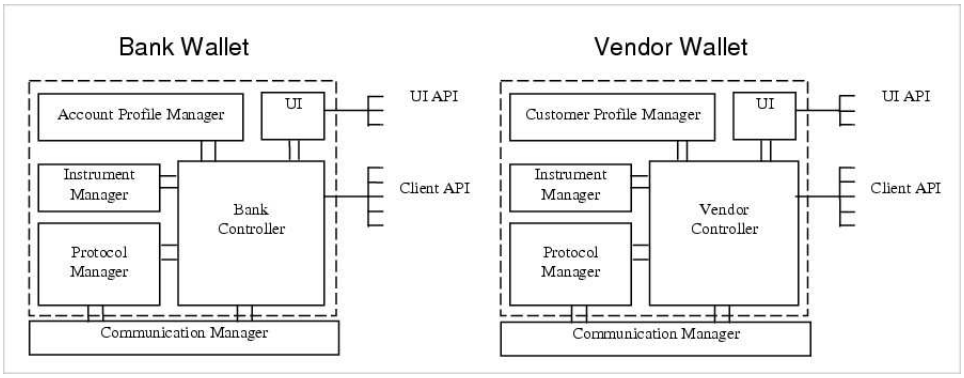

Figure 6.3: Symmetric Vendor and Bank Wallet Architectures

The end-user wallet interacts with bank and vendor wallets to execute commerce transactions. The bank and vendor wallets have architectures symmetric to the user's digital wallet, as shown in Figure 6.3.

In place of a User Profile Manager, the bank wallet has an Account Profile Manager that allows the bank to manage non-financial information about the bank's clients. (Financial information about the bank's clients is stored in instruments in the bank's Instrument Manager.) Similar to the way in which the User Profile Manager maintains access control information about instrument instances, the Account Profile Manager does the same for the bank wallet. The Bank Controller queries the Account Profile Manager before accessing instrument instance data to determine whether or not the user has the appropriate authorization to conduct a transaction; in the case that a users credit line has been overdrawn, or if the users credit card has been cancelled, the Account Profile Manager would return an error response to the Bank Controllers query.

In the vendor wallet, the User Profile Manger is replaced with a Customer Profile Manager, which is used to store access control information about its customers in the case of non-anonymous transactions. For example, the Customer Profile Manager might store the users age, and the Vendor Controller may query the Customer Profile Manager to determine if the user is above a certain age to purchase a product.

A key difference between the Wallet Controller running in the end-user application and the Bank and Vendor Controllers running in the bank and vendor applications, respectively, is that the end-user Wallet Controller drives the wallet interaction, and it is active, in that it generates requests and receives responses. The Bank and Vendor Controllers, on the other hand, are passive, in that they receive requests and generate responses. A Wallet Controller generates requests, and these requests are pushed down through its Communication Manager and out to the peer wallet. Peer wallets receive requests through their Communication Managers, and the requests are propagated to and are serviced by the Bank and Vendor Controllers.

The Instrument, Protocol, and Communication Managers used by the end-user, bank, and vendor are one and the same, and are re-used across the wallets.

6.4 Wallet Operation and Interaction Model

In this section, we describe how our symmetric, client- driven wallets operate and interact during a session to execute a payment. (Executing any other operation such as a refund happens in a similar fashion, and may possibly require only a subset of the steps described below.) In our example here, we assume that the "ordering" phase of the shopping interaction is complete, and the necessary "invoice" (containing information about what products and/or services the end-user would like to purchase) is stored in a property list called inv as a set of (name, value) pairs. We will describe the process from the point of view of the end- user wallet, and we will use the diagram in Figure 6.4 as an aid. To the right of the states in Figure 6.4, we supplement the figure with the method names from our wallet implementation that an application would invoke on the Wallet Controller. Methods in boldface are public in the Wallet Controller's interface, while methods in regular font are private to the Wallet Controller. Finally, the careful reader will note that the methods seem single-threaded; it is because the wallets in our implementation are single-threaded. However, we can easily extend to the multi-threaded case by passing the Session object to each method (excluding initiate-Session()), or by instantiating one Wallet Controller per session.

Although we describe the interaction from the end-user wallet point of view, it

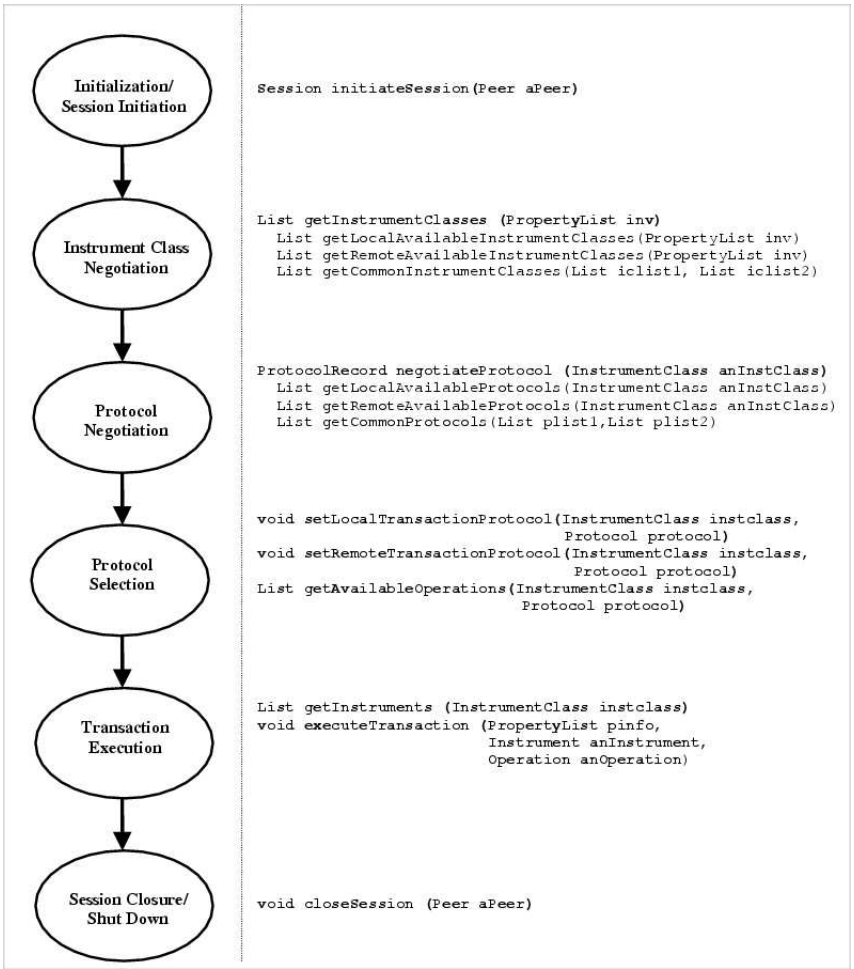

Figure 6.4: Wallet Interaction Model and Wallet Controller Interfaces

is important to keep in mind that a vendor or bank's wallet can also be a client in a wallet interaction. For example, as part of a transaction with the end-user wallet, a vendor's wallet may act as a client and initiate a session with a bank's wallet to obtain credit authorization information for a purchase.

6.4.1 Initialization / Session Initiation

When an application is launched, static initialization takes place before it starts executing. During static initialization, the wallet components including the Instrument

Manager, Protocol Manager, and Profile Manager are constructed and initialized. After static initialization, the wallet may be "unlocked" by supplying login/password information and a session with a peer wallet may be initiated.

The Wallet Controller presents the user's login and password to the User Profile Manager to determine if the user should be allowed to use the wallet. The Wallet Controller also passes the user's password to the Instrument Manager so that it may use the password to decrypt instrument instances and/or the user's private key from persistent store.

To initiate a session, the initiate-Session method in the Wallet Controller is called, passing a Peer object as a parameter. (A Peer object contains the peer's name and details about how to set up a session with that peer.) The Wallet Controller's initiate-Session, in turn, calls the Protocol Manager's initiate-Session to initiate the session. The Protocol Manager makes calls to the Communication Manager to set up the session with the remote peer using the underlying communication mechanism.

6.4.2 Instrument Class Negotiation

The first step that takes place after session initiation is instrument class negotiation. In this step, the clients wallet can determine what instrument classes are known to both wallets by 1) determining what instrument classes are known to its Instrument Manager, 2) determining what instruments are known to the remote Instrument Manager, and 3) computing the intersection.

Those instrument classes that are available to both wallets may offer different terms and conditions for purchasing a given set of products or services. As an example, the price of a product may vary depending upon whether the customer decides to pay using a credit card or using ecash. Furthermore, extended warranties may be offered in the case that a credit card is used to make a purchase. For each available instrument class, the instrument class negotiation step also determines these terms and conditions.

To start instrument class negotiation, the application calls the Wallet Controller's get-Instrument- Classes method with the invoice information, inv, as an argument.

(The invoice information is included to determine the available instrument classes because some instrument classes may not be applicable for certain types of purchases. Also, the inv property list is populated with the terms and conditions described above such that the terms and conditions become part of the invoice.) The Wallet Controller then calls its get-Local-Available-Instrument- Classes method to determine what instrument classes the local wallet supports and are available. The Wallet Controller makes this determination by, in turn, making a call to the Instrument Manager to determine what instrument classes are available to the wallet. Then, the Wallet Controller calls its get-Remote- Available-Instrument-Classes method to determine what instrument classes the remote peer is capable of dealing with.

The call to get-Remote-Available- Instrument-Classes results in a remote procedure call to the peer's Wallet Controller. To respond to the get-Remote-Available-Instrument-Classes procedure call, the remote Wallet Controller calls its get-Local-Available-Instrument-Classes. Other calls of the form get-Local... and get-Remote... described in the following sections work similarly. Also, in our implementation, the get-Remote- Available-Instruments call populates a "price" property stored in inv with a list of (instrument class, price) pairs to indicate the different prices that would be charged for using the corresponding instrument classes in addition to reporting the available instrument classes. The call chain for instrument class negotiation is depicted in Figure 6.5. Solid arrows in Figure 6.5 indicate method calls from the object from which the arrow originates, and dotted arrows indicates the return of control. (The Wallet Controller relies on the Communication Manager to handle the low-level details of executing the remote procedure call described above, but the actual calls that the Wallet Controller makes to the Communication Manager have been omitted from Figure 6.5 to keep the diagram simple. Also, arguments and return values for the methods have been omitted from Figure 6.5 for the same reason, but this information can be found in Figure 6.4.)

To digress momentarily from the end-user wallet's point of view, note that, in Figure 6.5, after the vendor's Wallet Controller calls its local Instrument Manager to determine what instrument classes are available, it may optionally "notify" the Vendor Application. The vendor Wallet Controller gives the Vendor Application

the ability to subscribe to various events and possibly filter the results before they are returned to the end- user's Wallet Controller. Although this capability is not of crucial importance during instrument class negotiation, it is useful to notify the Vendor Application of other events, such as the successful execution of a payment. If the Vendor Application is, for example, a front-end for a vending machine, the vending machine would dispense the appropriate product after receiving notification that payment was successfully transferred.

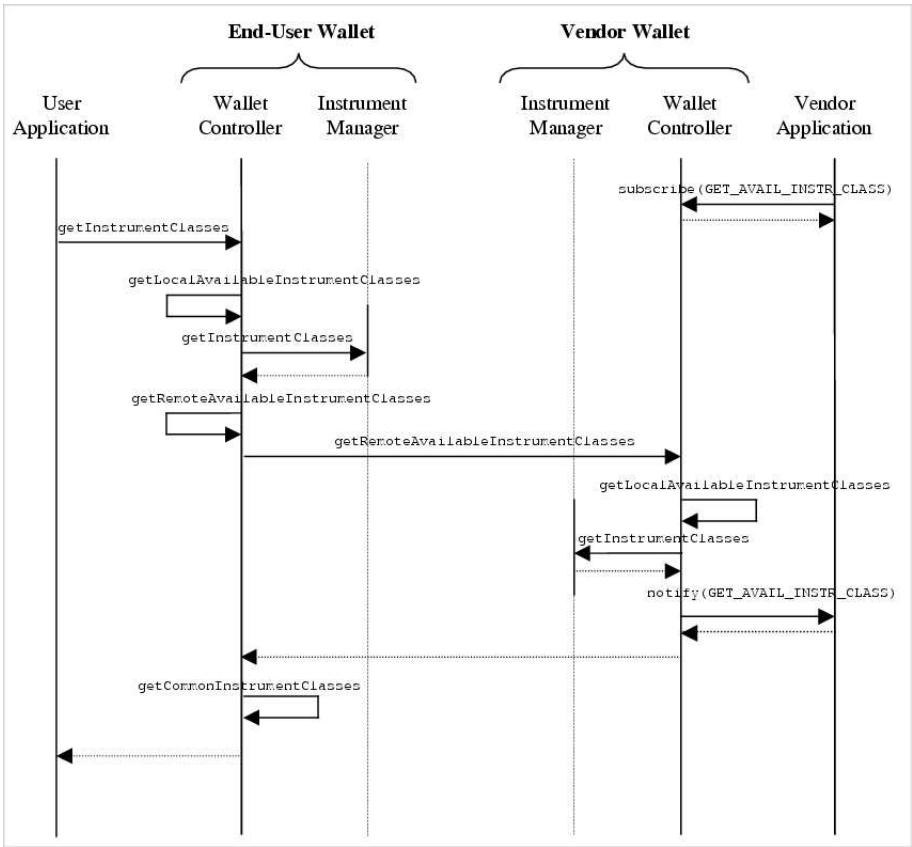

Figure 6.5: Instrument Class Negotiation Call Chain

To complete instrument class negotiation, the Wallet Controller calls get-Common-Instrument- Classes to compute the intersection of available instrument classes. The results received from get- Local-Available-Instrument-Classes and get-Remote-Available-Instrument- Classes are passed as parameters to get-Common- Instrument-Classes.

Once instrument class negotiation is completed, the application is presented with a list of instrument classes with which the user may execute a transaction. The user must select one or more instrument classes before the next step, protocol negotiation, can begin, since the choice of what protocols can be used to execute a transaction is dependent upon the instrument classes that the user selects. In a typical purchase, one instrument will be used to purchase the products and services specified in a specific inv record. However, multiple instruments, possibly of different instrument classes, may be used to make such a purchase, and protocol negotiation for each instrument class would need to be carried out.

6.4.3 Protocol Negotiation

Once the instrument class has been negotiated, the application may call the Wallet Controller's negotiate-Protocol method to start protocol negotiation. The Wallet Controllers negotiate- Protocol method, in turn, calls the Protocol Managers negotiate-Protocol method. The Protocol Managers negotiate-Protocol method then calls the doOperation method of the currently active Protocol Negotiation Protocol (PNP), and also sends a message to the peer informing it that the local wallet is entering the protocol negotiation step. The peer will symmetrically call its PNP's do- Operation.

The default PNP that we implemented calls the Protocol Manager's get-Local-Available-Protocols method to obtain a list of protocols available that are locally compatible with the selected instrument class, and then calls get-Remote-Available-Protocols to determine the protocols that the remote wallet supports for the selected instrument class. The PNP finally calls get-Common-Protocols to compute the intersection. This default protocol negotiation mechanism is similar to instrument negotiation.

Although we implemented the simple PNP above, the remote peer could respond to get-Remote- Available-Protocols with a list of available protocol names and the location of the signed code for those protocol classes. Such an implementation of a PNP may decide to download, dynamically link, and install a protocol that is not

locally supported by the Protocol Manager if the user agrees to permit the wallet to do so. Such a PNP may expedite the case in which the intersection of locally and remotely available protocols is the null set, which would prevent the wallets from executing a transaction. The Java Electronic Commerce Framework (see Section 6), for example, employs such a mechanism as the default behavior. Finally, since the notion of a Protocol Negotiation Protocol has been abstracted out in our architecture, a PNP such as JEPI's UPP protocol (see Section 6) could be used in place of our default PNP .

6.4.4 Protocol Selection

If the result of protocol negotiation for a given instrument class yields one specific protocol, then protocol selection is automatic, and that protocol will be used to execute transactions with instruments of the given instrument class. On the other hand, if more than one protocol is available for a given instrument class, then a preferred protocol needs to be selected by the client's instrument class.

If the PNP reports more than one protocol in common, the Wallet Controller calls the instrument class's get- Preferred-Protocol method as mentioned in Section 3.2. The instrument class selects an appropriate protocol among the ones available based on a variety of parameters, such as the type of device on which the wallet is running, the level of network connectivity and bandwidth available, the dollar amount of the transaction, or user preferences.

For example, when executing a transaction using a wallet on a PDA with limited processing power but with a direct connection (via a docking port or cradle) to a vendor's cash register, an unencrypted session protocol might be preferred over an encrypted session protocol since the link could be assumed to be secure, and key exchange processing overhead need not be incurred to provide for a faster transaction. Such information about device characteristics and connectivity may be obtained from a system properties table similar to the table returned by Java's System.getProperties method. In general, policies regarding how to carry out protocol selection based on such parameters is beyond the scope of this paper; our architecture does, however,

provide a framework and the appropriate "hooks" for such policies to be implemented by implementing and/or overloading the get-Preferred-Protocol method.

Once the protocol is selected for the given instrument class, the Wallet Controller invokes set-Local- Transaction-Protocol and set-Remote- Transaction-Protocol with the instrument class and selected protocol as arguments. Following protocol selection, the application may inquire what operations the selected protocol makes available by calling the Wallet Controller's get-Available-Operations method. Protocols offer different sets of operations depending upon the instrument class over which the protocol executes and, as described in Section 3.2, whether the peer wallet is a bank or vendor.

6.4.5 Transaction Execution

At this point, the application can present the user a list of the available operations and the user can select one of them for execution. After that, the user must select an instrument instance of a previously selected instrument class on which to execute the operation. To obtain a list of possible instrument instances that the user may choose from, the application calls the Wallet Controller's get-Instruments method passing the previously selected instrument class as a parameter. The application presents a list of the names of each of the returned instrument instances to the user, and the user can choose one of the instrument instances.

Once an operation and instrument instance are selected, the application calls the Wallet Controller's execute- Transaction method with these parameters, along with the invoice information stored in inv. After the Wallet Controller verifies that the user has the appropriate privileges to execute the transaction by querying the User Profile Manager, the Wallet Controller calls the Protocol Managers execute-Transaction method. The Protocol Manager then calls the do-Operation method of the appropriate protocol object. The peer wallet is sent a message informing it of the name of the protocol and operation that is to be executed, and it then starts executing that protocol. The peer will symmetrically call the appropriate protocol object's do-Operation method on its side. The protocol objects then execute all of the necessary actions to accomplish the operation.

The operation may or may not execute successfully. An exception will be thrown by the do-Operation method if the operation fails. The remote vendor or bank application will typically subscribe to its wallet's EXECUTE_TRANSACTION events, and will be notified of a failed operation.

Until this point in the discussion, we have described the various wallet states in succession, but it is important to note that the application may decide to negotiate over a different instrument class or select a different protocol. For instance, after transaction execution, the application may return to the instrument negotiation step to select another instrument class for the next operation. As a second example, after selecting an instrument class, and after protocol negotiation and selection takes place, the application may decide that the range of operations available from the vendor for that instrument class and protocol combination is not acceptable. At that point, the application may decide to choose another protocol instead, and protocol negotiation will be conducted once again until a protocol is chosen and another set of operations can be presented to the user. In general, after instrument class negotiation the application can return to any previous step in the wallet interaction. (The extra arrows entailed by this have been left out of Figure 6.4 to keep the diagram uncluttered.)

This capability can also allow payments to be executed in multiple steps with different instruments. For example, if the price of making a given purchase is constant regardless of the instrument class, the vendor may accept receiving large payments as a combination of several smaller payments of different instruments. After instrument class negotiation takes place, the end- user selects multiple instrument classes, and protocol negotiation and selection is completed for each one. The end-user then specifies the amount to be paid with each instrument, and the transaction execution step occurs for each instrument. If the payment fails for any one of the instruments, the user may be prompted to select an alternate instrument, or REFUND operations need to be executed for all the other instruments to abort the payment. The wallets involved must take advantage of transaction management services to ensure that recovery and rollback are executed correctly in the face of machine or network failures.

Finally, as presented above, when the user chooses an instrument class, protocol negotiation and selection are done for that instrument class, and the user then selects

an instrument instance and an operation to execute. If there are many different possible instrument class, protocol, and available operation options, this approach will best guide the user through the interaction. On the other hand, if there are relatively few instrument classes, protocols, and available operation options, then forcing the user to go through all those steps may be overkill. To avoid these extra steps, the application can hide some steps from the user even though both wallets must go through each of the steps.

Consider an example in which the user has only a few instrument instances, and wants to quickly make a payment. After initiating a session, the user application may call get-Instrument-Classes and then call get-Instruments for each instrument class returned to obtain all instrument instances, saving the user of having to select an instrument class. The user application can present the user with a choice of all instruments. After the user chooses an instrument with which to make the payment, the application determines the instrument class from the instrument object by calling the Instruments get-Instrument-Class method, and then does protocol negotiation and selection for the instrument class. If it turns out that no protocol is available for that instrument class, an error is reported to the user. Similarly, if it turns out that a protocol can be selected, but that the PAY operation is not available, an error is reported to the user. If a suitable protocol can be selected, and the PAY operation is available, the user is saved from having to explicitly select an instrument class.

6.4.6 Close Session / Shut Down

In general, the application may close the session at any time. In the typical case, the application may do so after transaction execution, but may also do so after, for instance, finding that it does not share any instrument classes in common with a vendor. A peer application has the option of closing the session (but may just return an error if desired) if the application makes an invalid call, such as calling a method with an instrument class that is undefined or with an instrument class that it did not return as the result of get- Remote-Available-Instruments.

The application should “lock” the appropriate instrument instances upon closing a

session. The results of all instrument negotiation, protocol negotiation, and protocol selection are forgotten upon close of the session, although the wallet architecture may be extended to include a feature to support caching of such information in the future.

After closing all sessions, the user may decide to shut down the wallet, at which point all wallet components save any unsaved information to persistent storage.

6.5 Related Work

- *Java Wallet / Java Electronic Commerce Framework.* [5] The Java Wallet is not purely client-driven. In Sun's Java Electronic Commerce Framework, electronic commerce operations are initiated when a merchant server sends a Java Commerce Message (JCM) to a client's web browser. A JCM has a MIME-type of application/x-java-commerce, and the client's web browser will invoke the Java Wallet once the JCM message is received [21]. By convention, a vendor should not send a JCM to a client unless the client clicks a "PAY" button on a form on the vendor's web site. However, there is nothing preventing a vendor from sending a JCM to the client and invoking their wallet in response to the client simply visiting a page on the vendor's web site. These JCMs may be generated by CGI (Common Gateway Interface) scripts or servlets on the server-side, and sent to the client to invoke the Java Wallet. Alternatively, a JCM may also be generated by an applet that is downloaded to the client browser. In this case, the applet can make a method call on the JECF installed on the client to invoke the Java Wallet. Another way in which a vendor can invoke a user's Java Wallet in an unsolicited fashion is by sending them HTML email with such an applet embedded within it. Users that run HTML email readers that are integrated with their web browser, such as Netscape Navigator or Microsoft Internet Explorer and also have the JECF installed can have their wallet automatically invoked upon reading unsolicited email received from vendors. In order to render the HTML email message, the HTML will be parsed and the Java Virtual Machine will be started to render the applet the vendor

embedded in the email. In this scenario, the applet on the page will start executing, generate a JCM, and make a JECF.startOperation method call passing the generated JCM as a parameter to invoke the user's Java Wallet.

Vendors can use these mechanism to urge customers to make impulse purchases simply by invoking an end- user's wallet when the end-user visits their web site or receives email from them. Customers may resent this feature since it gives vendors the ability to "take the customer's wallet out of his pocket." In our architecture, a user must explicitly launch the wallet to make a payment; this allows the user to make the payment when he or she pleases, and not when the vendor decides it is the appropriate time to pop-up the user's wallet.

Besides being client-driven, our digital wallet architecture supports the notion of a session while the JECF does not. In our model, once a client decides to initiate a session with a vendor, state information may be accumulated throughout the session, and multiple operations may take place during a single session. After initial instrument and protocol negotiation, the negotiated selections are retained as part of the state information in the session, and making additional payments thereafter does not require re-negotiation for each payment operation between the wallet and vendor. This mechanism allows us to implement lightweight instruments and protocols to execute micro-payments. However, in the JECF model, a separate JCM from the vendor is required to execute each commerce transaction, and all of the arbitration, negotiation, and selection may need to be done for each JCM. This can make the execution of micro-payments more costly in the JECF model.

- *Microsoft Wallet.* [130] The Microsoft Wallet is composed of two Active/X controls: an Address Selector control, and a Payment Selector control. This model is also not purely client-driven, since vendors embed these controls on web pages on their web site to prompt the user to make a payment. Furthermore, the selection of the protocol is not client-driven. Within an HTML tag passed as a parameter to the Payment Selector control is an "accepted types" string which contains a list of (instrument class,protocol) pairs that are acceptable to the

vendor. Upon choosing an instrument class, the accepted types are scanned from left to right searching for the first occurrence of the selected instrument class, and the corresponding protocol is chosen. Since the vendor orders the accepted types string, the vendor has the ability to choose a protocol that is advantageous to itself and disadvantageous to the client. For example, the vendor may choose a protocol that may lower its transaction cost, but that may take a longer time to execute over the network, costing the client more in network access charges. In our architecture, the Protocol Negotiation Protocol objects running on both the wallet and the vendor negotiate on the protocol to be used for a selected instrument class, and the client is not locked into using the fixed algorithm hard-coded within the Payment Selector control.

- *IBM Generic Payment Service. [2]* IBM Zurich Research proposes a Generic Payment Service as part of the SEMPER (Secure Electronic Marketplace for Europe) project. The service gives the user the ability to have multiple "purses," each representing a different payment system. The concept of a "purse" in the contexts of the Generic Payment Service roughly corresponds to a combination of an instrument instance and an associated protocol.

- *Shopping Models. [96]* The Shopping Models Architecture formalizes many different customer, merchant, and payment service interactions in terms of order, payment, and delivery event handlers. Our wallet architecture addresses payment in the context of Shopping Models. The wallet and vendor controllers, for example, expand on the interfaces that the CustomerPayment and MerchantPayment event handlers present in the context of that work.

- *U-PAI (Universal Payment Application Interface). [95]* U-PAI proposes a standard interface to multiple payment mechanisms. A U-PAI AccountHandle fits into our architecture as a combination of an Instrument and a Protocol object since the AccountHandle's interface provides methods to access instrument data, such as an account balance, as well as methods to execute a payment such as startTransfer.

- *JEPI (Joint Electronic Payments Initiative). [12]* JEPI was a joint initiative between the W3 Consortium and CommerceNet whose goal was to develop a payment selection protocol as an extension to HTTP. JEPI's UPP [60] protocol is an example of a Protocol Negotiation Protocol in our architecture. Assuming a Communication Manager that is capable of sending HTTP messages (with the appropriate PEP extensions), UPP may be implemented within our architecture as a Protocol Negotiation Protocol.

- *NetBill. [34]* In the NetBill protocol, payment is guaranteed to happen atomically with the delivery of goods by involving a trusted third party. To implement the NetBill payment mechanism in our architecture, the end-user wallet, vendor, and trusted third party applications would execute a NetBill payment protocol that would interact with NetBill money instrument objects.

- *SET. [183]* SET is a secure electronic transaction protocol developed jointly by Visa and Mastercard. As mentioned previously in examples throughout the paper, SET is a protocol object within our architecture that can be used to make payments, as well as execute other operations defined in the SET protocol, with Mastercard and VISA credit card instrument classes.

- *GEPS (Generic Electronic Payment Services). [7]* In the context of GEPS, the Instrument Manager takes advantage of Capabilities Management (CM) and Method Negotiation (MN) services. The Protocol Manager takes advantage of Capabilities Management (CM), Method Negotiation (MN), and Transaction Management (TM) services. The User Profile Manager takes advantage of Preferences Management (PM) services.

6.6 Chapter Summary

We propose a new generalized digital wallet architecture for nodes in a P2P network that is extensible, symmetric, non-web- centric, and client-driven. This architecture not only is extensible enough to inter-operate with multiple instruments and protocols as some existing wallet architectures are, but also incorporates other desirable

features of a digital wallet architecture in a comprehensive way. In particular, the SWAPEROO wallet architecture also

- Factors out symmetric infrastructure and interfaces that may be common among end-user, vendor, and bank wallets. The factoring out of this commonality allows us to design digital wallets for nodes that interact in a peer-to-peer fashion.
- Generalizes to operating environments beyond the WorldWide Web, such that digital wallets can be developed for "alternative" devices such as PDAs and smart cards without re-inventing a new design for the wallet, and
- Ensures that the client is the proactive party in the payment phase of the shopping interaction.

Our proposed generalized digital wallet architecture has been implemented in Java and C++. Using our implementation, we built a digital wallet application for the PalmPilot personal digital assistant, and a vendor application that interfaces with a vending machine; the details of that particular implementation using the SWAPEROO wallet architecture is described in the next chapter.

Chapter 7

E-Commerce on the PalmPilot

In the previous chapter, we described a digital wallet architecture that can be used for e-commerce between nodes in a P2P network. In this chapter, we describe an implementation of the SWAPEROO architecture that allows for payments between a PDA-based user node, and a server-based vendor node. We demonstrate that the PDA-based user node need not be *tamper-resistant*. That is, we need not build explicit hardware or software mechanisms that cause the device to self-destruct if a malicious party attempts to access its memory in an unauthorized fashion. Rather, we can provide an acceptable level of performance and security even without tamper-resistance.

The implementation work that we describe in this chapter was conducted in 1999. As such, experiments and performance measures reflect hardware and software available at that time. While the absolute performance of the operations that we describe is surely outdated, the *relative* performance of the various cryptographic operations (signatures, verifications, etc.) may still be valid. The characteristics of the hardware (memories, CPUs, persistent storage) have experienced significant change, as components have been miniaturized and their performance has increased orders of magnitude, as predicted by Moore's Law. Indeed, the functionality of the PDA (Personal Digital Assistant) has been folded into many cellular phones of today.

Nevertheless, for the purposes of this dissertation, our main point in this chapter is that the devices we describe can be viewed as non-tamper-resistant, portable

commerce devices that can function as peers in a P2P network.

7.1 Introduction

The explosive growth of the market for Personal Digital Assistants (PDA's) has led to a wealth of new applications for them. In this chapter, we experiment with e-commerce for PDA's. Our motivation is clear: since consumers are already carrying digital assistants, why not use them for payments? For example, one could walk up to a vending machine, connect the PDA to the machine via the infrared link and make a purchase. One application of our system does just that (see Section 7.4.2). One may wonder whether electronic commerce on a PDA is any different than e-commerce using other digital devices such as smartcards or desktops. The answer is simple – PDA's are a middle ground between smartcards and desktops. They posses advantages and disadvantages over both. As a result, payment schemes and e-commerce applications need to be fine tuned to properly execute on a PDA.

When compared with smartcards, PDA's appear to be at a disadvantage: (1) they are not *tamper resistant*, (2) unlike many smartcards they are not equipped with cryptographic accelerators, and (3) they do not have limited lifetime as many smartcards do. These features (especially tamper resistance) are helpful in simplifying real world payment schemes. On the other hand, PDA's have several advantages over common smartcards. First and foremost, they have a direct line of communication with the user. Common smartcards can only interact with the user through an *untrusted* reader. In addition, unlike smartcards PDA's have a reasonable (yet limited) amount of non-volatile memory[1]. Consequently, they can store a longer transaction log. PDA's are also better at general purpose computations; for example, computations that take place during protocol negotiation. We note that when comparing smartcards and PDA's the issue of unit cost is irrelevant – PDA's may become just as widely deployed as smartcards.

[1]The PalmPilot III comes with 2 megabytes of RAM. A Windows CE machine typically has four to eight Megabytes. We note that most memory on a PDA is devoted to applications such as an appointment book. A commerce application can only use a small portion of the memory on the device.

When compared with desktops, PDA's appear to be at a disadvantage once again. Due to their limited memory capacity, they can only store a limited amount of financial instrument data, and a limited number of transaction certificates. Similarly, due to their limited computing power they cannot engage in complicated cryptographic protocols. An intensive protocol such as SET [183] may take too long to execute on a PDA. On the other hand PDA's are portable and can be used in many environments where a desktop is not available.

The above discussion explains why many existing payment systems cannot simply be ported to a PDA. One must design the system keeping in mind both the limited security features and the limited computation resources. In this paper, we describe our experience with building a payment system for 3Com's popular PalmPilot [1]. Our system is designed to be portable, and the lessons we learned apply to other PDA's, e.g. ones based on Windows CE. Our choice for using the PalmPilot is due its current dominance of the PDA market. We note that existing financial applications [65] for the PalmPilot enable one to keep track of spending, but none provide a payment scheme.

To implement an e-commerce system one must start with a basic framework for making payments. Our starting point is the digital wallet architecture described in the previous chapter. We stress that the wallet only provides the skeleton for this work. Our main focus in this chapter is the design of payment components that can be adequately used on a PDA. We describe these in Section 7.3. The system implementation details are presented in Section 7.4, where we also describe our first vendor: the Pony Vending Machine. We begin by describing the (lack of) security features on the PalmPilot.

7.1.1 Existing PalmPilot Security Features

All data on the PalmPilot is stored in either ROM or RAM. The PalmPilot does not have a disk drive or any other form of persistent memory. The RAM on the PalmPilot is divided into *dynamic* and *storage* memory. Dynamic memory is used as working space. It holds the program stack and other short term data. On the other

hand, storage memory is non-volatile and plays the role of a disk drive on desktops. *Databases* in the storage RAM are the equivalent of files on a disk drive. A database on the PalmPilot is made up of a sequence of records. Database access on the PalmPilot is somewhat different than a traditional operating system. In traditional systems, a file is read into memory before data stored in it can be accessed or modified. On the PalmPilot, database records are always in memory. There is no need to move them into dynamic storage to operate on them. They can be *edited in place*.

Unfortunately, Palm OS provides very little support for access control. Although databases have a *creator ID* associated with them, the operating system does not prevent an application from opening a database that does not belong to it. Consequently, malicious applications can easily tamper with sensitive data on the PalmPilot. Individual records in databases have a "secret" attribute associated with them. However, the core OS only views this attribute as a suggestion. There is nothing preventing an application from gaining access to records marked secret. Essentially, the secret attribute tells the application not to display these records unless the user types in a password. The application itself is not prevented from processing records marked secret. It is a bit surprising that although the OS provides a facility for requesting the user to enter a password, it does not provide any support for encrypting data using the password.

The lack of support for access control has prompted a number of developers to attempt to remedy the situation [184]. A number of shareware applications for the PalmPilot enable the user to store data in an encrypted database. This is especially useful when using the PalmPilot to store sensitive information such as PIN's and login passwords. Unfortunately, at the moment these applications use weak encryption. We note that Ian Goldberg [75] ported to the PalmPilot the cryptography-related parts of Eric Yung's SSLeay library. We make use of this port in our protocols.

To summarize, access control on the PalmPilot is poor. The operating system does not enforce access control and there is no support for privacy via encryption. This is very different than a typical smartcard operating system where there is a clear partitioning of memory between applications. We hope future versions of the PalmPilot will pay more attention to security issues. This is likely to be the case

as indicated by the recent collaboration between Certicom and 3Com. Certicom is integrating its elliptic curve encryption technology into Palm OS. Finally, we note that things appear to be better on Windows CE, where some support for access control is provided by the operating system.

7.2 E-Commerce for a PDA

If PDA's are to be used for digital payments, they will most likely be restricted to small payments (i.e. transactions under 10$). For larger payments one may wish to rely on stronger security properties. Our hope is that PDA's may be used in both local and remote payment transactions. For local transactions, a user may walk into a shop, place the PalmPilot in the vendor's cradle (or use the infrared port) and make a payment. For remote transactions, one may connect the PDA to the Internet (using a PDA modem or by using a desktop as a gateway). The two modes are different: for local transactions there is no need to encrypt communication on the wire. For remote transactions, one must first establish a secure link to the vendor. Throughout the discussion in this section we do not distinguish between the two modes. At the moment, our implementation only supports local transactions that do not require link encryption.

7.2.1 Performance of Cryptographic Primitives on the PalmPilot

We provide timing measurements for cryptographic primitives running on the PalmPilot[2]. These timing measurements help determine if a payment scheme is feasible for the PalmPilot or if it is too complex. The PalmPilot Professional runs on a Motorola DragonBall chip (68K family) at 15MHz. The figures below are given in milliseconds.

We first note that generating RSA key pairs is very expensive on the PalmPilot. Typical PalmPilot users turn on their PalmPilot perhaps five or six times per day,

[2]The performance figures for DES, SHA-1, and RSA operations were obtained using Ian Goldberg's port of SSLeay. The figures for ECC operations were obtained using CertiCom's Security Builder SDK Release 2.1

Algorithm	Time	Comment
DES encryption	4.9ms/block	4900ms for 1000 encryptions
SHA-1	2.7ms/block	2780ms for a 1000 long hash chain
512 bit RSA key generation	360 seconds on average	
512 bit RSA sig. generation	7028ms	
512 bit RSA sig. verify	438ms	$e = 3$
512 bit RSA sig. verify	1376ms	$e = 65535$
163 bit ECC-DSA key generation	590ms	
163 bit ECC-DSA sig. generation	800ms	
163 bit ECC-DSA sig. verify	2340ms	

Table 7.1: Timing measurements for cryptographic primitives on the PalmPilot

and use the device on the average of a few minutes each time. Even when on, the CPU spends most of its time sleeping, waiting for user input, due to the power managment routines built into PalmOS. Given this usage model, the batteries last for approximately four to six weeks for the average user. To generate a 512-bit RSA key pair, six minutes of pure computation time is required of the CPU. In addition to inconveniencing the user at wallet setup time, RSA key pair generation drains the PalmPilot's batteries. One may argue that the key pair could be generated on a desktop and downloaded to the PalmPilot, but this limits mobility since setup must take place on the user's PC. Clearly, an arbitrary PC cannot be trusted to generate the user's private key. If 1024-bit keys are to be generated on the PalmPilot, approximately 20 minutes of pure computation would be required. On the other hand, elliptic curve key pair generation is about two orders of magnitude faster.

As might be expected, DES encryption and SHA-1 hashing are relatively fast as compared with signatures. This suggests that an efficient implementation of a payment protocol for a PDA-based platform should attempt to take advantage of these operations in favor of signatures as much as possible.

RSA signature generation is somewhat slow, taking approximately five seconds. Our payment protocols take about 900ms seconds when stripped of the cryptography. Hence, RSA signature generation time is significant in comparison to the entire transaction time. These figures suggest that protocols involving multiple signature generations (e.g. SET) are too complex for the PalmPilot.

Our choice of using 512 bits RSA, rather than 1024, is due to the fact that our application is targeted towards small payments. For small payments, 512 bits may provide adequate security.

The figures for 163-bit ECC-DSA key generation and signature generation are considerably more efficient than the corresponding 512-bit RSA operations which suggests that ECC-based key generation and signature operations are more feasible for use in commerce protocols on the PalmPilot. ECC-based verification does, however, take several times longer than RSA-based verification.

To summarize, there exists an asymmetry between the performance of RSA and ECC operations. We would like our payment protocols to be as efficient as possible, and both signature generation and verifications may be necessary parts of the protocols. If we need to generate signatures on the PalmPilot, we would like to generate ECC-based signatures, since ECC-based signature generation is faster than RSA signature generation. On the other hand, if we need to verify signatures on the PalmPilot, we would like to use RSA since RSA signature verification is faster than ECC-based signature verification. We will show how we take advantage of this asymmetry in our payment protocols in Section 7.3.2.

7.2.2 Authentication

Common smart cards do not have any means of directly communicating with their owner. They must do so through an untrusted card reader. As a result, it is difficult for the owner to authenticate himself or herself to the card. Often, smartcards require the owner to enter a password. However, there is nothing preventing the machine operating the card reader from recording and replaying the password [72]. The smartcard cannot use challenge-response authentication since humans are incapable of participating in such protocols.

On the other hand, a PDA has a direct line of communication with its owner. To activate a transaction, the wallet controller prompts the user for a password. This password is used for two purposes: first, it authenticates the owner to the PDA. Second, as we shall see in the next section, we use the password to decrypt

specific instrument data. Note that the user is prompted for a password only once a transaction is about to take place. Prior to this, the PDA may freely communicate with the vendor.

Unfortunately, entering a password on the PalmPilot is somewhat painful. Clearly, it is desirable to enter the password in no-echo mode. However, since characters are entered using error prone Graffiti, no-echo makes it hard to correctly enter the password. We note that the standard security utility on the PalmPilot prompts the user to enter a password in full echo mode!

7.2.3 Memory Management and Backups

Since Palm OS provides very limited access control, one must ensure that malicious applications do not have access to private financial information. They must also be prevented from tampering with such data. Our solution is to store all instrument data in encrypted form. This is automatically done by the Instrument Manager [40]. Encryption is done using the user's password which must be entered in order to initiate a transaction. We also append a cryptographic checksum to ensure data integrity.

Nevertheless, malicious applications can delete encrypted instrument data blocks. There is no way to prevent this other than to rely on a backup copy stored on a desktop PC. The HotSync Manager application distributed with the PalmPilot automatically backs-up these databases. In the case that the wallet application discovers corrupted data, it may ask the user to perform a HotSync with the desktop. Of course, care must be taken to ensure that already spent digital cash is not doubly spent after the backup process.

7.3 PDA-PayWord: A Payment Scheme Optimized for a PDA

The lack of tamper resistance on a PDA is unfortunate, although not fatal. Tamper resistance on smartcards can simplify payment protocols such as those used in stored value techniques (see e.g. Mondex [135]). However, one must keep in mind that

tamper resistance is not impenetrable (see [5]). As a result, payment schemes on smartcards must also have some mechanisms in place to discourage tampering. Here, a PDA has an advantage over a smartcard: it has more memory and processing power, and can incorporate stronger mechanisms to discourage tampering. We chose to avoid stored value techniques and instead focused on a hash-chain-based digital cash scheme based on PayWord [173]. PDA-PayWord is our implementation of PayWord.

Hash-chain-based micropayment schemes have been studied extensively. The idea of using hash chains to reduce the number of signature generations was independently discovered by a number of groups. These include PayWord [173], Pederson's tick payments [153], micro-iKP [83] and NetCard [6]. Jutla and Yung [23] study a variation where hash trees are used rather than chains. Hash chains are based on a technique due to Lamport [102].

Since we wish to focus on the PalmPilot (user) to vendor interaction (rather than vendor to bank), we allow the bank and vendor to reside on the same machine. This tight coupling between the vendor and bank can be broken at any point in the future.

7.3.1 Overview of PDA-PayWord

This section contains a high-level overview of PDA-PayWord. A discussion of the design trade-offs involved in PDA-PayWord is given in the next section, and a detailed discussion of our PalmPilot implementation can be found in Section 7.4.3.

PDA-PayWord is a pre-pay, vendor-specific variant of a hash chain system. To setup a chain of coins the PalmPilot generates $Y_k = h^{(k)}(Y_0)$ where $h^{(i)}$ denotes a repeated iteration of a hash function (we use SHA-1). Y_0 is kept on the PalmPilot as part of a *hash chain instrument*. The PalmPilot sends Y_k, k, d to the bank. The bank returns a *hash chain certificate* containing Y_k, k, d, a vendor-id, an expiration date, and a signature of the certificate using the bank's private key. k is the pre-paid height of the hash chain and d is the denomination of each Y_i. To spend the ith coin when Y_{k-j} was the last coin spent, the PalmPilot sends $Y_{k-j-i} = h^{(k-j-i)}(Y_0)$, i, and the hash chain certificate to the vendor. Initially $j = 0$, and $i = 1$. Recall, these are vendor specific chains and the vendor can verify that the PalmPilot is not

double-spending or over-spending.

7.3.2 Discussion of PDA-PayWord Design Choices

As described in [173], PayWord was designed to amortize the cost of signatures across multiple purchases and minimize on-line communication. In addition to these goals, PDA-PayWord: 1) minimizes the number of and time spent executing cryptographic operations on the user's wallet, and 2) minimizes storage requirements for the hash chain instrument on the user's wallet.

In PDA-Payword, the user's wallet must first generate a hash chain instrument and obtain a hash chain certificate from the bank that binds the "coins" in the hash chain to real value. In this "withdrawal" phase of the protocol, the user's wallet sends Y_k, k and d to the bank to request a widthdrawal of $k * d$ dollars. The bank needs to ensure that the request is being made by the user herself, and therefore PDA-Payword requires that this widthdrawal request be signed. Since this signature needs to take place on the PalmPilot, the withdrawal request is signed using an ECC-DSA signature (as opposed to an RSA signature) to minimize the signature generation time. (Recall that from Table 7.1, ECC-DSA signature generation takes 800ms, while RSA signature generation takes over 5000ms.)

After the bank receives the withdrawal request and verifies the user's signature, the bank then generates a hash chain certificate, signs it, and sends it to the user's wallet. The user's wallet then needs to verify that the bank's signature on the certificate is authentic. This verification takes place on the PalmPilot, and we would like the verification to be as efficient as possible. As such, in PDA-PayWord, the bank signs the hash chain certificate using a RSA signature (as opposed to an ECC-based signature) since RSA verification is several times more efficient on the PalmPilot than ECC-DSA signature verification. By taking advantage of the asymmetery between the performance of RSA and ECC signature and verification times, we are able to minimize the amount of time spent on the PalmPilot executing cryptographic operations.

To achieve our second goal of minimizing storage requirements, PDA-PayWord

only stores Y_0 as part of the instrument data in the user's wallet, rather than the entire hash chain. The chain is recomputed for every transaction. This is done to save storage space on the PalmPilot, and incurs little cost since hashing on the PalmPilot is sufficiently fast. In contrast, on a smartcard, one might have to store every ith payword along the hash chain since hashing may not be as quick. Of course, a PalmPilot could also take advantage of this to reduce the time spent hashing if the user decides to give the wallet more memory in which to run. In our implementation, we conservatively assumed the user would not want the wallet to use more than the minimal amount of storage necessary. Also, thanks to its speed, the PalmPilot can manage longer hash chains than a smartcard.

7.4 System Design

We briefly summarize some of our design choices. We demonstrate the user interface as well as explain the internal implementation of the various protocols involved. In Section 7.4.2 we describe our first vendor, a vending machine in our building.

7.4.1 Wallet Design & Implementation

Our SWAPEROO-based implementation consisted of a digital wallet application written in C++ for the PalmPilot, and a Java implementation of the vendor application. Timing measurements were obtained by running the user's wallet on a PalmPilot Professional with 1MB of RAM. The PalmPilot Professional runs on a Motorola DragonBall chip at 15MHz. The bank/vendor application runs on a dual processor 400 MHz Pentium II PC with 128MB of RAM.

Communication between the PalmPilot and the PC was conducted using TCP/IP over a serial RS232 communications channel.

User Interface

The user interface is a collection of forms running on the PalmPilot. When interacting with the vendor, they guide the user through the choice of available items for sale.

The user chooses an item and is then asked to choose the payment method, i.e. the payment instrument to be used for the purchase. Once the instrument is selected, the appropriate purchase protocol is activated. As a first step, the user is asked to enter a password to unlock the instrument data. A subset of the forms guiding to the user through its interaction with the vendor is presented in Figure 7.1[3].

Product negotiation

We implemented a simple protocol enabling a vendor to download a set of items along with their prices onto the PalmPilot. This protocol is used to convey the customer's choice back to the vendor. Once the required item is agreed upon by both parties the wallet controllers on both the vendor and client take control and execute the transaction.

Instrument and Protocol Negotiation

Instrument negotiation is done by computing the intersection of available instrument types on the vendor and the customer wallets. The set of instruments in the intersection is presented to the user who chooses one of them. We hope the list of available instruments will increase over time. Once the transaction is initiated, the two parties negotiate which protocol is to be used to carry out the transaction (for a given payment instrument there may be multiple protocols offering different levels of efficiency and security). Protocol negotiation is done via a Protocol Negotiation Protocol (PNP) as explained in [40].

Instrument Storage on the PalmPilot

Instruments carry some financial information with them. For example, in the case of a PDA-PayWord instrument, the data includes the bottom end of the hash chain, the current position on the hash chain to be spent, and other parameters as well. Each of our instruments is implemented as an encrypted database on the PalmPilot.

[3]These figures were produced by taking a snapshot of our wallet application running under the copilot emulator

Figure 7.1: Example interaction with the vendor. The top form displays available items for sale. The bottom form displays the user's choice and the list of available payment instruments. The textual description of items will eventually be replaced by icons.

This data resides in non-volatile memory and is backed up to the desktop during a HotSync.

Link Encryption

At the moment the implementation assumes all transactions are local, i.e. the PalmPilot is placed into the vendor's cradle. The PalmPilot does not establish a secure link to the vendor since there is little concern that a direct line between the PalmPilot and vendor can be tapped.

7.4.2 The Pony Vending Machine

The Pony vending machine[4] supports an interface through a proprietary connection. We were able to interface with it and send it commands through a laptop-PC which serves as a proxy to it. The laptop-PC runs the vendor application, and may issue commands to the Pony vending machine through its proprietary interface. The digital wallet application running on the PalmPilot connects to the vendor application running on the laptop-PC.

When the PalmPilot connects to the Pony-Vendor (the laptop-PC), the user can choose from a list of available products. The set of available products is downloaded onto the PalmPilot as soon as the connection is set up. The user then tells the PalmPilot directly which item to purchase and which instrument to use for payment. The PalmPilot then initiates a purchase protocol with the Pony-Vendor. Once the appropriate funds are transfered to the Pony-vendor it instructs the Pony-vending machine to dispense the required item. The connection is then closed and the Pony-Vendor is ready to service the next customer.

7.4.3 PDA-PayWord Implementation Details

Withdrawal Protocol

Before being able to make purchases from the vendor, the user must first make a "withdrawal" from the bank. In our current system, the user takes his or her PalmPilot to a bank teller and inserts her PalmPilot into a cradle attached to the bank teller's computer. The user authenticates herself by entering her password. The user then enters the withdrawal amount on the PalmPilot, and the PalmPilot generates a *hash chain instrument*. It then executes the withdrawal protocol. Since our implementation uses 5-cent denominations ($d = 5$), the wallet divides the withdrawal amount by 5, yielding the hash chain size, k. The wallet then constructs a hash chain instrument by generating Y_0 ... Y_k. Average hash chain generation times for various

[4]The Pony (short for Prancing Pony) is a vending machine on the fourth floor of the Gates Computer Science Building on the Stanford campus. Similar to the Prancing Pony in J.R.R. Tolkien's "The Fellowship of the Rings," the Pony vending machine is a common place which those in need of nourishment may go to.

Amount ($)	Hash Chain Size (words)	Average time (ms)
5	100	504
10	200	896
20	400	1667
50	1000	3970

Table 7.2: Average Hash Chain Generation Timing Results

field:	Y_k	k	d
byte-length:	20	2	2

Table 7.3: Withdrawal Request Message Format

dollar amounts using our 5-cent denomination hash chain instrument are shown in Table 7.2.

For amounts up to $10, a single 5-cent denomination hash chain instrument might provide acceptable performance on the PalmPilot; for larger amounts, using multiple hash-chains with larger denominations might be necessary to achieve acceptable performance.

The user's wallet then kicks off the withdrawal protocol by sending a signed *withdrawal request* to the bank. The user's password is used to decrypt the her private key, and the private key is used to sign the withdrawal request. For efficiency, an ECC-DSA signature is used to sign the withdrawal request. The withdrawal request is shown in Table 7.3.

The bank receives the withdrawal request, and verifies the signature on the request. The bank then either deducts the withdrawal amount $= k * d$ from the user's "bank account," or the user gives the bank teller the amount in physical currency. If the user has the appropriate funds in his or her account, or gives the bank teller the appropriate amount in non-counterfeit currency, the bank teller approves the transaction. Upon approval, a *hash chain certificate* is sent to the user's wallet and stored in the user's hash chain instrument. Note that the user's hash chain instrument has no value without a signed hash chain certificate. The hash chain certificate is also stored on the vendor's wallet in a vendor-side hash chain instrument that can be

field:	d	k	*vid*	Y_k	*sno*	*sublen*	*subject*	*elen*	*expdate*	*siglen*	*sig*
byte-length:	1	2	1	20	4	2	variable	2	variable	2	variable

Table 7.4: Hash Chain Certificate Format

d	denomination
k	hash chain size
vid	unique vendor identification number
Y_k	top word of hash chain
sno	unique serial number of this certificate
sublen	length of variable subject field
subject	"subject" of this certificate. currently used to store a comment.
elen	length of variable expiration date field
expdate	text representation of expiration date in "UNIX" format
siglen	length of variable signature field
sig	RSA signature of SHA message digest of this certificate

Table 7.5: Description of Hash Chain Certificate Fields

quickly looked up during execution of the purchase protocol. The vendor-side hash chain instrument also stores the last Y_i spent by the user. (In the case that the bank teller does not approve the transaction, an error message is sent to the user's wallet.)

Generation of the hash chain certificate requires the bank to generate a signature, and an RSA signature is used to sign the hash chain certificate in PDA-Payword (as explained in Section 7.3.2); note that this task takes place on the bank-PC. The PalmPilot verifies the bank's signature, but does not need to generate a signature itself. The signature generation time for our 400 MHz dual processor Pentium II bank-PC was only 50ms on average, and, as we will see, is negligible compared to the signature verification time on the PalmPilot. The total time for the PalmPilot to generate and sign the withdrawal request, send it to the bank, and receive a hash chain certificate in response was 1415ms on the average.

A hash chain certificate is shown in Table 7.4. Many of the fields in the hash chain certificate have already been described, but a brief explanation of all of them can be found in Table 7.5.

Using 512-bit RSA signatures on a 20-byte message digest of the certificate, the average length of the entire hash chain certificate is 358 bytes. (The length of a 512-bit RSA hash chain certificate can vary because of the variable subject and expiration date fields.) The bulk of the certificate is made up of the signature field, which is 302 bytes. Although a 512-bit RSA signature on a 20 byte message digest is only 64 bytes, the signature field also contains the bank's public key which can be used to verify the signature (81 bytes), and a public key certificate (151 bytes) signed by the certificate authority that can be used to verify the bank's public key. The CA's public key is hard coded in the wallet application.

Once the hash chain certificate is received by the user's wallet, the wallet verifies the hash chain certificate. Verification of the hash chain certificate took approximately 1000ms in our implementation. Note that two RSA verifications are necessary to check that a hash chain certificate is authentic: one verification using the bank's public key to check the signature on the certificate, and one verification using the CA's public key to check the signature on the bank's public key certificate. The RSA verifications take just under 500ms each, and hence a total hash chain certificate verification time of 1000ms is reasonable.

If verification of the hash chain certificate is successful, the certificate becomes part of the user's hash chain instrument. (If the hash chain certificate does not verify, an error message is displayed, and the user should demand his or her money back!)

To summarize, the key components contributing to the total transaction time of the withdrawal protocol are 1) hashing required to construct the instrument, 2) ECC-DSA signature of the withdrawal request, 3) communication overhead between the PalmPilot and the bank, and 4) verification of the hash chain certificate. The time reqiured for (1) is shown in Table 7.2, and is dependent upon the amount and denomination being withdrawn. The time required for (2) and (3) combined is approximately 1415ms, and the time required for (4) is approximately 1000ms.

Hence, withdrawal times range from just under 3 seconds to just under 6.5 seconds for instrument amounts from 5$ to 50$. From a usability standpoint, these withdrawal times would seem to be acceptable for most users. [5]

[5]Indeed, for larger amounts, the instrument creation time becomes the dominating factor and can

field:	Y_{k-j-i}	i	*hash-chain-certificate*
byte-length:	20	2	variable

Table 7.6: Purchase Request Message Format

instrument amount($)	*hash iterations required* (words)	*transaction time*(ms)
5	70	1090
10	170	1467
20	370	2267
50	970	4580

Table 7.7: Purchase Transaction Times

Purchase Protocol

Once the hash chain instrument's construction is complete by adding the hash chain certificate to it, a purchase may be accomplished by having the user's wallet send a *purchase request* to the vendor application. If the user has already spent j hash words, then to spend the next i words, the user's wallet sends the purchase request message shown in Table 7.6 to the vendor application.

The number of hash words to spend, $i = price/d$, is determined based on the price of the product the user wishes to buy and the denomination of the hash chain instrument. Since the hash chain instrument on the user's wallet only stores Y_0, the user's wallet needs to compute $k - j - i$ hashes to construct the purchase request message.

After the vendor wallet receives the purchase request, it retrieves the corresponding vendor hash chain instrument from its Instrument Manager and checks that $h^{(i)}(Y_{k-j-i})$ is equal to the Y_{k-j} that it had stored after the instrument's last use. Note that the first time the instrument is used, $j = 0$ and the vendor wallet checks that $h^{(i)}(Y_{k-i}) = Y_k$. The vendor wallet also checks the validity of the hash chain certificate that is presented in the purchase request, and if the certificate is valid,

be reduced by using larger denomination coins (choosing a larger value for d). Using multiple hash chains, each with a different d, can also be employed to reduce the amount of hashing necessary, but would necessitate the verification of multiple hash chain certificates.

the vendor wallet stores Y_{k-j-i} in its vendor-side hash chain instrument and sends back "OK" as the *purchase response*. If any of the above checks fail, an "ERROR" purchase response is sent back to the user wallet.

Purchase protocol timing measurements to do a first-time $1.50 buy with 5-cent denomination hash chain instruments of varying initial sizes are presented in Table 7.7.

7.5 Chapter Summary

Our experiments show that a PDA may be viewed as a *portable commerce device without tamper resistance* that is suitable for small payments. PDAs are computationally more powerful than smartcards. However, since they do not contain cryptographic accelerators certain operations take longer, and the performance of cryptographic primitives need to be taken into account when designing commerce protocols for PDA's. For instance, RSA signature generation is slow, while RSA verification is fast. On the other hand, ECC-based signature generation is fast while verification is slow.

Based on our experiments in this chapter, commerce protocols can be implemented to perform transactions efficiently on a PDA platform. PDA-PayWord is an example of an implementation of a commerce protocol that is designed to achieve acceptable performance on a PDA by taking advantage of the performance characteristics of both RSA and elliptic curve cryptograpy on the PalmPilot.

In conclusion:

- The commerce architecture that we described in the previous chapter can be implemented efficiently, even in a non-web-centric setting where client devices have constrained processing capabilities;
- both security and performance can be achieved without requiring that the client device be tamper-resistant; and,
- the client devices can serve as user nodes in a P2P network that also contains other user, vendor, and bank nodes.

Chapter 8

Conclusions and Future Work

8.1 Conclusion

Various types of P2P systems have been gaining popularity for many applications. P2P systems have architectures that can be classified as unstructured, structured, or non-forwarding. As with any distributed system, P2P protocols are subject to various forms of attacks including denial-of-service attacks. In this dissertation, we have identified that application-layer denial-of-service attacks can be quite damaging to P2P networks.

These attacks can be prevented, detected, contained, and recovered from using a variety of mechanisms. Preventative techniques may not be perfect, and may fail at preventing some malicious nodes from joining a P2P system. Indeed, in an Internet-scale system, there are always a few malicious nodes in the system, and the presence of malicious nodes needs to be treated as a fact-of-life, instead of as an error or exceptional condition. It may take some time to identify and/or detect malicious nodes before they can be disconnected from the network. In the interim time between which malicious nodes join, and are detected and disconnected, we would like to contain the amount of damage that they can cause to the normal operation of the network.

To highlight the importance of attack containment techniques, consider an analogy between the defenses of a P2P system, and national security defenses. On the morning

of September 11, 2001, at the time that the first hijacked airplane hit the North Tower of the World Trade Center, our nation's preventative defense mechanisms had already failed. The FBI, CIA, NSA, and INS had failed to identify and/or detain the terrorists that had entered the country and had been training to fly commercial airliners. The hijackers were let through the airport security check points and were allowed to board with knives on-hand. The first airplane already hit the tower, and hijackers were already in control of two other planes in the air.

After the first airplane hit the North Tower, it was, in fact, unclear as to whether or not what had just happened was an accident, or whether or not it was an atttack. Indeed, it would take the authorities some time to detect exactly what was going on. And, of course, regardless of whether or not the incident that had just occurred was an attack, it would take quite some time to recover from the situation, to the extent that such incidents can be recovered from. Immediately after the crash of the first airplane, and while the authorities were in the process of detecting exactly what was going on, it is clear that all possible efforts should have been focused on containing the effects of the incident, by saving as many lives as possible. Such containment techniques– whether they be protocols that emergency response teams should follow, the activation of additional secure radio frequencies and communication channels for use by authorities to coordinate life-saving efforts, or possible procedures for emergency scrambling of jet fighters– need to be designed, practiced, tested, and put in place well-ahead of any such incident.

In a P2P system, it is also important that once malicious nodes have breached whatever preventative mechanisms have been deployed, and while the existence, locations, and identities of the malicious nodes are in the process of being detected, attack containment techniques should be used to minimize the impact of the attack while detection and recovery algorithms are executing.

In the first part of this dissertation, we have developed and evaluated techniques that contain appliation-layer denial-of-service attacks in each of the major types of P2P networks (unstructured, structured, and non-forwarding). For unstructured and structured systems, we developed a traffic model that captured the key characteristics of how queries flow through the system. We designed various traffic management

policies that contained the effects of malicious nodes issuing a potentially large number of "useless" queries. We also defined metrics by which those traffic management policies could be evaluated and we used those metrics to build an understanding of the trade-offs between using various policies. In non-forwarding networks, we identified that the number of occurences of malicious node ids in the host (pong) caches of good nodes was the key factor in the success of an application-layer DoS attack, and we developed and evaluated an ID smearing algorithm and dynamic network partitioning scheme to mitigate the "poisoning" of caches.

It is important that the application-layer mitigation techniques that we developed are used in tandem with prevention, detection, and recovery algorithms. After preventative techniques may have been circumvented, and while a system is under attack, the techniques that we have developed can be used to provide users with a degraded quality of service while detection algorithms are concurrently being used to identify offending nodes and disconnect them from the network. By using a combination of prevention, detection, containment, and recovery algorithms concurrently, future P2P systems will not have "turtle shell" architectures in which if just a few malicious nodes get through an outer shell of defense, they can easily attack a soft inner core. Given the level of distribution, decentralization, and node autonomy in P2P systems, containment techniques protect P2P nodes even when some percentage of the nodes in the network are malicious.

In the second part of this dissertation, we turned our attention to how electronic commerce might be supported in P2P systems. In a P2P system, nodes may serve as both consumers and vendors. For example, a node operated by an enterprise might serve as a vendor for a document editing service, and might be a purchaser of spelling, grammar checking, and formatting services that are used together to provide the document editing service. We proposed that P2P nodes use symmetric digital wallets that can function as consumers or vendors during different points of their operation (or even at the same time) to execute e-commerce transactions. We propose an architecture for symmetric digital wallets, and also identify extensibility, device independence, and client-initiation as important architectural goals for digital wallets. Finally, we implemented a digital wallet that satisfied these goals, and built an efficient

digital cash system for micropayments based on our digital wallet architecture.

8.2 Future Work

In this dissertation, we have taken a first step in developing and evaluating containment techniques for application-layer denial-of-service attacks in P2P networks. However, there is much future work that can be done in the areas of attack prevention, detection, and recovery. We describe some potential directions for work on these areas, as well as some further areas in which attack containment should be explored.

- *Prevention.* In most public P2P systems, such as Gnutella and eDonkey, any host with an IP address and that can speak the protocol can join the network. As a result, it is easy for a malicious node to join the network since obtaining an IP address and obtaining source code that speaks the protocol is very easy. Organizations such as the RIAA and MPAA have taken advantage of how easily accessible such systems are to build "crawlers" that identify potential copyright violators, serve inauthentic content, and conduct various other types of mischief. Because there are virtually no preventative mechanisms deployed for public P2P networks, they may be very vulnerable to attacks.

  On the other hand, private and proprietary P2P systems such as Groove [78] can be configured to require each peer to have a certificate and authenticate itself before being allowed to join the network. Clearly, this approach is heavy-handed due to enterprise security requirements, and while unauthorized nodes might be prevented from joining such a network, enterprises are not as easily able to collaborate with outsiders or access content on public P2P networks.

  It might be reasonable to explore a middle ground between not deploying any preventative measures, and deploying public key infrstructure (PKI) that enables strong authentication to prevent unauthorized parties from joining a P2P network. One potential future direction might involve using *weak authentication* as a preventative measure for a P2P network. One way of implementing weak authentication might be to provide pseudonyms to nodes.

A public key certificate used as part of a heavy-handed public key infrastructure (PKI) may contain a user's full name address, and other identifying information. Such public key certificates can be used as part of a "strong" authentication required before a peer can join a P2P network. A pseudonym-based approach would involve assigning "ficticious identities" (or pseudonyms) to users that can be used to log in to a P2P network. Although such an identity is ficticious, what is important is that the user is given some identity to use when participating in the network, and that there be some well-defined and secure process to assign ficticious identities to legitimate users. Illegitimate or malicious users should not be able to easily obtain a pseudonym, and hence should not be able to easily participate in the system. Such an approach could be deployed as a preventative measure that does not require "strong" authentication. Some of the key open issues in designing a pseudonym-based approach range from how to implement pseudonyms (i.e. what state does a peer need to store? how does a peer obtain that state to begin with? how is the state used to authentication? how is revocation handled?), understanding the level of security that pseudonyms can provide, and comparing the performance of such a scheme to a full PKI.

Instead of attempting to prevent malicious nodes from joining the network, another approach might be to make it unproductive or reduce the incentive for malicious nodes to submit queries. For instance, one could require a node to solve a cryptographic puzzle before processing its query (as briefly mentioned in Section 3.4). The intuition behind this approach is that denial-of-service attacks are possible when clients are able to request a lot of work from the network without doing any work themselves. If we require that a client must solve a cryptographic puzzle before submitting a query, we are artificially requiring the client to prove that it is worth its time and computational resources to execute the query. An even better approach might be to develop a mechanism that has a node execute queries on behalf of the network if the network executes queries on behalf of the node. The general idea behind this approach would be to provide nodes incentives to execute queries on behalf of other nodes. Reference [195] takes a first step at studying some potential incentive mechanisms that nodes

in a Gnutella network could use.

- *Detection.* In Chapter 5, we studied how our attack containment techniques could benefit from malicious node detectors (MNDs). It is reasonable to guess that the traffic management policies we studied in Chapters 2, 3, and 4 may also be able to benefit from MNDs. At the very least, while nodes might usually use traffic management policies that optimize for performance, if MNDs report that the system might be under attack, nodes could switch to using policies that are more aggressive at containing attacks at the expense of application performance.

  While some approaches towards building implementations of MNDs have been studied [91, 124], there is much more work to be done in that direction. In addition, studying how containment and detection techniques can interact and benefit from each other is an open area for further investigation.

- *Containment.* While a central focus of this dissertation has been on containment of application-layer DoS attacks, we see this work as a first step. There are a number of future directions in which it would be worthwhile to "dig even deeper." We list a few below.

  - *Colluding malicious nodes.* While we did study colluding malicious nodes in non-forwarding P2P networks in Chapter 5, we assumed that malicious nodes act in isolation, for the most part, in our studies of unstructured and structured P2P systems in Chapters 2, 3, and 4.

    For instance, one way in which malicious nodes may collude in Gnutella is as follows: if good nodes use a LQF policy, then malicious nodes can optimize their attack by making sure that no two malicious nodes connect to a good node. Also, if a PA policy is used by good nodes, then malicious nodes could collude to coordinate their attack in such a way as to further maximize the probability that their queries will be selected for processing instead of a good node's queries. To protect against attacks in which malicious nodes collude, one would need to consider how malicious nodes

would choose how many queries to send to maximize the effect of their attack if more than one of them connects to a good node. We would then need to evaluate how our IASes stand up to such attacks, and determine if they need to be modified to deal with colluding malicious nodes.

– *Other attack models.* In the attack models that we studied for Gnutella and Chord, we had malicious nodes generate lots of queries with the intention of having an overall negative impact on remote work throughput. Also, in GUESS, malicious nodes posed a threat where their goal was to fragment the network or provide inauthentic results.

  However, one can imagine an attack model in which particular "important" good nodes are targeted instead of having malicious nodes generally try to disrupt the operation of the network. For instance, if a particular good node publishes a document that some government might like to censor, we might want to study what kinds of attacks malicious nodes can employ to censor such nodes and/or documents. To defend against such attacks, we could study what new types of policies might need to be developed to combat censorship or attacks against particular nodes.

  Also, while flooding a system with queries might be one possible behavior for a malicious node, another might be to attempt to flood the system with query-hits. For example, a malicious node could inject just a few queries that will have so many results that nodes may become overloaded forwarding query-hit messages.

– *DoS in Wireless Ad-hoc Networks.* Wireless ad-hoc networks have many similarities with Gnutella-like P2P networks because radio broadcasts are similar to flooding mechanisms. Our traffic management solutions for attack containment might also carry over the the world of wireless ad-hoc networks. However, one assumption that we made in our work on unstrucutred networks is that good nodes at least have the capacity to look at all of the queries that are arriving on their links. Nodes in a wireless ad-hoc network must expend battery power and energy even just to look

at queries that are arriving. If too many queries arrive in a particular time interval, a wireless ad-hoc node may drain its battery even just looking at all of the queries arriving. As such, it may make sense to model the act of looking at queries explicitly, and determining how we might modify our traffic management policies (perhaps by introducing polling) to contain DoS attacks in wireless ad-hoc networks as well.

- *Recovery.* Once malicious nodes have been detected, they need to be disconnected. However, the network may have been expecting that the malicious node should perform particular actions, such as routing. With the absence of the malicious node, the network may need to find a good node to take its place and re-configure itself by updating routing entries at other nodes. Malicous nodes that are disconnected can simply be modeled as regular nodes leaving the network, but in the case that a network is recovering from a significant attack that, say, resulted in network fragmentation, specialized re-construction algorithms could be employed. In addition, if frequent disconnections take place, or if the detection algortihms have a high rate of false positives, then the churn may upset the stability of the network. In any case, future work on specialized recovery algorithms may be warranted.

P2P protocols are likely to become more prevalent as more applications of them are discovered. As with any infrastructural technology, once P2P protocols are relied upon for more applications, it will be imperative that they are designed to be resilient to attacks. While computer security professionals and cryptographers have traditionally focused on preventative mechanisms for security, we believe that detection, containment, and recovery techniques are also important as Internet-scale systems may always suffer from some breaches of preventative mechanisms. We have developed and illustrated how attack containment techniques can work in P2P systems in this disseration, and we have outlined some future directions that can be taken to further study how to prevent, detect, and recover from such attacks.

In this dissertation, we have studied the containment of DoS attacks in various P2P networks. The solutions that we have developed and evaluated can be deployed together with other mechanisms that prevent, detect, and recover from DoS and other types of attacks. Once a P2P network has been protected from attacks, it can be used as the basis for commerce between its peers. We have thus also proposed an architecture for commerce in P2P networks, and described a prototypical implementation of it. The work we have described here takes a step towards providing a solid foundation for security and commerce within P2P systems. In addition, we expect that our work will motivate further research that can eventually result in P2P systems becoming as fundamentally disruptive a technology as the Internet itself.

Appendix A

Graph Topologies

Our evaluations were run on small networks of either 14 nodes (for complete, cycle, wheel, line, and star topologies) or 16 nodes (for grid and power-law topologies). The following diagrams are examples of structures for complete, cycle, wheel, line, star, grid topologies. The particular instance of a power-law topology that we used in our evaluation is shown here as well.

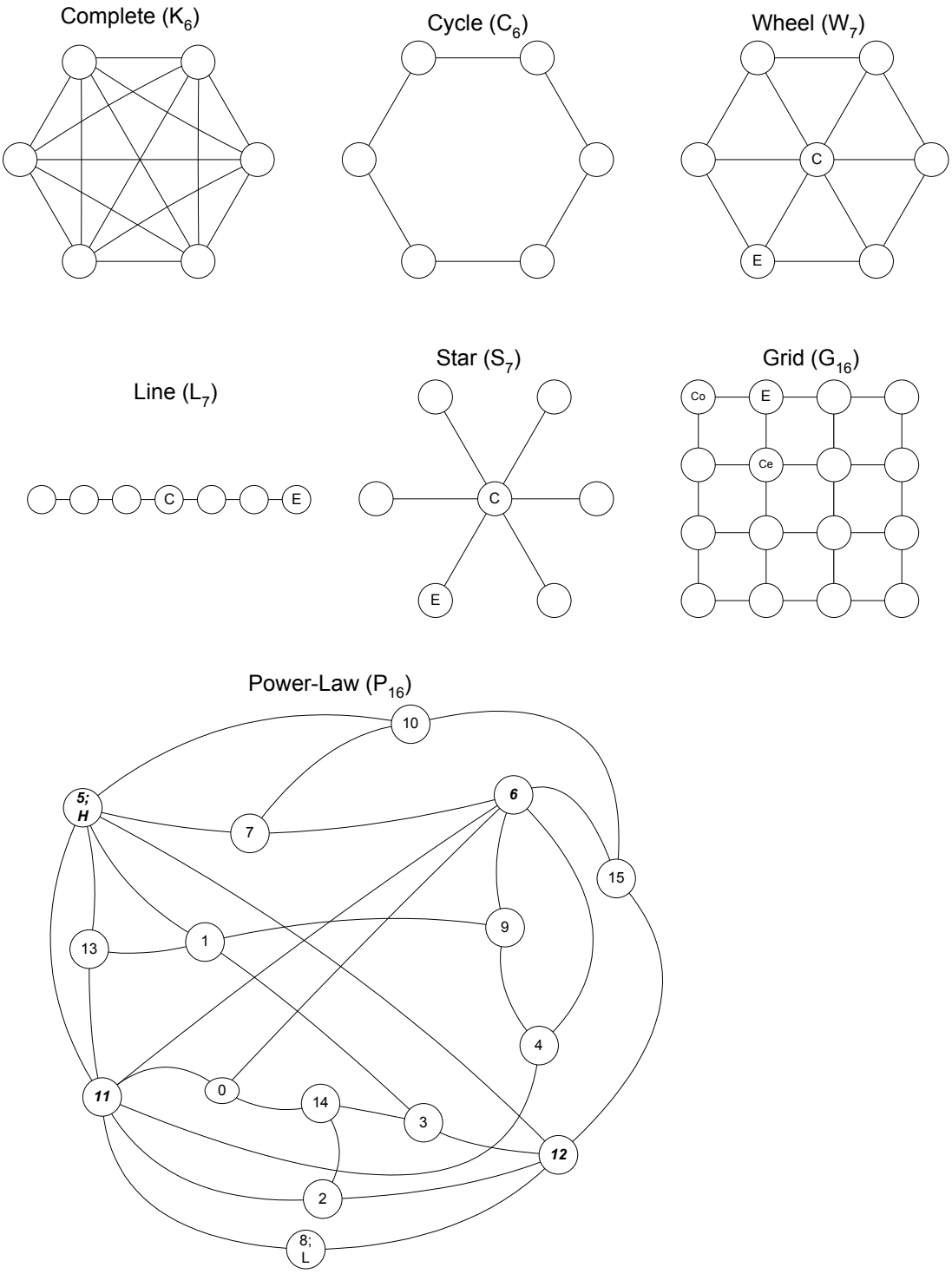

Figure A.1: Graph Topologies

Appendix B

Inevitability Proof

In this appendix, we consider a limited version of the GUESS protocol and formally prove that it is possible for a single malicious node to "poison" the pong caches of all other nodes if certain conditions hold. In other words, under certain conditions, it is inevitable that all cache entries in the system can be poisoned.

Section 5.4.5 informally argues that inevitable cache poisoning can occur even with much less stringent conditions than those discussed here. As a result, system designers must be very careful when designing GUESS-like resource discovery mechanisms for distributed systems in which the presence of malicious nodes is expected.

Definition B.0.1 *Poisoned Cache. A cache $P(N_i, t)$ is said to be poisoned at time t if and only if there exists a cache entry $m \in P(N_i, t)$ such that $m \in M(t)$.*

To illustrate what can go wrong even in a very simple GUESS network, consider a simple scenario in which the following three conditions hold:

(1) *Malicious Node.* There exists a single malicious node $N_m, 0 \leq m < n$. N_m has its own id in its pong cache, $P(N_m, 0) = \{N_m\}$. Node N_m always responds with $S = \{N_m\}$, and always chooses $X = Y = \emptyset$ when updating its pong cache (that is, it never updates it!);

(2) *Good Nodes.* All other nodes are good. Good nodes always choose $S = P(N_i, t)$ when responding to queries and choose $X = P(N_i, t)$ and $Y = S$ when updating their pong cache. In other words, good nodes share the entire contents of their pong cache

with other nodes, and always "trustingly" update the entire contents of their own pong cache with those received from other nodes; and

(3) *Directed Cycle Topology.* $\forall i, P(N_i, 0) = \{N_{(i-1) \bmod n}\}$ for each good node N_i. That is, initially, each good node has a single "neighbor" in its pong cache such that every node is included in exactly one other node's cache.

If conditions (1), (2), and (3) above hold, then at $t = \lceil \log_2 n \rceil$, $\forall i, P(N_i, t) = \{N_0\}$. That is, all pong caches will be poisoned when $t = \lceil \log_2 n \rceil$.

We show that the pong caches of all the nodes will inevitably be poisoned with a single entry containing the node id of the malicious node if these conditions hold.

The process by which pong caches are poisoned given the conditions in this theorem is similar to that of "pointer jumping" as described in parallel computing literature. We first prove a useful lemma before proving the inevitability theorem itself.

In the following proofs, we assume, without loss of generality, that $m = N_0$. That is, N_0 is the malicious node.

Lemma B.0.1 *"Pointer Jumping Lemma"* $P(N_i, k) = \{N_{max(i-2^k,0)}\}, 1 \leq i < n$

Proof

Base case. For $k = 0$, $P(N_i, 0) = \{N_{max(i-2^0,0)}\} = \{N_{max(i-1,0)}\}$ is true by conditions (1) and (3).

Inductive Step. We assume that $P(N_i, k) = \{N_{max(i-2^k,0)}\}, 1 \leq i < n$, and we show that $P(N_i, k+1) = \{N_{max(i-2^{k+1},0)}\}, 1 \leq i < n$. If $i = 0$, our inductive hypothesis tells us that $P(N_0, k) = \{N_0\}$. Since N_0 is malicious, it does not update its pong cache as per condition (1), and $P(N_0, k+1) = \{N_0\}$, so we have shown the inductive step holds for $i = 0$.

For $i \geq 1$, node N_i queries the node specified by its pong cache, $P(N_i, k)$. That is, N_i queries node $N_{max(i-2^k,0)}$ by our inductive hypothesis. It receives $S = P(N_{max(i-2^k,0)}, k)$. Again, by our inductive hypothesis, $S = P(N_{max(i-2^k,0)}, k) = \{N_{max(max(i-2^k,0)-2^k,0)}\}$. If $i - 2^k > 0$, then $S = \{N_{max(i-2^k-2^k,0)}\} = \{N_{max(i-2*2^k,0)}\} = \{N_{max(i-2^{k+1},0)}\}$. If $i - 2^k < 0$, then $S = \{N_{max(0-2^k,0)}\} = \{N_0\}$. However, if $i - 2^k < 0$, then surely $i - 2^{k+1} < 0$. Hence, in either case, node N_i sets $P(N_i, k+1) = \{N_{max(i-2^{k+1},0)}\}, 1 \leq i < n$. This completes the proof.

Now that we have established the "pointer jumping" lemma, we prove the inevitability result.

Theorem B.0.2 *"Inevitable poisoning of pong caches."*

If conditions (1), (2), and (3) above hold, then at $t = \lceil \log_2 n \rceil$, $\forall i, P(N_i, t) = \{N_0\}$. *That is, all pong caches will be poisoned when* $t = \lceil \log_2 n \rceil$.

Proof

Let M be the set of nodes N_i for which $P(N_i, t) = \{N_0\}$.

After the kth time step, $M = \{N_0, N_1, ..., N_{min(2^k, n)}\}$. After $\lceil \log_2 n \rceil$ time steps, all the nodes in the graph are in the set M.

Base Case. Initially, $M = \{N_0, N_1\}$. $N_0 \in M$ by condition (1), and $N_1 \in M$ by condition (3).

Inductive Step. Assume that after the kth time step, $M = \{N_0, N_1, ..., N_{min(2^k, n)}\}$. We will show that after the $k + 1$st time step, $M = \{N_0, N_1, ..., N_{min(2^{k+1}, n)}\}$.

From the inductive hypothesis, we know that $P(N_i, k) = \{N_0\}, 0 \leq i \leq min(2^k, n)$. We also know that $P(N_i, k) = \{N_{i-2^k}\}, 2^k + 1 \leq i \leq min(2^{k+1}, n)$, due to the "pointer jumping lemma."

Hence, each $P(N_i, k + 1) = P(N_{i-2^k}) = \{N_0\}, 2^k + 1 \leq i \leq min(2^{k+1}, n)$, and $M = \{N_0, N_1, ..., N_{min(2^{k+1}, n)}\}$. This completes the proof.

Bibliography

[1] 3com corporation. http://www.3com.com.

[2] J.L. Abad-Peiro, N. Asokan, M. Steiner, and M. Waidner. Designing a generic payment service. IBM Systems Journal Vol. 37 No. 1, 1998.

[3] H. Abelson, R. Anderson, S. Bellovin, J. Benaloh, M. Blaze, W. Die, J. Gilmore, P. Neumann, R. Rivest, J. Schiller, and B. Schneier. The risks of key recovery, key escrow, trusted third party and encryption. Digital Issues, No. 3, 1998, pp. 1-18., 1998.

[4] Edward Amoroso. A policy model for denial of service. In Proc. Computer Security Foundations Workshop III, pages 110–114, Franconia, NH USA, June 1990. IEEE Computer Society Press.

[5] R. Anderson and M. Kuhn. Tamper Resistance - a Cautionary Note. In *Proceedings of the Second Usenix Workshop on Electronic Commerce*, pages 1–11, November 1996.

[6] R. Anderson, C. Manifavas, and C. Sutherland. Netcard – a practical electronic cash system.

[7] Hector Garcia-Molina Arturo Crespo. Routing indices for peer-to-peer systems. In *ICDCS*, 2002.

[8] Auditing issues in secure database management systems. National Computer Security Center Technical Report-005, Volume 4/5, May 1996.

[9] Alireza Bahreman and Rajkumar Narayanaswamy. Payment method negotiation service. In *The Second USENIX Workshop on Electronic Commerce*, 1996.

[10] Iris: Infrastructure for resilient internet systems. http://iris.lcs.mit.edu/.

[11] D. Balenson, C.M. Ellison, S.B. Lipner, and S.T. Walker. A new approach to software key escrow encryption. TIS Report 520, Trusted Information Systems, Aug 1994.

[12] Mihir Bellare and Shafi Goldwasser. Verifiable partial key escrow. In *ACM Conference on Computer and Communications Security*, pages 78–91, 1997.

[13] Icmp traceback messages. http://www.silicondefense.com/ research/ itrex/ archive/ tracing-papers/ draft-bellovin-itrace-00.txt.

[14] S. Bellovin, D. Firewalls, and l November. Special issue on security. ;login: November 1999, Special Issue on Security, ISSN 1044-6397, Also at: http://www.usenix.org/, 1999.

[15] Hector Garcia-Molina Beverly Yang, Patrick Vinograd. Evaluating guess and non-forwarding peer-to-peer search. In *24th International Conference on Distributed Computing Systems (ICDCS 2004), Tokyo, Japan*, 2004.

[16] P. Biddle, P. England, M. Peinado, and B. Willman. The darknet and the future of content distribution. http://crypto.stanford.edu/DRM2002/darknet5.doc.

[17] M. Bishop, C. Wee, and J. Frank. Goal oriented auditing and logging. Submitted to IEEE Transactions on Computing Systems, 1996.

[18] Matt Bishop. A standard audit trail format. Technical report, University of California at Davis, 1995.

[19] Blaze. Oblivious key escrow. In *IWIH: International Workshop on Information Hiding*, 1996.

[20] Personal communication with dan boneh.

[21] Andrew Brown. Java commerce messages white paper. http://java.sun.com/products/commerce/docs/whitepapers/jcm_whitepaper/jcm.pdf.

[22] John Kelsey Bruce Schneier. Secure audit logs to support computer forensics. TISSEC 2(2): 159-176 (1999).

[23] M. Yung C. Jutla. Paytree: amortized signature for flexible micropayments. In *In 2nd USENIX Workshop on electronic commerce*, pages 213–221, 1996.

[24] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS, 1999.

[25] M. Castro and B. Liskov. Authenticated byzantine fault tolerance without public-key cryptography. Technical Report MIT/LCS/TM-595, 1999.

[26] Miguel Castro, Peter Druschel, Ayalvadi Ganesh, Antony Rowstron, and Dan S. Wallach. Security for peer-to-peer routing overlays. In *Fifth Symposium on Operating Systems Design and Implementation (OSDI '02) (Boston, Massachusetts)*, 2002.

[27] Cert advisory ca-2000-01 denial-of-service developments. http://www.cert.org/ advisories/ CA-2000-01.html, January 2000.

[28] A. D. Chambers. Current strategies for computer auditing within an organis ation. The Computer Journal 24(4): 290-294 (1981).

[29] W. Cheswick and S. Bellovin. Firewalls and internet security: Repelling the wily hacker. Addison-Wesley, 1994.

[30] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[31] E*trade, zdnet latest targets in wave of cyber-attacks. http://www.cnn.com/2000/TECH/computing/02/09/cyber.attacks.02/.

[32] Coffey and Saidha. Non-repudiation with mandatory proof of receipt. *ACM-CCR: Computer Communication Review*, 26, 1996.

[33] Brian Cooper and Hector Garcia-Molina. Peer to peer data trading to preserve information. In *ACM Transactions on Information Systems, 2002*, 2002.

[34] B. Cox, D. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *First USENIX Workshop of Electronic Commerce Proceedings*, 1995.

[35] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol.

[36] Arturo Crespo and Hector Garcia-Molina. Routing indexes for peer-to-peer systems. Technical report, Stanford University, Computer Science Department, 2001.

[37] Cybercash. http://www.cybercash.com.

[38] David Ratajczak Dahlia Malkhi, Moni Naor. Viceroy: A scalable and dynamic lookup network. Submitted for publication, 2001.

[39] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A reputation-based approach for choosing reliable resources in peer-to-peer networks. In *Proc. of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, USA, November 2002.

[40] N. Daswani, D. Boneh, H. Garcia-Molina, S. Ketchpel, and A. Paepcke. Swaperoo: A simple wallet architecture for payments, exchanges, refunds, and other operations, 1998.

[41] Neil Daswani and Hector Garcia-Molina. Query-flood dos attacks in gnutella networks. In *ACM Conference on Computer and Communications Security*, 2002.

[42] Neil Daswani, Hector Garcia-Molina, and Beverly Yang. Open problems in data-sharing peer-to-peer systems. In *Database Theory - ICDT 2003, 9th International Conference, Siena, Italy, January 8-10, 2003, Proceedings*, volume 2572 of *Lecture Notes in Computer Science*. Springer, 2002.

[43] S. Daswani. Personal communication. .

[44] S. Daswani and A. Fisk. Guess protocol specification. http://groups.yahoo.com/ group/ the_gdf/ files/ Proposals/ GUESS/ guess_01.txt.

[45] Davies. Applying the RSA digital signature to electronic mail. *COMPUTER: IEEE Computer*, 1983.

[46] D. Davies and W. Price. Digital signatures — an update. In International Conference on Computer Security, pages 845–849, 1984.

[47] D. W. Davies and W. L. Price. The application of digital signatures based on public-key cryptosystems. In *Proc. Fifth Intl. Computer Communications Conference*, pages 525–530, 1980.

[48] Donald Davies. Quality auditing: The necessary step towards the required q uality objectives. CAiSE 1990: 286.

[49] Herve Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[50] D. Denning. An intrusion-detection model. 1986 IEEE Computer Society Symposium on Research in Security and Privacy, pp 118-31, 1986.

[51] D. Denning. Critical factors of key escrow encryption systems. Proceedings of the 18th National Information Systems Conference, 10-13 October 1995, Baltimore, Maryland, pp384-394., 1995.

[52] Dorothy E. Denning and Dennis K. Branstad. A taxonomy for key escrow encryption systems. *Communications of the ACM*, 39(3):34–40, 1996.

[53] Tim Dierks and Eric Rescorla. The tls protocol, version 1.1. http://www.ietf.org/internet-drafts/draft-ietf-tls-rfc2246-bis-07.txt.

[54] Digicash. http://www.digicash.com.

[55] R. Dingledine, M. Freedman, and D. Molnar. The free haven project: distributed anonymous storage service. Proceedings of the Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, USA. Springer: New York (2001)., 2001.

[56] Roger Dingledine, Michael J. Freedman, David Hopwood, and David Molnar. A reputation system to increase MIX-net reliability. *Lecture Notes in Computer Science*, 2137:126+, 2001.

[57] Roger Dingledine, Michael J. Freedman, and David Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 67–95, 2000.

[58] Danny Dolev, Michael J. Fischer, Rob Fowler, Nancy A. Lynch, and H. Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

[59] J. Douceur. The sybil attack. In *IPTPS 2002*.

[60] D. Eastlake. Universal payment preamble specification. http://www.w3.org/ECommerce/specs/upp.txt.

[61] The e-bay home page. http://www.ebay.com.

[62] Paul Feldman and Silvio Micali. Optimal algorithms for byzantine agreement. In *ACM Symposium on Theory of Computing*, pages 148–161, 1988.

[63] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ ip source address spoofing. In IETF RFC 2267, 1998., 1998.

[64] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[65] Financial applications for the palmpilot. http://www.pilotzone.com/palm.html.

[66] Michael J. Freedman and Robert Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002)*, Washington, D.C., November 2002.

[67] A. Freier, P. Karlton, and P. Kocher. The ssl protocol. http://wp.netscape.com/eng/ssl3/draft302.txt.

[68] Zvi Galil, Alain J. Mayer, and Moti Yung. Resolving message complexity of byzantine agreement and beyond. In *IEEE Symposium on Foundations of Computer Science*, pages 724–733, 1995.

[69] J. Garay and Y. Moses. Fully polynomial byzantine agreement in t +1 rounds. In Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, pages 31–41, San Diego, California, May 1993.

[70] Lee Garber. Denial-of-service attacks rip the internet. Computer, pages 12-17, April 2000., 2000.

[71] Hector Garcia-Molina, Frank Pitelli, and Susan B. Davidson. Applications of byzantine agreement in database systems. *Database Systems*, 11(1):27–47, 1986.

[72] A. Glass. Could the smartcard be dumb. In *Proceedings of Eurocrypt '86*, 1986.

[73] Gnutella development forum (gdf). http://groups.yahoo.com/ group/ the_gdf/.

[74] Gnutella protocol specification. http://www9.limewire.com/ developer/ gnutella_protocol_0.4.pdf.

[75] I. Goldberg. pilotssleay-2.01. http://www.isaac.cs.berkeley.edu/pilot/.

[76] T. Goldstein. The gateway security model in the java electronic commerce framework. In *Proceedings of the Financial Cryptography First International Conference*, 1997.

[77] J. Gray. A comparison of the byzantine agreement problem and the transaction commit problem. Fault-Tolerant Distributed Computing, LNCS 448, Springer Verlag, 1987.

[78] Groove networks home page. http://www.groove.net/.

[79] Krishna Gummadi, Ramakrishna Gummadi, Steve Gribble, Sylvia Ratnasamy, Scott Shenker, and Ion Stoica. The impact of dht routing geometry on resilience and proximity. In *Proceedings of ACM SIGCOMM*, 2003.

[80] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In *Symposium on Principles of Programming Languages*, pages 316–329, 2000.

[81] James V. Hansen. Audit considerations in distributed processing systems. Communications of the ACM 26(8): 562-569 (1983).

[82] Ann Harrison. The denial-of-service aftermath. http://www.cnn.com/2000/TECH/computing/02/14/dos.aftermath.idg/.

[83] R. Hauser, M. Steiner, and M. Waidner. Micro-payments based on ikp. In *In 14th Wolrd-wide congress on computer and communication security protection*, 1996.

[84] Raquel Hill, Adon Hwang, and David Molnar. Approaches to mixnets.

[85] G. Hunt, G. Goldszmidt, R. King, and R. Mukherjee. Network dispatcher: a connection router for scalable internet services. In *Proceedings of WWW7*.

[86] E.A. Hyden. Operating system support for quality of service. Ph.D. Thesis, University of Cambridge, 1994.

[87] Sushil Jajodia, Shashi K. Gadia, Gautam Bhargava, and Edgar H. Sibley. Audit trail organization in relational databases. DBSec 1989: 269-281.

[88] S. Jarecki and A. Odlyzko. An efficient micropayment system based on probabilistic polling. In *Financial Cryptography*, 1997.

[89] Sun microsystems. java commerce home page. http://java.sun.com/commerce/.

[90] J. P. Melanson K. F. Seiden. The auditing facility for a vmm security kernel. IEEE Symposium on Security and Privacy 1990: 262-277.

[91] Sepandar Kamvar, Mario Schlosser, and Hector Garcia-Molina. Eigenrep: Reputation management in p2p systems. Preprint.

[92] David R. Karger and Matthias Ruhl. Simple efficient load balancing algorithms for peer-to-peer systems. In *Proceedings ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2004.

[93] Kazaa home page. http://www.kazaa.com/.

[94] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. Secure overlay services. In *ACM SIGCOMM 2002*.

[95] Steven Ketchpel, Hector Garcia-Molina, Andreas Paepcke, Scott Hassan, and Steve Cousins. Upai: A universal payment application interface. In *USENIX 2nd Electronic Commerce Workshop*, 1996.

[96] Steven P. Ketchpel, Hector Garcia-Molina, and Andreas Paepcke. Shopping models: A flexible architecture for information commerce. In *Fourth Annual Conference on the Theory and Practice of Digital Libraries*, 1997.

[97] J. Kilian and T. Leighton. Failsafe key escrow. In: Coppersmith, D. (ed): Advances in Cryptology – Crypto '95, Proceedings (Lecture Notes in Computer Science 963). Springer-Verlag (1995) 208–221, 1995.

[98] L. Knudsen and T. Pedersen. the dif- culty of software key escrow. In Advances in Cryptology - EUROCRYPT'96, Lecture Notes in Computer Science, pages 1-8. Springer-Verlang, 1996.

[99] P. Kocher. On certificate revocation and validation. In Proc. International Conference on Financial Cryptography, volume 1465 of Lecture Notes in Computer Science, 1998.

[100] H. Kopetz. Temporal firewalls. DeVa 1st Selective Open Workshop, Schloss Reisensburg, Sept. 1996.

[101] Sandeep Kumar. *Classification and Detection of Computer Intrusions*. PhD thesis, Purdue, IN, 1995.

[102] L. Lamport. Password authentication with insecure communication. In *Communications of the ACM, Vol. 24 (11)*, pages 770–771, 1981.

[103] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. ACM Transactions on Programming Languages and Systems, Vol.4, No.3, July 1982, pp. 382-401.

[104] Leslie Lamport. The weak byzantine generals problem. *Journal of the ACM*, 30(3):668–676, 1983.

[105] Leslie Lamport and Michael J. Fischer. Byzantine generals and transactions commit protocols. Technical Report Opus 62, Menlo Park, California, 1982.

[106] Wenke Lee and Salvatore Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[107] Wenke Lee, Salvatore Stolfo, and Patrick Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI methods in Fraud and risk management*, 1997.

[108] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. Mining audit data to build intrusion detection models. In *Knowledge Discovery and Data Mining*, pages 66–72, 1998.

[109] Wenke Lee, Salvatore J. Stolfo, and Kui W. Mok. Adaptive intrusion detection: A data mining approach. *Artificial Intelligence Review*, 14(6):533–567, 2000.

[110] Arjen K. Lenstra, Peter Winkler, and Yacov Yacobi. A key escrow system with warrant bounds. In *CRYPTO*, number Theory, pages 197–207, 1995.

[111] R. Lethin. Reputation chapter in peer-to-peer: Harnessing the power of disruptive technologies. Peer-to-Peer: Harnessing the Power of Disruptive Technologies ed. Andy Oram, O'Reilly and Associates, March 2001.

[112] Limewire home page. http://www.limewire.com/.

[113] Teresa F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, 1988.

[114] Teresa F Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, 1993.

[115] C. LV, P. CAO, E. COHEN, K. LI, and S. SHENKER. Search and replication in unstructured peer-to-peer networks. http://citeseer.nj.nec.com/lv02search.html, 2001.

[116] N. A. Lynch. Distributed algorithms. Morgan Kaufmann, 1996.

[117] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. In *ACM Symposium on Theory of Computing*, pages 569–578, 1997.

[118] M. Manasse. The millicent protocols for electronic commerce. In *Proc. of the 1st USENIX workshop on Electronic Commerce.*

[119] Petros Maniatis and Mary Baker. Secure History Preservation Through Timeline Entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA, August 2002.

[120] G. Manku. Balanced binary trees for id management and load balance in distributed hash tables. In *Proc. 23rd ACM Symposium on Principles of Distributed Computing (PODC 2004)*, pages 197–205, Newfoundland, Canada, 2004.

[121] Aviel D. Rubin Marc Waldman and Lorrie Faith Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, pages 59–72, August 2000.

[122] Olivier Markowitch and Steve Kremer. A multi-party optimistic non-repudiation protocol. In *Information Security and Cryptology*, pages 109–122, 2000.

[123] Olivier Markowitch and Yves Roggeman. Probabilistic non-repudiation without trusted third party. In *Second Workshop on Security in Communication Network 99*, 1999.

[124] Sergio Marti and Hector Garcia-Molina. Limited reputation sharing in p2p systems. In *ACM Conference on Electronic Commerce*, May 2004.

[125] Stephen M. Matyas. Digital signatures — an overview. *Computer Networks: The International Journal of Distributed Informatique*, 3(2):87–94, 1979.

[126] Patrick McDaniel and Sugih Jamin. Windowed certificate revocation. In *INFOCOM (3)*, pages 1406–1414, 2000.

[127] R. Merkle. A certified digital signature. Advances in Cryptology—CRYPTO '89, volume 435 of Lecture Notes in Computer Science, pages 218–238. Springer-Verlag, 1990.

[128] S. Micali. Efficient certificate revocation. Technical Report MIT/LCS/TM-542b, 1996.

[129] Silvio Micali and Ronald L. Rivest. Micropayments revisited. In *CT-RSA*, pages 149–163, 2002.

[130] Microsoft wallet home page. http://www.microsoft.com/wallet.

[131] C. Mitchell, F. Piper, and P. Wild. Digital signatures. Contemporary Cryptology, G. Simmons (Ed.), IEEE Press, 1992, pp. 325-378.

[132] C. Mohan, R. Strong, and S. Finkelstein. Methods for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing, 1987.

[133] http://www.mojonation.net/ docs/ technical_overview.shtml.

[134] Mojo nation technical overview home page. http://www.mojonation.net/ docs/ technical_overview.shtml.

[135] Mondex international. http://www.mondex.com.

[136] David Moore, Geoffrey Voelker, and Stefan Savage. Inferring internet denial of service activity. In *Proceedings of the 2001 USENIX Security Symposium, Washington D.C., August 2001*, 2001.

[137] Morpheus home page. http://www.musiccity.com.

[138] Biswanath Mukherjee, L. Todd Heberlein, and Karl N. Levitt. Network intrusion detection. *IEEE Network*, 8(3):26–41, / 1994.

[139] B. Yang. N. Daswani, H. Garcia-Molina. Open problems in data-sharing peer-to-peer systems. In *International Conference on Database Theory. Siena, Italy.*, 2003.

[140] NACHA. Nacha: The electronic payments association. http://www.nacha.org/.

[141] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *Proceedings 7th USENIX Security Symposium (San Antonio, Texas)*, Jan 1998.

[142] Napster home page. http://www.napster.com/.

[143] Auditing issues in secure database management systems. National Computer Security Center Technical Report-005, Volume 4/5, May 1996.

[144] J. Nechvatal. A public-key - based key escrow system. *Journal of Systems and Software*, 35(1):73–83, 1996.

[145] R. M. Needham. Denial of service. In Proceedings of the 1st ACM Conference on Computer and Communications Security, pages 151–153, Fairfax, Virginia, November 1993.

[146] Roger M. Needham. Denial of service: an example. *Communications of the ACM*, 37(11):42–46, 1994.

[147] Peter G. Neumann. Inside risks: denial-of-service attacks. *Communications of the ACM*, 43(4):136–136, 2000.

[148] http://news.cnet.com/ news/ 0-1005-200-7535856.html.

[149] R. Oppliger. Internet security: firewalls and beyond. Communications of the ACM, 40(5), May, pp. 92-102., 1997.

[150] The flow control algorithm for the distributed 'broadcast-route' networks with reliable transport links. http://www.grouter.net/ gnutella/ flowcntl.htm.

[151] Christopher R. Palmer and J. Gregory Steffan. Generating network topologies that obey power laws. In *Proceedings of GLOBECOM 2000*, November 2000.

[152] Paypal web site. http://www.paypal.com/.

[153] T. Pederson. Electronic payments of small amounts. Technical report, Aarhus University, Tech. Report, DAIMI PB-495, Computer Science Dept., August 1995.

[154] R. Perlman. Byzantine routing. PhD Thesis, Department of Computer Science, MIT, 19??.

[155] Peterson and Davie. Computer networks.

[156] Jeffrey Picciotto. The design of an effective auditing subsystem. In *IEEE Symposium on Security and Privacy*, pages 13–22, 1987.

[157] T. Ptacek and T. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., January 1998.

[158] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, 1998.

[159] Query-flood dos attacks in gnutella networks (extended version). Technical Report, *Name of Institution Anonymized*, May 2002.

[160] D. Molnar R. Dingledine, M. Freedman. Free haven. in peer-to-peer: Harnessing the power of disruptive technologies. Peer-to-Peer: Harnessing the Power of Disruptive Technologies ed. Andy Oram, O'Reilly and Associates, March 2001.

[161] M. Rabin. Digital signatures. Foundations of Secure Computation, R. DeMillo, D. Dobkin, A. Jones, and R. Lipton (editors), Academic Press, NY, 1978, 155–168., 1978.

[162] M. Rabin. Digitalized signatures and public-key functions as intractable as factorization. MIT Technical Report, MIT/LCS/TR-212, 1979., 1979.

[163] Marcus J. Ranum. Thinking about firewalls. Technical report, 1993.

[164] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. ACM SIGCOMM, 2001.

[165] Recording industry seeks to attack file swappers, network world fusion. http:// www.nwfusion.com/ newsletters/ fileshare/ 2001/ 01073393.html.

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

[167] Reputation research network home page. http://databases.si.umich.edu/reputations/.

[168] Reputation technologies home page. http://reputation.com.

[169] Paul Resnick, Richard Zeckhauser, Eric Friedman, and Ko Kuwabara. Reputation systems. Communications of the ACM, pages 45-48, December 2000.

[170] Jennifer Rexford, Flavio Bonomi, Albert Greenberg, and Albert Wong. Scalable architectures for integrated traffic shaping and link scheduling in high-speed atm switches. *IEEE Journal on Selected Areas in Communications*, 15(5):938–950, June 1997.

[171] R. Rivest, A. Shamir, and L. Adelman. A method for obtaining digital dignatures and public-key cryptosystems. Communications of the ACM 21 (1978), 120–126., 1978.

[172] Ronald L. Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Security Protocols Workshop*, pages 69–87, 1996.

[173] Ronald L. Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Security Protocols Workshop*, pages 69–87, 1996.

[174] D. Roger. The free haven project: Design and deployment of an anonymous secure data haven. Massachusetts Institute of Technology Masters Thesis May 2000., 2000.

[175] Sachrifc: Simple flow control for gnutella. http://www.limewire.com/ developer/ sachrifc.html.

[176] A. Rowstron, P. Druschel, and P. Scalable. Distributed object location and routing for largescale peer-to-peer systems. In Proc. IFIP/ACM Middleware 2001, Heidelberg, Germany, November 2001.

[177] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218, 2001.

[178] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[179] Steve Schneider. Formal analysis of a non-repudiation protocol. In *PCSFW: Proceedings of The 11th Computer Security Foundations Workshop*, pages 54–65. IEEE Computer Society Press, 1998.

[180] B. Schneier and J. Kelsey. Tamperproof audit logs as a forensics tool for intrusion detection systems. Computer Networks and ISDN Systems, 1999.

[181] Christoph L. Schuba, Ivan V. Krsul, Markus G. Kuhn, Eugene H. Spafford, Aurobindo Sundaram, and Diego Zamboni. Analysis of a denial of service attack on TCP. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 208–223. IEEE Computer Society, IEEE Computer Society Press, May 1997.

[182] George M. Scott. Auditing large scale data bases. VLDB 1977: 515-522, 1977.

[183] http://www.setco.org.

[184] Security applications for the palmpilot. http://www.pilotzone.com/palm/util_security_default.html.

[185] R. Sekar, Y. Guang, S. Verma, and T. Shanbhag. A high-performance network intrusion detection system. In *ACM Conference on Computer and Communications Security*, pages 8–17, 1999.

[186] A. Shamir. How to share a secret. Communications of the ACM, 22:612–613, 1979.

[187] A. Shamir. Partial key escrow: a new approach to software key escrow. Presented at Key Escrow Conference, Washington, D.C., September 15, 1995., 1995.

[188] Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. http://www.cs.rice.edu/Conferences/IPTPS02/173.pdf, March 2002.

[189] Alex C. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. Hash-based ip traceback. In *Proc. of the ACM SIGCOMM 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 2001.*, 2001.

[190] Oliver Spatscheck and Larry L. Peterson. Defending against denial of service attacks in scout. In *Operating Systems Design and Implementation*, pages 59–72, 1999.

[191] Anna Karlin Stefan Savage, David Wetherall and Tom Anderson. Network support for ip traceback. In *ACM/IEEE Transactions on Networking, 9(3), June 2001*, 2001.

[192] Anna Karlin Stefan Savage, David Wetherall and Tom Anderson. Practical network support for ip traceback. In *Proceedings of the 2000 ACM SIGCOMM Conference, pp. 295-306, Stockholm, Sweden, August 2000*, 2001.

[193] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. Technical Report TR-819, MIT, March 2001.

[194] Q. Sun, N. Daswani, and H. Garcia-Molina. Maximizing remote work in flooding-based peer-to-peer systems. In *17th International Symposium on Distributed Computing (DISC 2003)*, October 2003.

[195] Q. Sun and H. Garcia-Molina. Slic: A selfish link-based incentive mechanism for unstructured peer-to-peer networks. In *24th International Conference on Distributed Computing Systems*, 2004.

[196] Qi Sun. *Improving the performance of P2P Systems.* PhD thesis, Stanford University, 2004.

[197] R. Taylor. Non-repudiation without public-key. In *Australasian Conference on Information Security and Privacy*, pages 27–37, 1996.

[198] The bittorrent official home page. http://bitconjurer.org/BitTorrent/.

[199] Dimitrios Tsoumakos and Nick Roussopoulos. Adaptive probabilistic search (aps) for peer-to-peer networks. http://citeseer.nj.nec.com/568292.html.

[200] W3C. The world wide web consortium: Extensible markup language. http://www.w3.org/XML/.

[201] W3c website on web services. http://www.w3.org/2002/ws.

[202] S. Walker, S. Lipner, C. Ellison, D. Branstad, and D. Balenson. Commercial key escrow: Something for everyone now and for the future. TIS report 541, January 1995.

[203] S.T. Walker. Thoughts on key escrow acceptability. TIS Report 534D, Trusted Information Systems, November 1994.

[204] Gregory B. White, Eric A. Fisch, and Udo W. Pooch. Cooperating security managers: A peer-based intrusion detection system. In *IEEE Network*, volume 10, pages 20–23, 1996.

[205] M. Wright, M. Adler, B. Levine, and C. Shields. An analysis of the degradation of anonymous protocols, technical report, univ. of massachusetts, amherst, 2001.

[206] Rebecca N. Wright, Patrick Lincoln, and Jonathan K. Millen. Efficient fault-tolerant certificate revocation. In *ACM Conference on Computer and Communications Security*, pages 19–24, 2000.

[207] X12. The accredited standards committee (asc) x12. http://www.x12.org/.

[208] B. Yang and H. Garcia-Molina. Designing a super-peer network. Submitted for publication.

[209] Beverly Yang and Hector Garcia-Molina. Ppay: Micropayments for peer-to-peer systems. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[210] C. Yu and V. Gligor. A formal specification and verification method for the prevention of denial of service. In Proc. 1988 IEEE Symposium on Security and Privacy, pages 187–202, Oakland, CA USA, April 1988. IEEE Computer Society Press. 117, 1988.

[211] Yongguang Zhang and Wenke Lee. Intrusion detection in wireless ad-hoc networks. In *Mobile Computing and Networking*, pages 275–283, 2000.

[212] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, April 2001.

[213] Zhou and Gollmann. Observations on non-repudiation. In *ASIACRYPT: Advances in Cryptology – ASIACRYPT: International Conference on the Theory and Application of Cryptology*. LNCS, Springer-Verlag, 1996.

[214] Zhou and Gollmann. An efficient non-repudiation protocol. In *PCSFW: Proceedings of The 10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.

[215] J. Zhou and D. Gollmann. Evidence and non-repudiation. Journal of Network and Computer Applications, London: Academic Press, 1997.

[216] J. Zhou and D. Gollmann. Towards verication of non-repudiation protocols. In T. Vickers J. Grundy, M. Schwenke, editor, International Renement Workshop and Formal Methods Pacic 1998, pages 370-380. SpringerVerlag, 1998., 1998.

[217] Jianying Zhou and Dieter Gollmann. A fair non-repudiation protocol. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 55–61, Oakland, CA, 1996. IEEE Computer Society Press.

[218] E. Zwicky, S. Cooper, D. Chapman, and D. Ru. Building internet firewalls. O'Reilly and Associates, Inc., 2nd edition, 2000.