Kudzu:

A Decentralized and Self-Organizing Peer-to-Peer File Transfer System

by

Sean K. Barker

Jeannie Albrecht, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

May 25, 2009

Contents

1 Introduction **8**
- 1.1 Goals . . . 10
- 1.2 Contributions . . . 10
- 1.3 Contents . . . 11

2 Background **12**
- 2.1 Networking Paradigms . . . 12
- 2.2 P2P Paradigms . . . 13
  - 2.2.1 Napster . . . 13
  - 2.2.2 Kazaa . . . 14
  - 2.2.3 Gnutella . . . 15
  - 2.2.4 BitTorrent . . . 16
  - 2.2.5 DHTs . . . 18
- 2.3 Properties of P2P Networks . . . 19
  - 2.3.1 Scalability . . . 19
  - 2.3.2 Incentives . . . 20
  - 2.3.3 Download Performance . . . 21
- 2.4 Summary . . . 21

3 Kudzu: An Adaptive, Decentralized File Transfer System **22**
- 3.1 Design Goals . . . 22
- 3.2 Network Structure and Queries . . . 23
  - 3.2.1 Query Behavior . . . 23
  - 3.2.2 Keyword Matching . . . 24
- 3.3 Network Organization . . . 25
  - 3.3.1 Organization Policies . . . 26
  - 3.3.2 Naive Policy . . . 27
  - 3.3.3 Fixed Policy . . . 27
  - 3.3.4 TF-IDF Ranked Policy . . . 28
  - 3.3.5 Machine Learning Classifier Policy . . . 30
- 3.4 Download Behavior . . . 33
  - 3.4.1 File Identification . . . 33
  - 3.4.2 Chunks and Blocks . . . 34
  - 3.4.3 Swarms . . . 35
  - 3.4.4 Gossip . . . 36
- 3.5 A Distributed Test Framework . . . 37
  - 3.5.1 Simulating User Behavior . . . 37
  - 3.5.2 Replayer Design . . . 38
- 3.6 Summary . . . 39

4 Implementation: The Kudzu Client 40
4.1 Communication Framework . . . 40
4.1.1 Java RMI . . . 41
4.1.2 Java Serialization . . . 42
4.1.3 Protocol Buffers . . . 42
4.1.4 Kudzu Message Encoding . . . 43
4.1.5 Connection Management . . . 44
4.2 Message Types . . . 46
4.3 Test Framework . . . 47
4.3.1 Data Parsing and Cleaning . . . 49
4.3.2 Virtual User Assignment . . . 49
4.3.3 Simulation . . . 50
4.3.4 Logging . . . 50
4.3.5 Bootstrapping . . . 51
4.4 Summary . . . 51

5 Evaluation 52
5.1 Evaluation Metrics . . . 52
5.1.1 Bandwidth Utilization . . . 52
5.1.2 Query Recall . . . 53
5.1.3 Download Speeds . . . 54
5.2 Dataset Peer Selection . . . 54
5.3 Bandwidth Motivation . . . 55
5.4 Organization Strategies . . . 57
5.4.1 Policy Bandwidth Use . . . 58
5.5 Query Recall Tests . . . 59
5.5.1 Network Organization . . . 60
5.6 Download Tests . . . 69
5.7 Summary . . . 71

6 Conclusion 72
6.1 Future Work . . . 72
6.1.1 Organization with Machine Learning Classifiers . . . 72
6.1.2 Incentive Model and Adversaries . . . 73
6.1.3 Testing Environment . . . 73
6.1.4 New Datasets . . . 74
6.1.5 Anonymity and Privacy . . . 74
6.2 Summary of Contributions . . . 75

List of Figures

2.1 Client-server network (left) and peer-to-peer network (right). 13
2.2 Example Napster network. 14
2.3 Example Kazaa network with three supernodes. 15
2.4 Example BitTorrent network with two seeders and three leechers. 17

3.1 A non-optimal separating hyperplane H1 and an optimal separating hyperplane H2 with margin m. Test point T is misclassified as black by H1 but correctly classified as white by H2. 32
3.2 A Kudzu network of 5 nodes containing 3 download swarms. Solid lines indicate peer connections, while dotted lines indicate swarm connections. 34

4.1 User interaction with the Kudzu client. 41
4.2 One of Kudzu's protocol buffer definitions. 43
4.3 Protocol buffer specification of base container message. 44
4.4 Protocol buffer specification of all message payload types. 48
4.5 An example dataset user entry with 1 file and 2 queries. 49

5.1 Unique query ratios in a network with uncapped TTL. 56
5.2 Aggregate bandwidth usage across a range of max TTL values. 57
5.3 Aggregate bandwidth usage versus max TTL for each of the four organization strategies. 58
5.4 Query recall versus max TTL for each of the four organization strategies. 60
5.5 Network topology resulting from naive organization. Note the weakly connected cluster in the upper right. 62
5.6 Circular network topology resulting from naive organization with passive exploration. 64
5.7 Circular network topology resulting from naive organization with active exploration. 64
5.8 Naive organization with passive exploration and noted coverage gaps (shaded regions) and highly interconnected node groups (demarcated by lines). 65
5.9 Circular network topology resulting from TFIDF organization with passive exploration. 67
5.10 Circular network topology resulting from TFIDF organization with active exploration. 67
5.11 Aggregate bandwidth usage versus max TTL including naive with active exploration. 68
5.12 Query recall versus max TTL including naive with active exploration. 68
5.13 Download completion CDFs for Kudzu and BitTorrent. 70

List of Tables

2.1 Overview of P2P network paradigms. 18

5.1 Overview of benefits and limitations of our four organization strategies. 69

Abstract

The design of peer-to-peer systems presents difficult tradeoffs between scalability, efficiency, and decentralization. An ideal P2P system should be able to scale to arbitrarily large network sizes and be able to accomplish its intended goal (whether searching or downloading) with a minimum amount of overhead. To this end, most P2P systems either possess some centralized components to provide shared, reliable information or impose high communication overhead to compensate for a lack of such information, both of which are undesirable properties. Furthermore, testing P2P systems under realistic conditions is a difficult problem that complicates the process of evaluating new systems. We present Kudzu, a fully decentralized P2P file transfer system that provides both scalability and efficiency through intelligent network organization. Kudzu combines Gnutella-style querying capabilities with BitTorrent-style download capabilities. We also present our P2P test harness that replays genuine P2P user data on Kudzu in order to obtain realistic usage data without requiring an existing user base.

Acknowledgements

Foremost thanks are due to my advisor, Jeannie Albrecht, for mentoring me both in this thesis and in the rest of my computer science education at Williams. This work would not have been possible without her guidance and suggestions. Thanks are also due to Tom Murtagh, my second reader, for helpful comments during editing as well as to the rest of the department for providing an engaging academic environment for the past four years. I am also grateful to my girlfriend Lizzie and the rest of my family for their patience and understanding while I worked on this thesis. Finally, a thanks to my fellow thesis students Catalin and Mike and the rest of my computer science friends for many shared late nights in the lab.

Chapter 1

Introduction

In the past decade, one of the greatest beneficiaries of increasing consumer broadband adoption has been the development of peer-to-peer (P2P) systems. The traditional model of online content consumption is based around dedicated providers such as corporate web servers that provide upstream content to home users and other content consumers. In this model, providers are generally companies or technically savvy users, but the majority of Internet users do not share content directly with each other due to technical barriers such as the knowledge required to set up and manage a server. The onset of high-bandwidth, always-on broadband connections and a greater prevalence of high-demand electronic media such as MP3s brought with it new opportunities to provide services through users themselves. To this end, peer-to-peer systems emerged in which users were able to share content directly with each other, circumventing both intermediary services and often (to the chagrin of the traditional content providers) legal restrictions. In recent years, P2P usage has seen dramatic increases and is now one of the most prevalent forms of online activity: recent surveys of net usage have ranked P2P traffic as the largest consumer of North American bandwidth, accounting for nearly half of all online traffic and roughly three quarters of upstream traffic [29].

P2P systems have been applied to a variety of functions, with file sharing being the most widely known. However, P2P systems have diverged widely according to various design choices. One of the most important factors separating one P2P system from another is the system's degree of decentralization. Under the traditional provider-consumer model, centralization and the problems that come with it were taken for granted, and steps were taken to compensate, usually by adding backup machines. In the P2P paradigm, however, there is the opportunity to build systems that do not rely on specific machines, network connections, or users to function normally. In such a system, service downtime is typically significantly less and maintenance to keep the service running is greatly reduced if not outright eliminated.

Centralization, however, has some clear benefits when applied to an (ostensibly) P2P systems. Centralized systems are easy to design, well understood, and simple to control. It is likely no coincidence that the first successful P2P system, Napster, was totally reliant on a centralized server to match users and initiate file transfers. Though it was heralded as a P2P system both by proponents and detractors, Napster was effectively a centralized service that simply delegated the final pieces

of work to the users themselves. Napster ultimately fell victim to its centralization and was forcibly shut down, thus completely eliminating the service overnight. More decentralized networks, while not subject to the same sort of problems as Napster, have made various sacrifices to centralization. The Gnutella network, for instance, was in its original incarnation fully decentralized, but did not scale to large network sizes due to excessive network overhead. Later incarnations of the network compensated by promoting certain peers to special status, thereby forming hubs in the network and introducing potential problem points. BitTorrent networks, while offering efficient and high-performance parallel downloads, sacrifice the entire capability of file querying in favor of centralized 'trackers' and rely on centralized repositories of torrent files to allow users to connect to the network. This means that third parties such as Google or sites like The Pirate Bay are relied on to actually find content on a BitTorrent network.

While decentralized P2P systems have been heavily studied, in practice, truly decentralized systems have been shown to be prone to serious scalability issues. In large part, this has been a result of the difficulty of finding resources on a decentralized network when there is no central authority to query. Systems have turned to searching significant portions of the network to compensate for a lack of central information (resulting in excessive bandwidth consumption, as occurred in the original Gnutella), or have centralized parts of the network to reduce the amount of searching required (as is the case in Kazaa and later versions of Gnutella).

A substantial amount of work has been done in addressing the problems of decentralized P2P systems. One of the primary issues, scalability, has been approached by imposing organization schemes on peers in the network in order to keep peers connected to the 'best' neighbors. Several metrics have been used for this, such as social network properties [23] and peer bandwidth capacities [7].

However, one issue pertinent to most of this work is the difficulty of performing realistic tests of new systems (both in isolation and for comparison to existing systems). This difficulty is due primarily to three issues:

1. Real-life P2P networks are often comprised of hundreds or thousands of users covering a wide geographical area. With a new system (and thus without an existing user base), scaling a test to realistic sizes is difficult, particularly if real machines are used to model the network. One way to test P2P networks that has recently emerged is PlanetLab [21], a global wide-area testbed of roughly a thousand machines freely available to researchers. While not as large as many real P2P networks, PlanetLab is nevertheless a significant asset in evaluating a P2P system on an actual network without resorting to a network simulator.

2. P2P networks are subject to a variety of exceptional occurrences and problems, including network congestion, machine failures, and any other agents in the network that may interfere with regular operations (such as firewalls). Accounting for all of these variables in a simulation is difficult when using a network simulator, especially since some of these variables may be unanticipated. Simulations conducted on a live network, while subject to the problems of scale discussed above, deal with all exceptional cases of a real deployment, potentially resulting in more realistic results.

3. User behavior is non-uniform and difficult to model, yet critical for determining a system's real-world feasibility. One effective way to model actual users is to employ actual user data, which must be captured from an existing network and mapped onto a new system. Comprehensive data of this kind has begun to emerge in recent years [12, 4]; however, we are not aware of any large-scale efforts to use this data in the evaluation of new systems on realistic networks. The use of such data, however, presents an opportunity to run more realistic experiments than those that infer user data and/or behavior.

One approach to dealing with these problems is to create extensions on top of other systems; for instance, Tribler [23] is implemented as a set of extensions on top of a standard BitTorrent client. While granting access to a preexisting network of many users, this approach forces the system into compliance with an existing system, which may not be desirable. Employing preexisting test data, however, removes one of the hurdles to evaluating a brand new P2P design.

1.1 Goals

This thesis presents Kudzu, a new peer-to-peer file sharing system. The first goal of Kudzu is to be completely decentralized; that is, every peer in the network is no more and no less important than any other peer. Peers should be able to connect to the network through any other peer in the network and should continue to function in spite of arbitrary network outages (down to the simplest case of two peers communicating with each other). Peers should be able to form a new Kudzu network or join an existing one with nothing other than the standard client.

The second goal of Kudzu is to have the network intelligently organize itself in the context of total decentralization. This is roughly equivalent to saying that Kudzu must be efficient; inter-peer communication should not be excessive and desired resources in the network should be located quickly and easily. Kudzu should also display download performance comparable to leading P2P systems by maximizing the use of available bandwidth while minimizing communication overhead – this should demonstrate the potential of fully decentralized P2P systems to also display high performance.

The third goal of Kudzu is to present a series of realistic simulations that allow us to draw conclusions about decentralized P2P systems. The simulations should account for variability in network and machine conditions and should reflect the behaviors of actual users, which provides results more applicable to real deployments of the system. We carry out these tests using the PlanetLab testbed and a set of real user data gathered from a Gnutella network.

1.2 Contributions

We present **Kudzu**, a new P2P file transfer system design that draws on successful ideas from past and present P2P systems while addressing many of their individual shortcomings. Kudzu aims to encompass high performance, reliable querying, and high efficiency, all within a completely decentralized environment. We also present an implementation of Kudzu, which we use to evaluate

the efficacy of our design and draw conclusions about decentralized P2P systems of this type. In order to ensure that our results are applicable to a real-world setting, we employ a real-world dataset and run our experiments on a wide area network of nodes. We demonstrate our system's performance in comparison to existing systems such as BitTorrent and our system's ability to scale to large numbers of peers. Finally, we describe our experiences during the process of designing and building the system and discuss the ways in which we believe decentralized P2P systems stand to be improved by employing intelligent, adaptive behavior.

1.3 Contents

The thesis is organized by chapter as follows:

Chapter 2 provides an overview of major, well-known P2P systems as examples of the varying degrees of centralization, scalability, and capabilities in P2P systems today. We also provide an overview of related work on improving these types of P2P networks, with particular attention paid to systems aiming to be highly decentralized. This discussion frames the design choices we made for Kudzu and the ideas we chose to incorporate into the system.

Chapter 3 describes the design of Kudzu, a file sharing system that aims to efficiently organize the network and facilitate powerful query and download capabilities while remaining completely decentralized. We describe Kudzu's network structure, querying capabilities, and download behaviors and the factors that led us to make our design decisions. We also describe the design of our wide-area test harness that allows for realistic tests of the system.

Chapter 4 provides a technical overview of our implementation of Kudzu. We discuss the messaging framework for communication between Kudzu peers and the way in which information is encoded. As experiments on wide-area networks are often significantly more nuanced in practice than in theory, we also discuss relevant technical details behind our test harness and our coordination of large numbers of machines in order to run cohesive tests.

Chapter 5 presents our empirical results from running experiments on Kudzu using our test harness. We discuss the conclusions that can be drawn from our results as well as their potential applications to other types of P2P networks.

Chapter 6 provides an overview of our work and discusses future work on the system. We also detail several aspects of P2P systems that we did not explore in depth and discuss how they could be incorporated into future versions of the system.

Chapter 2

Background

2.1 Networking Paradigms

Traditionally, approaches to building large-scale networked systems have been dominated by a client-server approach, in which a service is provided to a user base exclusively by a few centralized servers. This type of approach is natural to consider at first – it is simple to design and implement, since all information is processed centrally, and easy to control, as the whole service is contingent on the small, pre-designated set of server machines.

There are a variety of drawbacks, however, to the standard client-server approach. Perhaps the greatest is the difficulty of scaling up to a large user base. Since the set of servers is effectively statically serving a dynamic (and often growing) number of users, the load of each server is liable to continuously increase. Once the servers' capacity is reached, new servers must be added; this adds the cost of installing new hardware, the complexity of running more servers in parallel, and a greater chance of a server failure, leading to possible service outages. Of course, the risk of server failure is always present in a client-server approach, and is another significant problem with the paradigm. The servers are inherently a central point of failure for the model; if the servers go down, the service is immediately and completely shut down. The addition of failover servers can alleviate this issue, but is still only a temporary solution to a problem that may still present itself if the user base grows large enough or a significant enough failure occurs.

While the client-server model has dominated networked systems since the dawn of the Internet, a new paradigm has emerged relatively recently in the form of peer-to-peer (P2P) networks that promises to address the problems of the client-server model. A P2P network may loosely be defined as a network in which communication occurs not between users and a centralized server but directly between the users of the service. This has several immediate advantages: with the elimination of servers comes not only the removal of the central points of failure but also a (theoretically) infinite capacity, as adding more users to the network not only increases the demand on the network but the bandwidth and computational capacity available to it. Diagrams illustrating typical client-server and P2P architectures are shown in Figure 2.1.

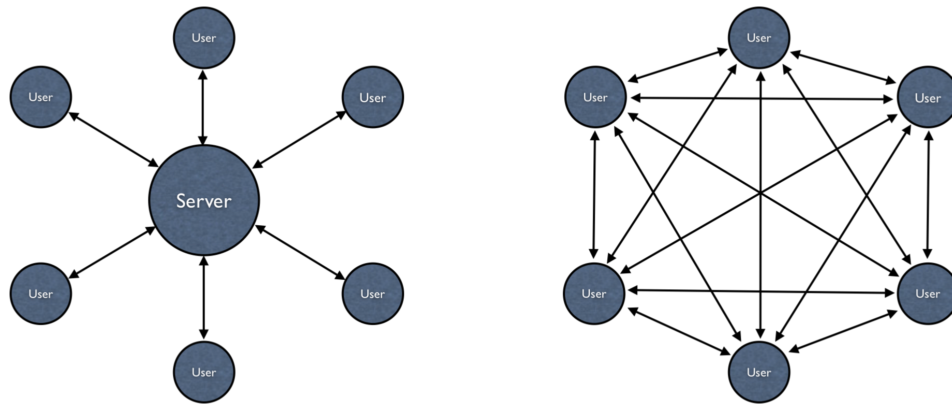

Figure 2.1: Client-server network (left) and peer-to-peer network (right).

2.2 P2P Paradigms

A peer-to-peer system has been used as an umbrella term to refer to many types of systems that adhere in varying degrees to the description of a "pure" P2P system given above. Rather than attempting to enumerate every point along this spectrum, it is most informative to consider several of the most popular and well-known P2P systems that have emerged (and in some cases, dissolved) in recent years. Though these all have been widely accepted as examples of "P2P systems", they vary significant in their technical underpinnings, and each represents a distinctive approach to designing P2P systems.

The core purpose of the systems that we consider here is the transfer of files. A P2P file transfer is generally a two-step process: first, a desired file must be located on the network (querying), and second, the file itself must be transferred (downloading). These two functions can be separated fairly naturally, since locating and transferring the resource are non-overlapping tasks. As a result, some systems focus on one function or the other while mitigating or ignoring the other completely. The most notable instance of this is BitTorrent, which by design facilitates downloads only and provides no function to query for files. Our discussion will take into account both the query and download aspects of these systems – though the lack of one or the other is not exactly a deficiency, we are ultimately interested in an integrative system that performs both functions.

2.2.1 Napster

Probably not coincidentally, the first popular P2P system that emerged was also the furthest from the true P2P paradigm, as it possessed considerable similarities to a client-server architecture. This was Napster, which allowed its users to exchange music files directly with each other[1]. Napster was indeed a P2P system in the sense of having users connect directly to each other; however, it relied on a central server to match users together who wished to exchange music with each other. When

[1]Note that the Napster we refer to here is the original (circa 2000) incarnation. While a service with the Napster name still exists, it is unrelated to the original and not relevant to our discussion.

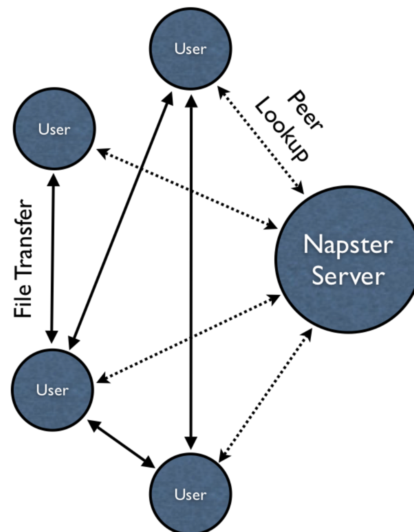

Figure 2.2: Example Napster network.

a peer wished to find a file, it contacted the central server, which looked up which peers had the desired file, then instructed the requester to connect to those peers. This system has significant scalability benefits, as the server's role was effectively limited to serving only as a catalogue that users queried to determine appropriate peers with which to connect However, the single point of failure remained, as the entire network relied on Napster's central server to find out where other peers were located and what files they had to share. An example Napster network with four users (and arbitrary inter-peer connections) is shown in Figure 2.2.

Napster's central point of failure proved to be its downfall. After a series of lawsuits filed against the network alleging copyright infringement [2], a court order forced Napster to shut down the central server – and with that, the Napster P2P network disappeared overnight. While this was an artificially imposed outage rather than a technically related one, it illustrated many of the problems behind Napster's architecture that were inherited from the client-server paradigm. Napster was succeeded by several P2P systems that addressed many of its problems.

2.2.2 Kazaa

The Kazaa system came into popularity around the same time as Napster, but was closer to a 'pure' P2P system than Napster, and as such was not subject to many of Napster's problems. A Kazaa network does not maintain a single central repository of content information, as Napster did. Instead, each peer is assigned to be either a regular node (RN) or a 'supernode' (SN). Each supernode is responsible for a set of regular nodes and maintains all file information for those nodes as well as connections to other supernodes [16]. Thus, the supernodes function as mini-servers of sorts, performing distributed file lookups over the entire network. The network ends up shaping itself into a tree, with ordinary nodes as leaves attached to supernodes above them. File queries are

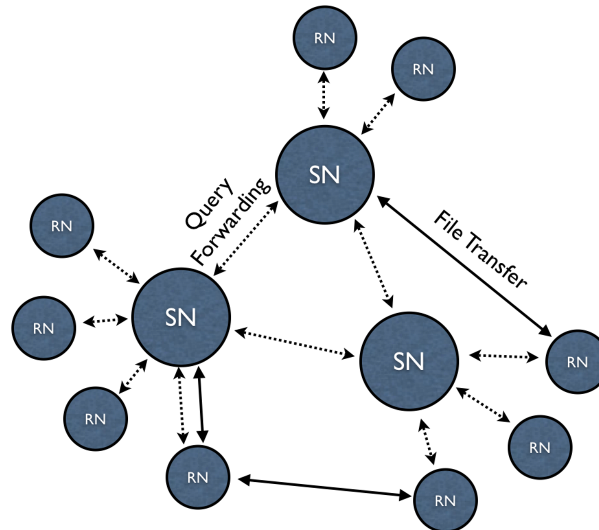

Figure 2.3: Example Kazaa network with three supernodes.

directed to the node's supernode, which then may forward the query onto other supernodes, thereby searching some subset of the network. As in Napster, once a file sender and receiver is determined, a direct connection between the two is opened to perform the transfer, as shown in Figure 2.3.

Since the supernodes are dynamic and constantly changing, the network will continue to function if individual nodes or sets of nodes are taken offline. However, the Kazaa architecture introduces several new issues. Maintaining a useful set of supernodes imposes network overhead – if the set of supernodes is poor (for instance, if the supernodes become overloaded or have too little bandwidth to begin with), the network will function sub-optimally. Additionally, nodes have no control over when they become supernodes, which is troublesome from the perspective of fairness when a user's machine suddenly becomes a mini-hub for the network and begins to route a large amount of traffic for other users. However, the specifics of Kazaa's protocol (called FastTrack) are proprietary and not entirely known [33], so Kazaa is generally less understood than the other systems described here.

2.2.3 Gnutella

The purest well-known P2P system we discuss here is that of Gnutella. A Gnutella network closely resembles our original description of a P2P systems – the network is functionally homogeneous, so unlike the other systems discussed, there are no peers that can be considered servers of any kind. Functionally, it operates fairly similarly to a Kazaa network, in that nodes search for files by querying their set of connected peers, which in turn forward to their connected peers, and so forth, up to a maximum number of hops. If a peer receives a query matching one of its files, it connects back to the requester and starts the transfer [7].

In this pure form, a Gnutella network is clearly unscalable, as the load on each node grows linearly with the number of queries (which increases as the network grows in size). While this may

seem manageable at first glance, note that this means the total amount of traffic the network has to handle grows exponentially; each new node has to handle each new query, resulting in more and more bandwidth used as the network grows. An analysis of early Gnutella bandwidth usage estimated that in a Gnutella network with as many users as Napster in its prime, the network might have to expend as much as 800 MB handling a single query [25]. The same analysis continues on to conclude that the same network as a whole would have to transfer somewhere between 2 and 8 gigabytes *per second* in order to keep up with demand. While many assumptions are used in order to arrive at these measurements, the scale of the results alone is enough to raise questions about the viability of a large Gnutella network.

While scalability is problematic for a Gnutella network, however, the network also possesses many positive qualities. For one, it is extremely robust to node failures and changes in network topology and requires very little organizational overhead [11]. Furthermore, the query model is quite powerful; queries are routed from node to node and each individual node is left free to match their files against queries in any way that they wish. This means that arbitrarily powerful matching algorithms can be used as drop-in replacements to the network to improve query results. The compromises that other systems make away from a Gnutella-like query approach typically sacrifice flexibility in order to achieve better network efficiency and scalability.

While early versions of Gnutella adhered to the fully decentralized model described above, later versions of Gnutella introduced 'UltraPeers', which are high-capacity peers similar to Kazaa's supernodes. UltraPeers alleviated the unscalable query load on most peers by handling most of the query traffic for the entire network. UltraPeers maintained connections to many (typically around 32) other UltraPeers, thus allowing regular nodes to maintain only a few connections to UltraPeers and shielding them from the majority of queries passing through the network. Most properties of Kazaa previously discussed can be applied to an UltraPeer-era Gnutella network. We are mostly interested in Gnutella as an example of a fully decentralized network, and so generally refer to 'Gnutella-like' networks as loosely organized networks in which any centralization is kept to an absolute minimum.

2.2.4 BitTorrent

Lastly, we discuss BitTorrent, which is important not only because it represents a unique approach to P2P downloads but also because it is one of the most successful mainstream P2P systems today and is rapidly growing in use [3]. BitTorrent functions not as a single large network but as a large number of small networks, each controlled by a tracker. Each tracker is setup to transfer a single file among all peers connected to its network (this set is called a 'swarm'), and new peers join by contacting the tracker. Since every peer connected to the tracker is interested in sharing ('seeder' nodes) or downloading ('leecher' nodes) the same file, transfers can be conducted efficiently in a distributed, block-by-block fashion. An example BitTorrent network is shown in Figure 2.4.

While trackers themselves do not represent a particularly serious central point of failure due to the number of trackers in use and the ease of starting a new tracker, trackers are still a problem for several reasons:

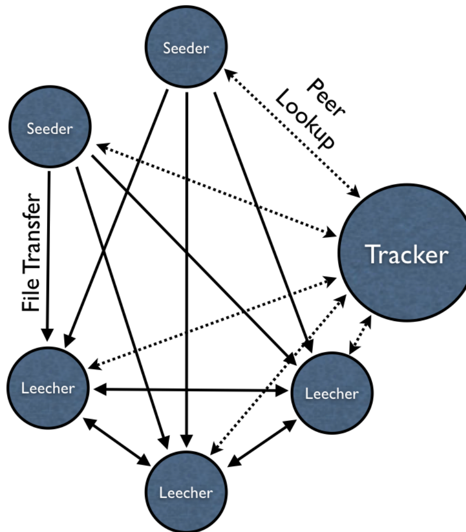

Figure 2.4: Example BitTorrent network with two seeders and three leechers.

- A file can only be shared if someone has actively set up a tracker to share that file. This is in contrast to the other systems, in which it is only necessary for someone on the network to possess the file in question. This means that a file will only be transferred if both the uploader and downloader have decided it is worthwhile to share. However, there is no obvious incentive for the uploader to start up the tracker vs waiting for someone else to start one, so the net result will be many files that may have interested downloaders but no trackers and thus no one to upload.

- The file required to locate a particular tracker must be acquired externally (a 'torrent' file, or simply torrent), since having the file is a prerequisite to joining the BitTorrent network. Typically, tracker files are downloaded from web repositories that serve the dual function of housing tracker files and locating trackers for a desired file (another function that cannot be built into a BitTorrent network). This, however, introduces another dependency and possible point of failure into the network. Many of these tracker sites have come under litigation similarly to the original Napster service [20].

Furthermore, because each BitTorrent network exists to transfer a specific file, BitTorrent networks possess no search capabilities at all. This is one of BitTorrent's significant weaknesses vs Gnutella, which allows search engine-like queries across the network to find relevant files without resorting to an external service (e.g., Google) to locate a torrent file. Of course, one might ask why this is something to be avoided; a search engine like Google employs highly sophisticated search algorithms and is adept at finding desired files. There are a few problems with using a third party like Google for searches, however. One is that since the torrent file does not contain the actual file itself, the only indication of what's contained in the torrent is the torrent filename (which may be misleading). A larger problem is that finding a torrent file does not equate to finding an active

	Centralization	**Query Model**	**Scalability**	**Overhead**
Napster	High; central server	Direct server lookup	High	Low
Kazaa	Moderate; SuperNodes	Query flooding	Moderate	Moderate
Gnutella	Low (pre-UltraPeers)	Query flooding	Low	Low
BitTorrent	Moderate; trackers	N/A	High	Moderate
DHT	Low	Direct lookup (exact)	High	High

Table 2.1: Overview of P2P network paradigms.

network – many torrent files point to old networks that have gone dormant and no longer have any uploaders sharing the file. This means that finding a network with enough (or any) uploaders to obtain a file may be more difficult than simply making a Google search and downloading the first torrent file found.

2.2.5 DHTs

One final type of system that bears mention is a Distributed Hash Table (or DHT). DHTs, while not complete P2P systems in the same manner as the others described here, are distributed lookup tables that can serve as backbones for P2P networks, performing efficient $O(log\ n)$ file lookups across data distributed amongst the nodes in a network. DHTs typically organize their nodes in a structure that indexes a subset of the other nodes and allows particular pieces of information to be retrieved without traversing most of the network. DHTs themselves are an active field of research with many well-known and highly studied systems such as Chord [31], CAN [24], and Pastry [27].

DHTs have also been proposed for use in P2P systems. Some BitTorrent clients possess 'trackerless' operation modes in which a DHT is used in order to allow the network to function without a tracker [18]. However, the use of DHTs in P2P systems is far from an ideal solution. Chawathe et al [7] outline several of the problems of using DHTs in a P2P network. One issue is the high degree of churn in a typical P2P network. Since DHTs are highly structured, there is significant overhead incurred when nodes are added or removed from the network. In a typically P2P network, peers are frequently entering and leaving, and this will imposes a significant maintenance burden if a DHT is in use. Another issue is that while DHTs perform exact match queries very well, they generally cannot perform keyword searches. Users will often not know the exact file they wish to locate, so the sacrifice of keyword searches is seriously detrimental to the network. Also note that in the specific example of BitTorrent, DHTs also do not alleviate the problem of needing to find a torrent file before joining the network. Finally, [7] argues that since most requests in P2P systems are for highly replicated files, precise DHT lookups are unnecessary.

An overview of the properties and tradeoffs of each of these network types is given in Table 2.1. While there are many specific P2P networks other than the ones listed, we feel that the 5 discussed above typify the majority of P2P systems in use today.

2.3 Properties of P2P Networks

The P2P designs discussed above vary widely in their comparative advantages and disadvantages. Some of these properties are closely tied to the high-level system design, whereas others are more flexible and have been explored by previous researchers. We discuss related work involving some of these properties below.

2.3.1 Scalability

As previously discussed, scalability is primarily a concern in a Gnutella network (and, to a lesser degree, in a Kazaa network). Gnutella captures the benefits of true decentralization but eschews the scalability gains of using a central catalog (as in Napster), a tiered structure of supernodes (as in Kazaa), or a series of small, self-contained networks (as in BitTorrent). Creating a truly scalable Gnutella-like system would have the potential to yield a system that eclipses all existing approaches.

Query Approaches

Since the number of queries is the most significant factor in scaling a Gnutella-like system, one approach to improving scalability is to adjust the manner of query forwarding from the standard flooding-based approach [11]. Gia [7] replaces flooding with a random walk biased towards high-degree nodes. Additionally, it employs one-hop replication of file data, meaning that each peer has knowledge of not only its own files but those of its neighbors. This type of approach may be used to reduce the need to employ complete flooding or low query TTLs while still affording a high probability of finding files on the network. Ges [34] takes an approach similar to Gia in using a random walk and one-hop replication but biases the walk based on node capacity rather than performing Gia's topology adaptations; this has the useful effect of controlling which nodes receive the majority of queries.

Work has also been done in merging flooding-style queries with more sophisticated techniques. Loo et al [19] propose a hybrid search approach consisting of flooding for well-replicated (that is, popular) files and DHT searches for rare files by only pushing rarer files into the DHT, thereby reducing the overhead of maintaining the DHT (which is much higher than simple flooding). Their rationale stemmed from measurements suggesting that Gnutella is good at finding well-replicated content, but often fails to return matches on rarer files, even when the network does contain peers with matches.

Social Networking Influences

Other attempts to scale decentralized systems have focused mostly on organizing the network in such a way that peers with similar interests are joined closely together. Prosa [6] leverages similarities in peer files and queries to build specific types of links between peers depending on the contact and interests shared between them — initially only 'acquaintance links', as peers communicate and display shared interests through queries and files the links change to more powerful 'semantic links'. The product is tightly bound social groups that allow rapid query propagation to those peers likely

to respond. Tribler [23] adds a more active, user-involved facet to building social networks in a P2P system by allowing users to give themselves unique IDs and then specify other users to favor and draw information from in recommending files and forwarding queries. The implicit trust in this sort of social network derived from out-of-band means also allows various performance improvements (see Section 2.3.3).

Machine Learning

A lesser explored way to build links between peers likely to exchange files in the future is to employ local machine learning algorithms to measure the usefulness of a connection to a particular peer. One approach proposed in [5] builds a classifier for neighbor suitability using support vector machines (a standard machine learning classifier). Using the query, file, and prior query match information from a small random selection of nodes in the network as training data, the algorithm predicts a small number of features (in this case, words) that are representative of the types of files the peer is interested in. Using machine learning allows the classifier to learn subtle but useful features likely to be missed by other approaches — for instance, the world 'elf' is likely to be an important feature for a node making queries for 'Tolkien' or 'Return of the King', even though 'elf' does not appear in either query. The small set of resulting features is used to predict good neighbors for future queries based on their file stores, without any input on preferences required of the user.

We were intrigued by this approach to solving the problems of decentralized networks through intelligent network organization. The simulator results given in [5] suggested that the potential of network organization to improve query performance was high. One of our goals was to determine whether this type of strategy would be effective in practice. We predicted that both heavy-weight machine learning approaches and lighter ML-derived approaches could be used to improve the performance of Gnutella-like querying in a decentralized network.

2.3.2 Incentives

One factor that has been instrumental to BitTorrent's success has been its incentive model, in which peers who are more generous uploaders are rewarded with improved download speed and selfish uploaders are punished with reduced download speeds [8]. P2P file transfer systems are inherently plagued by the problem of selfish peers (also known as 'free riders'), as they rely on (relatively) anonymous cooperation and donations of files and bandwidth in order to function well. Studies of free-riding on Gnutella demonstrated that nearly 70% of participants on the network were free-riders and roughly half of query responses came from the top 1% of sharers [1]. Even BitTorrent is not immune to the problem; the BitThief [17] system demonstrated that a fully free-riding client could achieve comparable download speeds to official clients, implying problems with BitTorrent's incentive model. Other work has been done in enforcing fairness through a trusted third party – AntFarm [22] manages block downloads through the exchange of tokens issued by a trusted server which are difficult for ordinary nodes to forge. AntFarm also leverages the token servers to manage and improve transfer speeds by viewing sets of download swarms as a bandwidth optimization problem.

Work has also been done on the price of selfishness in a Gnutella-like setting. [4] examines the

impact of reasonable self-interest in P2P networks from a game-theoretic perspective compared to altruistic behavior. The same work also proposed methods for peers to organize themselves so as to result in greater numbers of query matches. The ease with which intelligent network organization fits into a incentive-based model is one reason it shows promise for use in real systems.

2.3.3 Download Performance

Performance by itself is largely a secondary problem to scalability and is typically easier to address. Actual download speeds stem primarily from the number of peers from which downloads can proceed simultaneously. BitTorrent's model is close to ideal in this case, since everyone who has the file and is willing to share it is found effectively instantly. Assuming only modest delays in query propagation as a request travels from one end of the network to the other, a Gnutella network may be trivially modified to achieve 'optimal' performance by simply removing the max hop count on queries. Since this has the effect of drastically increasing the total number of queries propagating throughout the network, it reformulates the performance problem as a scalability or network organization problem. Total (rather than individual) download speeds on the network are a more complex issue but will still generally depend on the organization of the network and any incentive algorithms in effect.

Several proposed performance enhancements have made use of the incentive model or network organization. Collaborative downloading refers to the use of extra peers in a file transfer (i.e., neither the requester nor the original file holder) to increase available bandwidth by distributing the transfer over more peers. This requires altruism on the part of the helper nodes; Tribler [23] leverages the implicit trust in its social networks to implement the 2Fast collaborative download protocol. Collaborative downloading could probably also be applied to other, more anonymous types of incentive models.

Finally, actual observed performance in BitTorrent-like networks is heavily influenced by a large number of parameters and various settings that may have impacts ranging from minor to significant. While we do not investigate the particular effects of varying these settings, P2P clients in real networks finely tune these parameters to maximize the absolute performance observed by their users.

2.4 Summary

In recent years, P2P systems have gradually moved further away from the traditional client-server model towards a fully decentralized model in order to realize the benefits of scalability, cost, and performance possible. However, technical and scalability roadblocks have prevented the widespread adoption of truly decentralized systems in favor of systems such as BitTorrent, which sacrifice robustness and decentralization in favor of efficiency. Using intelligent network organization to compensate for decentralization, however, poses one approach to building a system that merges the benefits of a system like BitTorrent with a system like Gnutella. P2P file transfer systems stand to improve dramatically once the intersection of these two types of systems is realized.

Chapter 3

Kudzu: An Adaptive, Decentralized File Transfer System

Work on this thesis presented two general design challenges. The first was designing the Kudzu system itself; in addition to being completely decentralized, it needed to be efficient, scalable, and practical to implement. The second was designing a realistic testing framework for evaluating the performance of the system. While we built the testing framework in the context of evaluating Kudzu, there is nothing that inherently ties the framework to Kudzu, nor to our specific testbed, and the issues we faced designing a distributed testing platform are applicable to many types of distributed systems. Likewise, the decisions we made with respect to Kudzu itself are widely applicable to other P2P systems. This chapter discusses our design goals and decisions comprising both Kudzu and our test harness.

3.1 Design Goals

At its core, Kudzu is a P2P file transfer system. As with any such system, the overarching goal is to enable users of the system to locate and transfer desired resources spread out across many users with as little overhead as possible, both on the part of the user (complicated searches or excessive waiting) and the system itself (computational and bandwidth overhead). Within this context, we designed Kudzu according to the following core principles:

1. The system must be fully decentralized; that is, every agent in the network is equivalent as far as network functionality is concerned. The removal of any piece of the network should not impede the capabilities of the remaining network, and the removed piece should remain a fully functional network itself. As discussed in Chapter 2, most successful P2P systems in the past have made decisions that violate this goal by introducing some form of centralization. As we were specifically interested in exploring fully decentralized networks, the goal of decentralization was paramount in Kudzu and taken as a given for the rest of our design.

2. The system should scale to networks of arbitrary size. More specifically, the system should not degrade even when a network of only a few peers is scaled up to one with many. Real-life P2P networks often span hundreds or thousands of simultaneous users and can only be expected to grow; as such, scalability is a highly important concern of any P2P design. Moreover, the system should effectively leverage the resources of its peers. In other words, peers should be able to reliably find desired resources located in unknown locations on the network. This goal was especially interesting to consider in the context of our first goal of decentralization.

3. The system should provide the keyword searching capabilities of a network like Gnutella while also providing download capabilities comparable to a high-performance network like BitTorrent. Gnutella provides a flexible search platform in which to locate files on the network, but suffers from scalability problems (as discussed in Section 2.2.3). BitTorrent, in contrast, scales very well while maintain high speeds, but provides no search capabilities. We wish to provide both of these functions while mitigating their downsides through the use of efficient network organization.

4. The system should be feasible to implement and evaluate under live conditions. Especially given that Kudzu is a system designed from scratch rather than an extension built on top of an existing system, it was important to consider how the system could be empirically evaluated under realistic usage. This requirement led to the design of the testing and data gathering harness.

3.2 Network Structure and Queries

A Kudzu network is comprised of a set of connected peers identified by IP address. Each pair maintains a number of two-way connections to other peers in the network. Communication in the network may be visualized as exchanging messages along edges (peer connections) in an undirected graph. Loops (that is, connections to oneself) are disallowed. Each peer is capable of accomplishing every function of the network, thus making every peer itself a fully functioning Kudzu network. Of course, a node with no connections will have no one to exchange files with and thus is not useful. In practice, however, a Kudzu network must be bootstrapped by starting one or more nodes in isolation and having other peers subsequently connect. Since all connections in the network are bidirectional, the bootstrapping node will then participate in the network exactly as the other nodes do.

3.2.1 Query Behavior

In order to locate resources on the network to download, Kudzu nodes send out queries along their connections. As in a standard Gnutella network, queries are sent along all of a node's connections, and the recipients then forward the query along all their connections except for the one on which the query arrived. This process continues until queries have been forwarded a specified number of hops, at which point receiving nodes stop forwarding the query. This maximum time-to-live (TTL) assigned to every new query is specified as a global constant. When a node receives a query for

which it has matches (as detailed in Section 3.2.2), the node sends a response back to the node who generated the query. Note that although answering a query may involve opening a new connection, this does *not* change the set of connections along which the node forwards queries. Furthermore, a query is always forwarded regardless of whether the peer matched the query (so long as the TTL is nonzero). We refer to the node that originally sent a query as the query's **requester** and all nodes that return matches to the query as the query's **responders**.

It is easy to see that both the maximum TTL and the network's average node degree (the average number of connections per peer) play a major role in the exhibited behavior of a Kudzu network (or any other type of flooding-based network). Let c be the number of connections per node and k be the max TTL. Assuming a fairly random network structure, a query will have encountered c nodes after the first hop, $c(c-1)$ nodes after the first two hops (since queries are not forwarded backwards along the links they arrived), and $c(c-1)^{n-1}$ nodes after the first n hops for all $n \leq k$. Thus, a query will reach at most $c(c-1)^{k-1}$ nodes regardless of the total size of the network. Users, of course, would like their queries to reach the entire network, as this will return the largest possible set of results. Let's explore this possibility for a network of total size N. Solving for the TTL k gives the following:

$$
\begin{aligned}
N &= c(c-1)^{k-1} \\
\ln N - \ln c &= (k-1)\ln(c-1) \\
k &= 1 + \frac{\ln N - \ln c}{\ln(c-1)}
\end{aligned}
$$

Thus, for a modestly sized network of $N = 1000$ nodes with $c = 3$, this gives us $k \approx 8.4$, or 9 hops (on average) to reach every other node in the network. While this may seem manageably small, the number of nodes reached is exponential in k; this means that the corresponding query load induced on every node is also exponential in k for sufficiently large N. Thus, if we allow N to be arbitrarily large (which we want to do to be sure that the network will scale), minimizing k is paramount to keeping the network from being overloaded by query traffic. This is why a relatively low max TTL is important. In early versions of Kudzu, we experimented with removing the TTL and found the resulting network to be not only heavily loaded but extremely inefficient (described in Section 5.3).

3.2.2 Keyword Matching

One of the benefits of the gossip-like queries in unstructured networks such as Kudzu or Gnutella versus arguably more efficient queries in systems based on DHTs is that the former allows keyword searches, while the latter is restricted to exact lookups. Keyword searches allow for a great degree of flexibility in the way query matches are actually determined, which translates into more powerful search capabilities for the end user. In a keyword search, the recipient of a query receives a set of keywords and is free to use any arbitrarily simple or complex algorithm to determine the set of matching files. For Kudzu, however, we were primarily interested in the organization of the network and opted for the simple matching algorithm of matching a file to a query only when every keyword

in the query is a substring of the filename. This is also the standard approach used by some versions of Gnutella. For example, a query for "ring lor" will match a filename "lord of the rings", since both keywords are contained in the filename, but a query for "ring lore" will not. Matching is case insensitive and discards punctuation and all occurrences of standard stopwords (e.g., "the", "of") and topical stopwords (e.g., "mp3"). Both types of stopwords are common enough in practice that queries end up returning so many matches to render the query useless at the network's expense. Our complete keyword matching procedure is given in Algorithm 1 for query string Q, filenames F, and stopwords S. Note that the matching algorithm can be made arbitrarily complex without impacting other parts of the system.

Algorithm 1 Keyword Substring Matching

Require: Q, $F = \{f^1, f^2, \ldots, f^a\}$, $S = \{s^1, s^2, \ldots, s^b\}$
  $M \Leftarrow \{\}$
  $K \Leftarrow tokenize(Q) \setminus S$
  for all f^i in F **do**
    $add \Leftarrow true$
    for all k^j in K **do**
      if $not_substring_of(k^j, f^i)$ **then**
        $add \Leftarrow false$
        break
      end if
    end for
    if add **then**
      $M \Leftarrow M \cup \{f^i\}$
    end if
  end for
  return M

A straightforward implementation of the algorithm is effectively linear in the number of files on the node, since the number of keyword tokens per query is almost always small (less than 10). The same policy can be implemented more efficiently using more complex data structures such as suffix trees [26], but we did not focus our attention on optimizing local node operations and did not encounter any CPU-related bottlenecks. A variety of other matching policies may be employed as well (such as matching prefixes rather than substrings), but we found that keyword substring matching was perfectly sufficient for our needs.

3.3 Network Organization

We discussed in Section 3.2.1 how allowing queries to propagate without limit makes the network unscalable to large sizes, as adding new peers increases not only the global query load but each node's individual query load. Query load – specifically, the bandwidth necessary to handle all query traffic through a node – was the primary factor in Gnutella's shift from a fully decentralized network to one with many local, high-capacity hub nodes ('Ultrapeers') that handled the vast majority of query traffic for the entire network [30]. This system allowed queries to traverse a much greater portion of

the network without requiring large numbers of connections or excessive query hops through ordinary peers. However, this system placed a much heavier, involuntary burden on those nodes chosen to be ultrapeers: ultrapeers maintain a much larger number of connections to other ultrapeers than other nodes do (roughly 32). This compensates for the exponential TTL behavior by allowing the TTL to be set relatively low while still covering a very large number of nodes.

So far, we have framed the issue of network organization only by discussing the portion of the network that each query can cover. However, we note that node coverage is not the metric that we actually wish to maximize; in contrast, what is actually relevant is the number of matches retrieved. For a given query Q, there are likely to be only a small number of possible matches in the network, which furthermore are likely to be distributed across only a very small subset S of the network. We wish to maximize query **recall**, which we define as the ratio of the number of matches returned by the network to the total number of matches possible. The total number of possible matches, of course, will be equivalent to the number of matches returned if queries reach every node in the network. However, we can also achieve the optimal recall of 1.0 if each query only reaches those nodes that can actually match it. In fact, this is much better than the former 'optimal' case, since this latter case means that recall is maximized while communication overhead and bandwidth usage is minimized.

We thus consider the problem of network organization as finding a process of connecting nodes such that we achieve high query recall while permitting a low TTL value; in other words, while covering only a small portion of the entire network. We approach this problem by first defining a simple framework for these processes, which we refer to as organization policies.

3.3.1 Organization Policies

In order to evaluate multiple organization approaches easily, we separate policy from mechanism using the idea of an **organization policy**. An organization policy specifies how a node chooses its peer connections and consists of an optional initialization procedure and the following two operations:

- $chooseNewPeer(existingPeers)$: This operation takes as input the set of currently connected peers and returns a single new peer to which the node should connect, or *none* to stay with the current set of peers. The policy may use any algorithm to choose the new peer, although it must not be contained in the existing peer set.

- $chooseExcessPeer(existingPeers)$: This operation takes as input the set of currently connected peers and returns a single peer from $existingPeers$ from which the node should disconnect, or *none* to stay with the current set of peers. As with $chooseNewPeer$, there are no restrictions on how the policy chooses the peer other than it being one to which the node is currently connected.

Recall that the two values determining average query coverage are the max TTL and the average degree of each node. For any organization policy, increasing either of these values is guaranteed to improve (or not affect) recall, though at the expense of bandwidth. To be able to compare different approaches effectively, we choose to fix the average node degree across all approaches and observe,

for a particular approach, how the network operates across varying TTL values. Let MIN and MAX be two variables fixed across all nodes in the network (they may have the same value) and let C be the current set of connections for some node n. For any organization policy p, the following two statements are always enforced: if at any point $|C| < MIN$ (this could be due to a network failure, neighbors terminating connections, or any other reason), then the node will repeatedly call $chooseNewPeer$ at short intervals until $|C| \geq MIN$. Likewise, if at any point $|C| > MAX$, the node will repeatedly call $chooseExcessPeer$ until $|C| \leq MAX$. Since p may choose to return *none* for either of these operations, the size of C may remain outside of the range $[MIN, MAX]$ (depending on p), but will usually return to within the range (the particulars are left to the policy). Finally, we impose one additional restriction: peers that are newly connected are given a brief period of immunity from being disconnected. This is to prevent situations in which a node joins the network by way of an overconnected node only to be immediately disconnected before it can query for additional (less connected) peers. Given this framework in which organization policies operate, we now detail the specific policies that we explored for Kudzu.

3.3.2 Naive Policy

This represents the simplest 'realistic' policy, and was the one we initially applied to Kudzu. Simply choose peers randomly from available peers so as to maintain a valid number of links. Likewise, peers to disconnect are chosen randomly from the current set of connections. Real networks in which no particular organization is used will operate in a similar way, since peers will join at public entry points and then find other peers to add to the connection set through the entry node.

- *init*: Seed with one 'known' peer to form the first connection. This is akin to a real network in which a small set of public, permanently active nodes are hardcoded to act as potential entry points.
- $chooseNewPeer$: Choose an existing peer at random. If no such peer exists (that is, $|C| = 0$), return *none*. Otherwise, send a request to the chosen peer for MIN additional random peers. Randomly choose one of the returned set of peers that are not already in C and return it, or return *none* if no such peer exists, which may be the case if the chosen peer did not have MIN peers to send.
- $chooseExcessPeer$: Choose and return an existing peer at random.

3.3.3 Fixed Policy

A fixed policy is one in which we pre-select the node's connections and it simply attempts to maintain the connections specified. The given connections may be determined randomly or by some other process (we used several fixed policies in our tests, which we describe in Chapter 5). In realistic usage, of course, a policy such as this is useful for little more than bootstrapping, and it is unlikely that good general-case performance can be obtained from such a policy.

- *init*: Seed the policy with a list L of predefined peers.

- *chooseNewPeer*: Choose and return the next peer in L to which the node is not presently connected. If every peer in the list is presently connected, return *none*. This has the effect of simply populating the node's available connections with the peers that were initially given to the policy.

- *chooseExcessPeer*: Choose and return an arbitrary currently connected peer that does not appear in L. Return *none* if no such peer exists. Note that since a fixed policy ignores the settings of MIN and MAX, keeping the number of connections within this range must be done when the L is decided upon.

3.3.4 TF-IDF Ranked Policy

We now consider a more sophisticated organizational approach. An 'optimal' policy is one that chooses peers most likely to match future queries that the node sends. One way we can approximate an optimal policy is by choosing peers whose files most resemble our queries. If a peer's files match our queries exactly, then clearly that peer is a good neighbor to choose. We calculate these matchings by employing a **vector space model**. A VSM is an algebraic model for representing and comparing objects formulated as vectors of identifiers – in this case, the objects we represent are documents built from a node's files or queries (or potentially both).

Let's consider a node i. We define two 'documents' for each node: a **file store** F_i, which is comprised of all words in the node's filenames, and a **query store** Q_i, which is similarly comprised of all words in the node's queries. Let $W_i = F_i \cup Q_i$ and let $W = \bigcup_i W_i$ be the global set of word tokens. We can represent each F_i or Q_i as a vector $\vec{v}$ of size $|W_i|$ in which each entry v_w represents a specific word token in W_i. Given two document vectors $\vec{v_i}$ and $\vec{v_j}$, we can calculate their shared relevancy by using the cosine similarity metric:

$$\cos\theta = \frac{\vec{v_i} \cdot \vec{v_j}}{||\vec{v_i}||\ ||\vec{v_j}||}$$

This will be a value from 0 to 1 representing how relevant the documents are to each other: a value of 0 means they share no tokens in common and a value of 1 means they are comprised of the same tokens.

To calculate the vector weights, we use a well known statistical measure called **term frequency-inverse document frequency** [28]. TF-IDF calculates the importance of a word in a document or collection of documents, thus providing us with weights needed to determine the cosine similarity as given above. As the name suggests, TF-IDF attempts to account for two primary properties:

1. Words that appear many times in a document are more important than those that do not (term frequency). Clearly, if a term appears frequently, it is likely to be more relevant to the overall content of the document.

2. Words that appear in many documents are less important than those that are rare (inverse document frequency). If a term appears in most documents, it is likely a word that does not

impart specific information about those documents. This will include, for example, common language words that have nothing to do with content (e.g., ‘a’, ‘the’).

The term frequency is normalized to the document length, since we do not wish to assign higher weights to documents that are simply larger. Thus, for a term w_i in document d_j with frequency $f_{i,j}$, we have the term frequency as follows:

$$\mathrm{tf}_{i,j} = \frac{f_{i,j}}{\sum_k f_{k,j}}$$

For the inverse document frequency, we need to consider the entire document corpus $D = \{d_1, d_2, \ldots, d_x\}$. For a term w_i, we take the logarithm of the total number of documents over the number of documents containing the term:

$$\mathrm{idf}_i = \log \frac{|D|}{|\{d : w_i \in d\}|}$$

Note that assuming each node has complete information about all other nodes, the inverse document frequency is the same for any given term across all nodes. Finally, to calculate the TF-IDF, we simply multiply the two components:

$$\mathrm{tfidf}_{i,j} = \mathrm{tf}_{i,j} \times \mathrm{idf}_i$$

Using this to calculate the vector weights used in the cosine similarity computation, we end with a measure from 0 to 1 of the similarity between two file and/or query stores. Returning now to the problem that led to this discussion, we can use TF-IDF and cosine similarity to design our organization policy as follows:

- *init*: Bootstrap with a preset entry peer as in the naive policy.
- $chooseNewPeer$: For each potential peer, calculate the TF-IDF between this node's file store and the potential peer's file store. In using this node's file store, we are making the assumption that there is a correlation between a node's files and the queries it issues; work done in [4] suggests that this holds in practice. Rank the peers by TF-IDF and return the highest-ranking peer not already in the connection list. If no known peer exists not already in the connection list, return *none*.
- $chooseExcessPeer$: Repeat the ranking procedure described in $chooseNewPeer$ and return the lowest-ranked peer from the list of existing connections other than the peer that just arrived.

Note that in determining the ranking, we could have compared the potential peer's file store to the node's query store rather than its file store. While a stronger correlation is likely to exist between queries and the files of good potential peers, using queries has two significant downsides: one, most nodes have far fewer queries than files, and two, using such a scheme would require queries to have already been issued to see any benefit. Furthermore, file store information is likely to be

more current than query information, since while a node's query store may change rapidly as the node issues a sequence of queries, its file store will generally remain fairly consistent.

Note that this organization scheme requires some way to build a list of potential peers so that a useful ranking can be computed. In an ideal (but unrealistic) situation, all peers know about all other peers and can thus organize optimally. In a realistic situation, peers need a way to conduct exploration of the network. Our exploration consists of repeated applications of Algorithm 2 taking as input a list of known peers L. Initially, L is comprised of only the entry node.

Algorithm 2 Network Exploration

Require: $L = \{p^1, p^2, \ldots, p^n\}$
  $p \Leftarrow remove_first(L)$
  if $peer_online(p)$ **then**
    $add_last(p, L)$
  else
    return L
  end if
  $L' \Leftarrow request_peers(p)$
  for all p' in L' **do**
    if $p' \notin L$ **then**
      $add_first(p', L)$
    end if
  end for
  return L

Each time a new peer is found through this exploration, the node requests the new peer's file store to update its TF-IDF information. If the new peer's score is higher than the current best MIN connections, the node swaps the new peer in and ends its worst ranked connection. One aspect of this exploration that bears particular mention is the implicit incentive model that results from it. As nodes remain on the network for longer periods of time, they will explore more of the network and gradually improve their similarity scores with neighboring peers. This exploration continues even when nodes are not issuing queries, thus providing users an incentive to simply remain online while they explore more of the network.

3.3.5 Machine Learning Classifier Policy

We describe one final policy that represents a sophisticated but heavyweight approach to the peer organization ideas discussed in the previous section. This final policy, however, is much more difficult to implement in a real-world system. As such, we have not yet actually implemented this policy in Kudzu; some of the difficulties in applying this policy to a real system are discussed later in Section 6.1.1.

The TF-IDF ranking, while much more sophisticated than random selection, is still premised on fairly simple relationships between document sets. Furthermore, evaluation of peer connections requires transferring entire sets of file store tokens, which may be nontrivial in size. We can improve on these problems by turning to full-blown machine learning classifiers. Another way of stating the network organization problem is that, given only a small amount of input information (like file store

tokens, but preferably not the peer's entire file store), we want to create a classifier that determines whether a given peer is a suitable neighbor for the future.

Rather than computing the TF-IDF on entire file stores and ranking potential peers using the results, we'd like to determine a small set of keywords that predicts neighbor suitability. Note that these keywords may not correspond to simple file-file or query-file matches as they do in some capacity when using TF-IDF. For example, suppose a node is issues queries for Star Wars content such as "star wars", "death star", and "star destroyer". Suppose also that the node is evaluating two potential peers, each of which is advertising a single file. The first is offering "the jedi handbook.txt", while the second is offering "stars for astrophysicists". Although the keyword matches all point to selecting the second peer as the neighbor (due to the matches for "stars"), humans can immediately see that this is wrong. This is because we have learned from the node's previous queries that the node is searching for Star Wars content rather than astronomy content; as a result, we can see that 'jedi' is a better predictor of good neighbors than 'star'. We can try to build a machine learning classifier that learns these types of relationships.

Formulation as a Classification Problem

One approach we can take is like that described in [5]. For training a peer classifier, we first need a way to formulate a peer as a data point. Let each peer i be described as a feature vector of binary features where each binary feature b_x represents whether the word token $w_x \in W$ appears in the peer's file store:

$$\vec{p_i} = \{b_1, b_2, \ldots, b_k\} \mid b_a \in \{0, 1\}$$

As in TF-IDF, we consider the complete set of word tokens W to be the set of all tokens encountered. Given a set of these data points $\vec{p_i}$, the objective is to learn a binary class label y_i specifying whether the peer p_i is a good or bad neighbor for the node in question.

As with any supervised machine learning algorithm, we need a training set (that is, a set of instances for which the class label is known) in order to build a classifier for unknown instances. The easiest way to empirically determine class labels for particular peers is to simply interact with them by sending queries – if a potential peer matches many of the node's queries, the peer is probably a good neighbor and can be assigned a positive class label, while peers that do not provide any benefit for the node can be assigned negative class labels. Once a suitable corpus of training data is gathered from interaction on the network, these points can be fed into an off-the-shelve machine learning classifier algorithm.

Support Vector Machines

Support Vector Machines (SVMs) were found in [5] to perform well on this task while avoiding excessive overfitting to the data. Support vector machines operate by taking a set of points in an n-dimensional space and finding the separating hyperplane that separates positive from negative class labels (assuming a binary decision problem such as the one here) while maximizing the distance from the hyperplane to the instances on either side – this is known as 'maximizing the margin'.

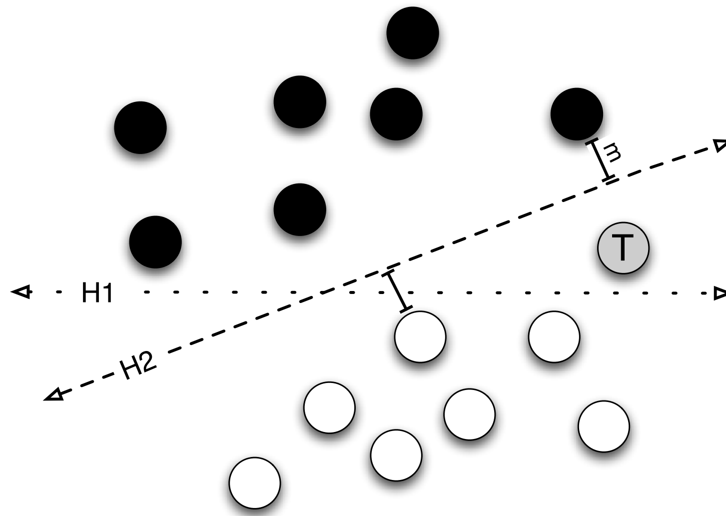

Figure 3.1: A non-optimal separating hyperplane H1 and an optimal separating hyperplane H2 with margin m. Test point T is misclassified as black by H1 but correctly classified as white by H2.

This is an optimization problem which can be solved computationally using quadratic programming techniques. An example of an optimal separating hyperplane for a binary decision problem in two dimensions is shown in Figure 3.1. However, SVMs are robust even in extremely high dimensional spaces, which is useful to our problem because the total size of the word corpus (which corresponds to the dimensionality of our data) is likely to be quite large. For this reason, SVMs are frequently used in many types of text classification problems.

Feature Selection

Once we have a classifier for the word features $W = \{w_1, w_2, \ldots, w_k\}$, we can use feature selection to choose a small subset or W containing only the most useful features. This is the process of selecting features to gradually minimize the classifier's error. For instance, one feature selection procedure we could choose to use is greedy forward fitting (FF): on each iteration, FF simply greedily chooses the next feature $w_i \in W$ such that the subsequent error of the classifier is decreased as much as possible. Using a feature selection algorithm allows us to create a classifier that performs comparably to one using every feature while only using a small fraction of the total feature set.

This is of particular interest in our case because larger feature sets mean larger amounts of information that need to be exchanged between peers in order to predict whether a connection is likely to be fruitful. Given the final classifier (which uses only a small set of word features $F = \{f_1, f_2, \ldots, f_i\} \mid f_j \in W$), to classify a potential peer we only need to know the binary values of each f_a. In other words, to represent the potential peer, we need only know whether each of the representative keywords appears in the potential peer's file store. Once we have this information, we can feed the feature vector into the classifier, which outputs the class label telling the node whether

it should or should not connect to the potential peer. We can thus formulate an organization policy using an SVM classifier as follows:

- *init*: Gather training data by participating on the network. Train a classifier using all features W (W is likely to be quite large), then use feature selection to select a useful but much smaller subset F.
- $chooseNewPeer$: For each potential peer p_i (found through exploration, as in the TFIDF policy), request the binary values of each feature in F for peer p_i. Store the result into a feature vector $\vec{p_i}$. Feed this data point into the classifier. If the classifier outputs a positive class label, return p_i. Otherwise, move onto p_{i+1}.
- $chooseExcessPeer$: For each peer in the list of existing connections, simply repeat the above procedure and return the first peer for which the classifier returns a negative class label. If there are none, the node could either retain all its connections or select one at random to remove.

One of the important things to note about this approach is that the discriminative keywords identified are specific to the peer in question and may be completely different on another node performing the same algorithm. Returning to our Star Wars example, the classifier may well determine that 'jedi' is a good feature for that peer, even if it is a poor feature for other peers.

3.4 Download Behavior

We modeled the process of conducting file transfers in Kudzu after the highly successful model employed by BitTorrent. BitTorrent's high performance largely comes from the ability to leverage the bandwidth of many peers downloading or sharing the same file. The primary difference in Kudzu we do not have a tracker like that used in any BitTorrent network.

Due to the similarity of BitTorrent's download model, we reuse some of BitTorrent's terminology in describing the download process. For a given shared file, a **swarm** is the set of all peers participating in the file transfer, including both uploaders and downloaders. A **seed** is a peer that is sharing the entire file, while a **leech** is a peer that is downloading the file without sharing. All other peers involved in the file transfer have downloaded a portion of the file (which they upload to peers who do not have that portion) while downloading the remaining portions from other peers – note that this means two peers may be both uploading and downloading from each other.

Since BitTorrent networks operate only on a single file being shared, it does not exactly map onto the Kudzu network. Instead, each swarm in a Kudzu network functions as an overlay network on top of the main Kudzu network. An example of this organization is shown in Figure 3.2.

3.4.1 File Identification

Files in a Kudzu network are located using keywords searches, but keywords (or even the exact filenames returned) do not uniquely identify desired files. BitTorrent deals with this issue with the

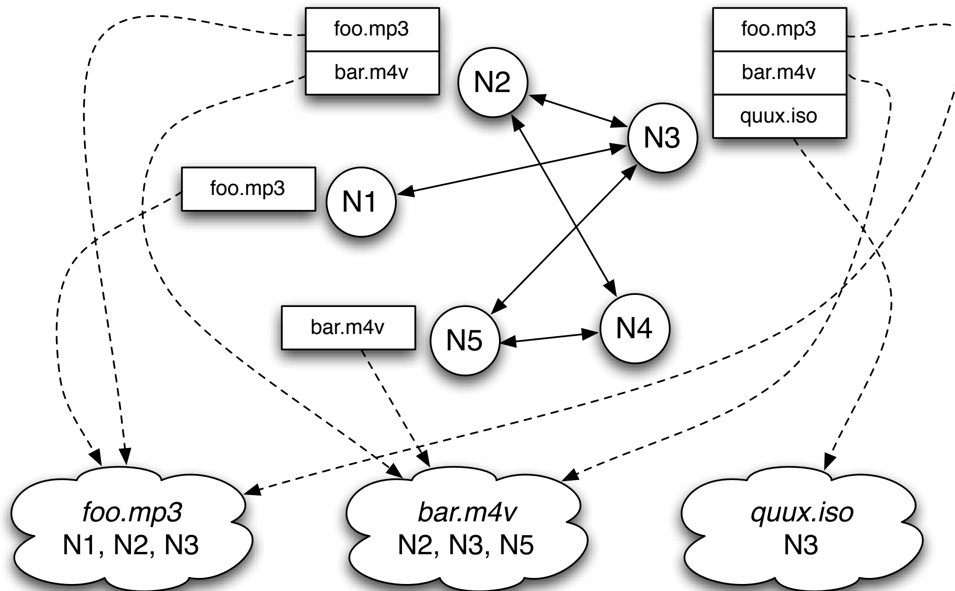

Figure 3.2: A Kudzu network of 5 nodes containing 3 download swarms. Solid lines indicate peer connections, while dotted lines indicate swarm connections.

use of .torrent files, which contain a unique signature for the file to download. Kudzu calculates unique signatures as well, but allows users to locate file signatures using keyword searches in the network itself rather than requiring an external search engine for `.torrent` files. Each node calculates an Adler32 [9] checksum to uniquely identify each of its files (though any similar checksumming algorithm, e.g. CRC32, could be used). Checksums are computed based only on the file's contents; thus, files that have been renamed will still be recognized as the same file.

For each query response that returns to a node, the node stores the responder's IP address and the names and checksums of the matched files. If the node is already storing responses that contain one or more of the same files, the IP addresses are stored together – this gives a record of all responders that contain that particular file. Actually starting the download is left to the user, which is accomplished by choosing the desired filename. This tells the node which checksums is desired, at which point the node can connect back to all the nodes who responded with that file and begin the download.

3.4.2 Chunks and Blocks

In order to leverage the bandwidth of many users transferring the same file, it is important to be able to both download and upload simultaneously from multiple other peers. Similar to BitTorrent, Kudzu facilitates this by breaking up a shared file into multiple **chunks**, each of which is further broken up into multiple **blocks**. The primary distinction between the two is that chunks are the

smallest units that are advertised by peers as ready to be uploaded, while blocks are the actual atomic units of transfer. The actual sizes of a chunk and a block are constants that may be set arbitrarily but manifest several important tradeoffs. Smaller chunks have the benefit of allowing downloaders to begin uploading rapidly (since data may be uploaded with finer granularity), but at the price of bandwidth overhead that is linear in the total number of chunks. Larger blocks reduce overall bandwidth usage since fewer messages need to be exchanged, but can pose problems with nodes on slow or congested network connections – since blocks are the smallest units of transfer, transferring a large block from a slow peer may cause the download (or single chunk) to take significantly more time than if a smaller block size was used (which would result in a lesser request to the offending peer). We set our defaults values at 16 kilobytes per block and 512 kilobytes per chunk, which are typical values in a BitTorrent swarm.

3.4.3 Swarms

An active download consists of a single manager that delegates download chunks to multiple download streams, each of which requests blocks from a single other peer. Download streams are given their own connections from the rest of Kudzu to avoid slowing down query traffic and do not count towards the node's current number of connections. Optimizing the process of downloading involves several primary considerations:

- Since nodes can upload chunks that they have completed downloading, it is in the network's best interest to ensure that peers are downloading different chunks, thus allowing them to subsequently share those chunks with each other. Clearly, downloading chunks sequentially is a poor strategy – a much better strategy is to download the chunk that the fewest members of the swarm already have. For ease of implementation, however, we opted for pseudo-random selection. Truly random selection is impossible, because from a given peer we can only download a chunk that the peer already has. We deal with this by first choosing a random point in the file as if the peer already had the entire file, then choosing the peer's next available chunk in a round-robin fashion.

- Subject to the manner of chunk selection, chunks that are already in progress should be prioritized due to the reasons mentioned in Section 3.4.2. Since chunks are broken up into blocks, we can assign multiple download streams to a single chunk, thereby hastening its completion and subsequent upload capability. Thus, we always assign a stream to an existing chunk transfer (if possible) before applying the random process described above.

- For reasonably fast connections with high round-trip times (which are common in global networks), the small size of a block transfer is likely to be insufficient to saturate the link's bandwidth-delay product. In these cases, the amount of data transferred can be increased dramatically by allowing multiple unacknowledged block requests to a single peer at once – this is called pipelining. Pipelining is an example of a simple parameter (the number of simultaneous requests allowed) that can have a major impact on performance, as an incorrect choice can either waste or underutilize large amounts of bandwidth.

Chunks themselves are represented simply as bits corresponding to having or lacking each chunk. Our chunk selection algorithm is given in Algorithm 3 for downloading from a peer with chunks C given already downloaded chunks D and in-progress chunks P (all containing n bits). Once a stream has been assigned a chunk, it sequentially (in concert with all other streams assigned to the chunk) downloads all blocks in the chunk, then requests another chunk to download from the manager and repeats the process. New swarm members that are added after the download has started are assigned chunks upon arrival and operate in exactly the same process.

Algorithm 3 Download Chunk Selection

Require: $C = \{c^1, c^2, \ldots, c^n\}$, $D = \{d^1, d^2, \ldots, d^n\}$, $P = \{p^1, p^2, \ldots, p^n\}$
  $C \Leftarrow C \setminus D$
  if $|C| = 0$ **then**
    return *none*
  end if
  $S \Leftarrow C \cap P$
  if $|S| > 0$ **then**
    return S_1
  end if
  $x \Leftarrow rand_range(1, |C|)$
  for $i = x$ to $|C|$ **do**
    if $C_i = 1$ **then**
      return C_i
    end if
  end for
  for $i = 1$ to x **do**
    if $C_i = 1$ **then**
      return C_i
    end if
  end for

3.4.4 Gossip

Several types of communication occur over a download swarm besides the actual transmission of file data. We refer to these ad-hoc communications as gossip. To facilitate continued upload/download activity, active swarm members periodically exchange their chunk sets, thereby updating all other nodes on new chunks that are available for download. This is where we pay a price for smaller chunks, as more data must be transmitted to account for a larger total number of chunks. Downloaders in the swarm also periodically choose another peer in the swarm at random and exchange the known swarm peers. Thus, any peers in the swarm known to one of the gossip participants will be relayed to the other. The long-term effect of this gossip is that a node only needs a query to reach a single member of a download swarm in order to eventually discover everyone who has the file. This allows us to be more lax with the query TTL while not compromising swarm performance.

It is also interesting to note that initially, every shared file on the network is effectively its own download swarm with the host peer as the single seed and no other participants. This means that the entire Kudzu network may end up with multiple active swarms for the same file depending on where

queries originate and reach. A positive effect of our swarm gossip is that if any node's query reaches members of multiple swarms for a particular file and then begins downloading it, the new node forms a link between the two swarms and effectively merges them into a single, more effective swarm. As gossip occurs, the new node will gather the swarm members from both individual swarms, and each of the two swarms will learn of the peers in the other. This automatic merging is an improvement over BitTorrent, where many swarms may exist in isolation for the same file – typically, some of the swarms are unsuccessful in maintaining a critical mass of peers and ultimately go dormant, resulting in useless torrent files leading to no seeds.

3.5 A Distributed Test Framework

Gathering empirical data on a large-scale distributed system is a difficult problem, especially in systems like a P2P network where its behavior is heavily dependent on the actions of its users rather than simply the system's design. A traditional approach to evaluating large scale networks is creating a discrete event simulator, which can then be used to model very large networks locally. The ability to arbitrarily scale is certainly a draw towards using a network simulator. However, simulators also suffers from several shortcomings. One of the most important is that a simulator cannot easily model all network conditions – the number of variables involved are numerous and are often interdependent. For instance, users on the same local area network will experience the network quite differently relative to each other as they will to other users in the wide area. These types of situations make accurate simulation quite difficult. Furthermore, in a P2P system like Kudzu, the scarcest resource in the system is bandwidth, and accurate bandwidth measures between machines over a large and unpredictable network such as the Internet are difficult to employ in a simulator.

Since we opted to forgo a simulator to run our experiments, we designed a test framework to run a real network using a large testbed on a wide-area network. The obvious testbed for this is PlanetLab [21], a global network of roughly 1000 machines spread across the world available for running distributed system experiments. Running live tests on PlanetLab solves the problem of unrealistic network conditions and subjects our system to all the perils (latency, unresponsive peers, etc.) that a deployed P2P system is subjected to in a live deployment.

3.5.1 Simulating User Behavior

Running on a real wide-area network is only one of the major hurdles in running useful tests of the system. The other is that simulating user behavior is extremely difficult. Since our system has (as of yet, at least) no actual users, the only way to measure statistics is with simulated users. User querying behaviors and shared file stores are impossible to model in a useful way without working from preexisting data. Thus, rather than attempting to model users from scratch, we take data captured from an actual network in the past and replay it on the testbed, thereby subjecting the system to actual user behavior observed on a similar network.

The dataset we use is a 2005 trace of a Gnutella network captured by Goh et al [12] that contains information describing roughly 3500 unique users observed on the network over a period of 3 months.

For each user, the dataset contains two sets of information: the set of queries issued by the user, and the complete set of files shared by the user. Each query consists of a set of keywords and the timestamp at which the query was issued. Each file consists of a filename and a filesize. The dataset also contains some miscellaneous information such as user connection speed (e.g., dialup or DSL) and the user's Gnutella client software.

3.5.2 Replayer Design

Deciding how to replay the Gnutella dataset for Kudzu posed several design questions. One problem was the actual number of users in the datasets, which was significantly greater than the number of machines available in our testbed. Before discussing our approach to the problem, we give a few definitions. A **virtual user** refers to a single logical user (that is, a set of files and queries) running on some testbed machine. A **real user** refers to an actual testbed machine communicating across the network with other machines. There is generally (but need not be) a one-to-one correspondence between virtual users and real users. We considered several ways to account for this issue:

1. Assign a single random virtual user to each available real user and simply replay as many virtual users as the testbed allows. This is the most straightforward option and will not have unexpected side effects, but does not fully exercise the dataset. If the testbed is not large enough, too little data will be replayed to generate meaningful results.

2. Merge multiple virtual users into a single virtual user (by merging the file and query sets) and assign the result to a real user. This would allow us to exercise the entire dataset, but is also likely to interfere with organization policies, because the net result will be that the original (pre-merge) users will compete for the best peer connections to match their queries. Keeping the virtual users separate ensures that connections are established only based on the activity of the real-life user from which the virtual user was captured.

3. Run multiple virtual users as distinct entities on a single real user. This would also allow us to exercise the entire dataset, but is likely to have unintended side effects of running multiple clients on a single machine. Virtual users that are highly active will negatively impact the performance of other virtual users on the same machine. Furthermore, assigning multiple users to a single machine results in greater overall disruption when a machines fails or acts unexpectedly.

We ultimately opted for option 1 after deciding that the size of the user subset we could replay was sufficient for running useful experiments (see Chapter 5 for the results of our experiments). In this case, the simplest approach is also the most realistic, as virtual and real users become effectively the same entity.

Another issue was the length of time covered by queries in the dataset (roughly 3 months). We obviously could not afford to play back the dataset in realtime, so we modify the timestamps of all queries in the dataset by speeding up time by a large multiple. This means that the sequential ordering of queries is still correct while allowing us to run large-scale experiments in the span of

only a few minutes or hours rather than multiple months. The choice of time multiple is a tradeoff between the amount of time required and result fidelity, since larger multiples will cause the network to be significantly more congested during testing. Our particular dataset, however, is fairly sparse (in that most users only issue a few queries), so congestion was never an issue during testing.

In practice, the actual coordination of testbed nodes to replay the dataset presents a number of additional hurdles to overcome. This is due both to the large number of machines we wish to coordinate and to the general unreliability of the PlanetLab testbed. We discuss the ways we dealt with these types of problems in Chapter 4.

3.6 Summary

Kudzu is designed to be an efficient, scalable P2P transfer system that merges successful aspects of both Gnutella and BitTorrent-like systems while remaining completely decentralized. Furthermore, it improves on their basic design by employing adaptive behavior to intelligently organize the network. In order to evaluate Kudzu under real-world settings, we also designed a test framework that replays real user data on Kudzu using a live network, thereby introducing all the variables normally encountered in a real-world network setting.

Chapter 4

Implementation: The Kudzu Client

We have implemented a Kudzu client according to the specification described in Chapter 3, as well as the test harness for running experiments on our client. Since a Kudzu network is comprised entirely of clients with no higher-level coordination required, the client itself implements all aspects of a Kudzu network. Our implementation of the client is a Java program of roughly 3000 lines.

The client is started on the command line and is provided a directory from which to share files and download into and a hostname or IP address of an existing Kudzu peer to connect to. If an existing peer is not provided, the client starts but has no connections, and thus will not be part of any greater network until other peers connect to it. Once the client is started, it presents a simple command-line interface to the network controlled primarily through the following three commands:

- *query [keywords]*: Issues a query for *[keywords]* to all connected peers. Since there is no upper limit on the amount of time that may pass before matches returned (and no matches may ever occur), this operation has no immediate effect visible to the end user.

- *responses*: Displays all responses that have been received for previously issued queries. For each query that has received matches, a list of the matches received is outputted along with a download id for each file match.

- *download [download id]*: Initiates a download of the file identified by the given download id. The id is provided to the user by issuing the *responses* command. Once the download starts, progress measurements are outputted until the download is complete.

An example session in which a client issues a query and downloads a file from two peers is shown in Figure 4.1. Note that a small amount of waiting (a few seconds) is implied in between issuing the query and checking the result set to allow for queries to reach matching peers.

4.1 Communication Framework

The most important aspect of most P2P systems is the communication that occurs between peers, and Kudzu is no exception. Peers in a Kudzu network are constantly exchanging messages with each

```
$ kudzu -d sharedir -n planetlab1.williams.edu
Starting node and connecting to planetlab1.williams.edu...
You are connected to Kudzu.
> query coaster
Sent request for ‘coaster’ to peers.
> responses
Query ‘coaster’:
    id 0: ‘roller_coaster.mp4’ (3907036 bytes):
        Peer planetlab2.williams.edu
        Peer planetlab1.williams.edu
    id 1: ‘glass_coasters.mp4’ (2688476 bytes):
        Peer planetlab3.williams.edu
> download 0
Downloading ‘roller_coaster.mp4’ (3815 KB)...
Received 464 of 3815 KB (475 KB/s, 2 peers)
Received 1432 of 3815 KB (695 KB/s, 2 peers)
Received 2536 of 3815 KB (482 KB/s, 2 peers)
Received 3815 of 3815 KB (612 KB/s, 2 peers)
Validating file contents...file validation succeeded.
Download complete of ‘roller_coaster.mp4’ (average speed 514 KB/s).
```

Figure 4.1: User interaction with the Kudzu client.

other, so a robust and efficient communication framework is extremely important to ensure that a Kudzu network exhibits both high performance and low overhead. This section discusses how the communication in a Kudzu network is managed.

The communication internals of Kudzu went through three distinct iterations, according both to our requirements and problems we identified along the way. With each new version, the primary consideration was improving efficiency (that is, reducing the number of bytes transferred over the network), but many of our changes brought about other improvements as well. We describe each implementation of the communication system here.

4.1.1 Java RMI

Early implementations of Kudzu communicated with other nodes using Java Remote Method Invocation [32]. Under RMI, servers contain registries that publish Java objects to a public interface, which allows remote machines to obtain references to those objects and invoke methods upon them. Using this API in Kudzu, each peer published a single `Node` object that contained methods to perform all actions required by other peers. Under this model, for a peer p_1 to communicate with a peer p_2, p_1 needed only to fetch the object reference from p_2 and could then call on it whatever methods are required (e.g., `sendQuery`) without ever again explicitly dealing with network operations.

Our RMI implementation was motivated primarily by simplicity. RMI is extremely clean from a programmatic perspective, as peers are represented by logical objects, which parallels the actual communication that occurs. Since the underlying network activity is almost completely abstracted away, we could focus only on the core network logic without worrying about communication details.

However, while the RMI implementation was functional, it had several major flaws. The most serious was that it was inefficient; the price of RMI's generalized abstraction is significant overhead both on the network and on the CPU. RMI layers additional abstractions on top of the normal overhead of Java serialization (which alone is already significant). Another problem was that due to the high level that RMI operates at, it was difficult to tell how nodes were using resources such as bandwidth and file descriptors for socket connections. In practice, we found that RMI also caused problems in tests when we attempted to aggregate results from many nodes at a single machine. Lastly, RMI's inability to easily make asynchronous calls meant wasted overhead waiting for messages to finish a round trip when the response was to be ignored anyway.

4.1.2 Java Serialization

Once we identified how RMI was ill-suited to Kudzu as described above, we redesigned the client to use Java serialization through regular sockets. This gave us much greater control over the lower-level networking details at the expense of added complexity. However, with this added complexity we were able to add the capability to easily pass messages both synchronously and asynchronously (see Section 4.1.5 for details). Since we then needed to explicitly represent messages to be passed (unlike in RMI where the communication was implicit), we defined a class for each message type, thus making connections between peers simply streams of message objects passing back and forth. Java itself handles all the details of writing the objects to the network. While convenient, however, this meant that we had limited control over the amount of information actually sent over the network. Serialized objects contain a significant amount of metadata which the programmer has no control over; while this may only amount to overhead of tens of bytes per object, the primary operation in a Kudzu network is exchanging messages, and peers may be handling hundreds of messages per second. Furthermore, since Kudzu messages (and messages in most similar P2P systems, for that matter) are flat, one-shot communications that have no extended lifetime, the benefits of full-blown Java objects (such as inheritance) went unused.

4.1.3 Protocol Buffers

For the final version of the messaging framework, we wanted to use a format that allowed tight control over the underlying wire format while being as compact as possible. We settled on protocol buffers [15], a low-level message interchange format developed for internal use and subsequently open-sourced by Google. The protocol buffer wire format generates messages much smaller than their equivalent Java counterparts with comparable or better CPU usage [13]. Protocol buffers operate by taking as input a `.proto` file defining one or more message types and compiling it into standard Java class code that reads and writes the message types over the network. An excerpt of Kudzu's `.proto` file that defines query messages is shown in Figure 4.2.

The wire format of protocol buffer data is highly tuned to efficiency. Unsigned `ints` are encoded as `varints`, in which the top bit of each byte flags whether the entire int has already been read or whether it continues on into the next byte. This means, for instance, that the values 0 to 127 may be encoded using only a single byte. Signed ints are encoded using ZigZag [14] encoding, in which

```
message QueryRequest {
  required string keywords = 1;  // query keyword string
  required bytes requesterAddress = 2; // IP address of requester
  required int32 ttl = 3;  // query's remaining number of allowed hops
}
```

Figure 4.2: One of Kudzu's protocol buffer definitions.

the sequential unsigned integers are used to encode 0, -1, 1, -2, 2, and so forth. This saves bytes when encoding ints whose absolute value is low. Complete protocol buffer messages are also quite efficient and include almost no metadata. A message is encoded as a series of key-value pairs, one pair for each field of the message (e.g., 'keywords' in the above). Keys are encoded as three bits specifying the value type ('int32', 'string') and then a `varint` specified in the `.proto` file used to signal the particular field. This means that for fields with identifying values needing no more than 5 bits (effectively the first 16 fields of each message), the field key is contained in a single byte.

Returning to our example message definition, we have three fields and have specified the identification values 1, 2, and 3 (the trailing numbers are *not* value assignments). This means that on top of the actual message data, we have only 3 bytes of overhead. Suppose we have a query for 'beatles'. Given this 7 byte string (plus 1 to delimit the length with a 1 byte `varint`) and assuming a standard 4 byte IP address and 1 byte TTL, the entire size of the message is $3 + 8 + 4 + 1 = 16$ bytes. Note that since this represents only the Kudzu payload, the amount of data transferred over the wire is actually dominated by TCP and IP; assuming a standard TCP and IP header of 20 bytes each, the total number of bytes required to send the message will be 56.

In addition to the efficiency and control afforded by protocol buffers, they have the added benefit of being language and implementation agnostic. Given Kudzu's small `.proto` file, a third-party could with fairly minimal difficultly write a fully functional Kudzu client in any language for which there exists a protocol buffer compiler (at present, this includes Java, C++, and Python).

4.1.4 Kudzu Message Encoding

Since protocol buffer messages are simply key-value pairs with no other identifying information, it is difficult to determine what type of message has actually arrived after it's been read. To deal with this, we encapsulate each message in a common wrapper message. The only field a wrapper message is guaranteed to contain is a one byte `int` specifying the message type. Each message type that requires specific information has a payload message defined (like the one in Figure 4.2), and the wrapper message has an optional field for each payload type. The type indicator `int` in the base message signals not only the message type, but also which payload is contained in the message (if any). Since optional fields that are not provided add nothing to the binary form of a protocol buffer message, defining these optional fields does not increase the size of messages at all. The full protocol buffer specification of the base message class is shown in Figure 4.3.

The last encoding issue is that protocol buffer messages are not self-delimited; that is, they do not provide a way to determine when a complete message has been a received. This is problematic for a

```
message Message {
  required int32 type = 1; // type specifying which (if any) content field is filled
  optional int32 id = 2; // message id to identify a response message
  optional BlockRequest blockRequest = 3;
  optional BlockResponse blockResponse = 4;
  optional ChunkSetRequest chunkSetRequest = 5;
  optional ChunkSetResponse chunkSetResponse = 6;
  optional ErrorResponse errorResponse = 7;
  optional FileStoreResponse fileStoreResponse = 8;
  optional HostRequest hostRequest = 9;
  optional HostResponse hostResponse = 10;
  optional PeerExchangeRequest peerExchangeRequest = 11;
  optional PeerExchangeResponse peerExchangeResponse = 12;
  optional QueryRequest queryRequest = 13;
  optional QueryResponse queryResponse = 14;
}
```

Figure 4.3: Protocol buffer specification of base container message.

network connection on which bytes are continuously arriving because the receiver cannot determine when one message ends and the next begins. To deal with this, we simply append a `varint`-encoded byte length onto the front of each message. Since most messages (with the notable exception of download block messages) are less than 128 bytes in size, this header is generally only a single byte.

In summary, the process of receiving Kudzu messages from a peer is as follows:

1. Read a single byte at a time until a full `varint` is received (this will never be more than 4 bytes). Call the value of this `varint` n.

2. Read n additional bytes and parse them into a `Message`. Call this message m.

3. Read the type field of m. Retrieve the appropriate payload field as specified by the type (if any).

4. Handle the message payload appropriately. Optionally send a response message.

5. Repeat while the peer connection remains open.

4.1.5 Connection Management

In addition to handling the transfer of specific messages using protocol buffers, we needed to manually manage peer connections to provide useful response-request. Each two-way peer connection is handled as a pair of one-way connections. The sender in each of these connections sends data on the socket whenever needed and optionally waits for a response message to appear on the same socket. The receiver client runs a background thread that consumes incoming input from the socket, handles it, and sends response messages if needed. Since a connection of this type exists in either direction between the two peers, clients are always sure whether they are receiving requests or responses

to their own requests from the other peer. To avoid wasting time and bandwidth on constantly reestablishing connections, connections between peers are left open continuously until one of the peers leaves or intentionally terminates the connection for some other reason.

Our client provides the following three types of communication calls over peer connections:

- **Synchronous requests**: a thread sends a message and blocks until a response is returned, at which point the thread is woken up and returns the response. If a specified amount of time passes before a response is received, the thread is woken up and a peer timeout exception is thrown. If an error response is received rather than a response of the expected type, a peer error exception is thrown. This type of call is used for operations in which a response is needed before the thread can proceed, such as requesting new peers from an existing peer so that they can be added as new connections of the node.

- **Asynchronous requests**: a thread sends a message and returns immediately without waiting for a response. If a response is ever received to the request, it is discarded. This is used for operations such as forwarding queries in which no response is expected and errors can safely be ignored.

- **Asynchronous requests with callbacks**: a thread sends a message and passes a callback function specifying two operations: a standard response handler and an error handler. The thread returns immediately without waiting for a response. When a response arrives, the callback function is executed on a separate thread (the operation executed depends on whether an expected or error response was received). If the timeout runs out before a response has arrived, the error handler is executed on the separate thread. This is used for operations in which a response is expected but the order of responses is irrelevant and nothing is waiting on the result. For instance, this call type is used for fetching many download blocks concurrently.

Since a P2P application like Kudzu is highly concurrent, many communications may be occurring over a single peer connection simultaneously. This means that for multiple threads waiting on responses, the response messages may arrive in any order. We deal with this via an additional integer field defined in the base protocol buffer message definition. When a request message is sent out over a connection, the node first stores a message id into this field. The recipient reuses this id when constructing the response message. The requester maintains a map of message ids to the threads waiting on them; thus, it needs only to check the id on incoming responses to determine which thread should handle the response.

Request ids are assigned in round-robin style from 0 to 127 so that they always fit in a single `varint` byte. Since ids are only intended to be unique on a per-connection basis, the only assumption we make in restricting ids to this small range is that no node will ever have more than 128 outstanding message requests to a single node – in practice, this assumption has never been a problem. Furthermore, for asynchronous requests without callbacks, the request id can be omitted entirely.

4.2 Message Types

Kudzu peers exchange information using 16 distinct message types. We give a brief description of the purpose and contents of each message type here. The complete protocol buffer specification of all message payload types is given in Figure 4.4.

- **ping**: a message with no payload that simply returns another ping message. Used to verify that peers in a download swarm are still alive to avoid gradual accumulation of disconnected peers in the list.

- **backconnect**: a message with no payload that signals to establish the second half of a two-way connection back to the message sender. Typically the first message sent after a node decides to add another node as a query neighbor.

- **disconnect**: the counterpart of a `backconnect` message: signals to terminate the second half of a two-way connection back to the message sender. Sent after a node decides to remove one of its query neighbors.

- **block_request**: a request for a block of a file. The payload contains the checksum identifying the file in question and the block's byte offset from the start of the file. This type of request is how all download blocks are fetched.

- **block_response**: the response to a **block_request** message. The payload contains the binary data of the requested file block.

- **chunk_set_request**: a request for an updated listing of all chunks of a file that the recipient has in full and is ready to upload. The payload contains the checksum of the file in question. This is used by peers in a download swarm to learn about chunks that other peers have obtained so that they can be subsequently fetched from those peers.

- **chunk_set_response**: the response to a **chunk_set_request** message. The payload is a bit string whose length is the number of chunks in the file and each bit set to 1 indicates that the peer can upload that chunk.

- **error_response**: a generic response that can be sent in response to any request message indicating that something unexpected happened. The payload is a string describing the error that occurred. For example, an error of this type will be sent if a block is requested of a file that the node does not have.

- **filestore_request**: a message with no payload that signals a request for a listing of all words (and associated frequencies) in the recipient's set of filenames.

- **filestore_response**: the response to a **filestore_request** message. The payload contains a single string consisting of concatenated substrings for each filestore word token. Each substring consists of the word, a space, the frequency, and finally a newline.

- **host_request**: a request for a random assortment of the target peer's neighbors (not including the requester, of course). The payload contains an int specifying how many new neighbors are desired. This message is used by node organization policies to populate their neighbor sets.

- **host_response**: the response to a **host_request** message. The payload contains up to the number of requested peer addresses (but may contain fewer).

- **peer_exchange_request**: a request for all known peers in a download swarm. The payload contains both the checksum of the file in question and the sender's own swarm set so that the receiver doesn't need to make the same request in reverse. This effectively syncs the swarm between the two peers.

- **peer_exchange_response**: the response to a **peer_exchange_request** message. The payload contains the addresses of all known peers in the specified swarm.

- **query_request**: either a new request the sender is issuing to the network or a request that the sender received and is now forwarding on. The payload contains the query's keywords, the address of the peer that originally generated the query, and the remaining number of allowed hops. For the purposes of testing, we also insert a randomly generated int into every newly generated query to be able to distinguish duplicates. This is because in practice, users often send the same query several times in a short time period, and we want to be able to tally duplicate queries without simply discarding them. In a deployed network, however, the id field is unnecessary.

- **query_response**: this message type is somewhat different from the other response messages because although logically it is sent as a response to another node's query, it is sent as a request itself. This is because query requests are asynchronous; the requester has no idea of when (if ever) to expect responses, so there is nothing waiting on query responses to quickly return. The message payload contains the keywords of the query being answered and an arbitrarily-sized list of matches to the query. Each match is an embedded message consisting of a filename, a filesize, and a checksum.

4.3 Test Framework

In addition to the Kudzu client itself, we also implemented the wide-area testing framework described in Chapter 3. Given the dataset described in Section 3.5.1, the goal is to replicate the conditions in the dataset as closely closely as possible on a set of real machines. The framework consists of two primary components: a centralized, standalone manager (a Java program of about 1000 lines) that coordinates all test participants, and a small test wrapper around the standard Kudzu client that is run on all peers in the test network. Coordination of a replay test consists of several major stages, which we discuss here.

```
message BlockRequest {
  required sint64 fileChecksum = 1;
  required int64 offset = 2;
}

message BlockResponse {
  required bytes block = 1;
}

message ChunkSetRequest {
  required sint64 fileChecksum = 1;
}

message ChunkSetResponse {
  required bytes chunkSet = 1;
}

message FileStoreResponse {
  required string fileStore = 1;
}

message HostRequest {
  required int32 numHosts = 1;
}

message HostResponse {
  repeated bytes addresses = 1;
}

message ErrorResponse {
  required string errorMessage = 1;
}

message PeerExchangeRequest {
  required sint64 fileChecksum = 1;
  repeated bytes peerAddresses = 2;
}

message PeerExchangeResponse {
  repeated bytes peerAddresses = 1;
}

message QueryRequest {
  required string keywords = 1;
  required bytes requesterAddress = 2;
  required int32 ttl = 3;
  optional int32 id = 4;
}

message QueryResponse {
  required string keywords = 1;
  message FileStubMsg {
    required string name = 1;
    required int64 size = 2;
    required sint64 checksum = 3;
  }
  repeated FileStubMsg matches = 2;
}
```

Figure 4.4: Protocol buffer specification of all message payload types.

```
<USER>
    <PROPERTY>
        <USERID>436</USERID>
        <CONNECT_SPEED>Modem</CONNECT_SPEED>
        <CLIENT_SW>LIME</CLIENT_SW>
    </PROPERTY>
    <SHARED_FILE>
        <FILENAME>foo.mp3</FILENAME>
        <FILESIZE>4353681</FILESIZE>
    </SHARED_FILE>
    <QUERY>
        <KEYWORDS>bar</KEYWORDS>
        <TIMESTAMP>325360</TIMESTAMP>
    </QUERY>
    <QUERY>
        <KEYWORDS>quux baz</KEYWORDS>
        <TIMESTAMP>326988</TIMESTAMP>
    </QUERY>
</USER>
```

Figure 4.5: An example dataset user entry with 1 file and 2 queries.

4.3.1 Data Parsing and Cleaning

The dataset itself is simply a large (roughly 20 MB) XML file containing users and their files and queries. An example user entry is shown in Figure 4.5. The first step in conducting a test is parsing the XML file into virtual users. While the dataset is mostly free of errors, there are several files or queries with incorrect information, which we simply ignore during parsing. After parsing has completed, we shift all query timestamps so that the first query is sent at time 0 – this allows us to interpret the timestamps as simply delays from the start of the simulation. Timestamps are also scaled so that the simulation has the desired duration (roughly 2 minutes).

The last step in prepping the data is ordering the users. Since we have over 3000 users in the dataset but only about 1000 machines on PlanetLab (of which we can only harness about half at any given time), most virtual users will not actually participate in the test. Each test either generates and stores an ordering of the dataset user IDs or reuses an existing ordering. This gives us much greater confidence in running series of experiments, since most of the participating virtual users will be the same. We initially generated user orderings randomly, but found that sorting by file and query counts provided a better set of test users (discussed further in Chapter 5).

4.3.2 Virtual User Assignment

Once the manager has parsed the dataset and is ready to assign users according to the chosen ordering, it begins listening on the network for test peers to report in. Each test node is given the manager hostname as a command line argument so that it knows what address it should connect to for instructions. Once a test node connects to the manager, the manager takes the next available virtual user, serializes it, and sends it across the network. The connection is then terminated so as

to avoid requiring the manager to keep open connections to hundreds of other machines.

Once a test node has received its virtual user assignment, it populates itself with the files specified in the virtual user by writing a blank file of the correct size and filename. This is a convenience to ensure that no problems occur from not actually having the files in question. To keep file checksums from clashing, we write a small number of random bytes of data onto the front of each file. Since virtual users may have hundreds of even thousands of files, the process of populating the test node can take some time. Once the node is fully populated, it starts up a Kudzu node (but does *not* schedule its queries) and connects back to the manager, writing a single byte indicating that the node is ready to proceed. The node then waits for the manager to signal to start the simulation.

The manager, meanwhile, simply assigns users and waits for assigned users to report back for as long as is desired. Since some PlanetLab machines are on slow or otherwise inhospitable network connections, many machines will never actually report in to the manager, and some of those assigned may not signal that they are ready for a long time (for example, due to running out of disk space). The manager continues assigning and waiting for users until we tell it to proceed, at which point it stops accepting new connections and ignores any further peers that complete the file population.

4.3.3 Simulation

If not using a fixed organization policy, the manager can immediately tell every ready test peer to proceed with the simulation. Once a test node receives confirmation to proceed from the manager, it schedules every one of its queries at the specified timestamps. If the test is using a fixed organization policy, there is the added step of specifying which nodes should actually be used as neighbors. Unfortunately, this cannot be decided until the manager has a list of the peer that successfully reported in, since we don't want to assign unusable neighbors to peers. Thus, once the manager has finalized its list of 'live' machines, it creates a matching of them such that each peer has the required number of connections, then sends each of those assignments to the test machines. Once the test machines have confirmed their connection lists (as with the initial user assignment, some of the peers generally fail to confirm), the manager signals the start of the simulation as before.

4.3.4 Logging

Each test machine runs the simulation Kudzu node for the duration of the test data (that is, the maximum query delay in the dataset), plus an additional buffer time to account for the variation in the time at which nodes receive the simulation start order. During the simulation, nodes log every message type sent and received as well as message byte sizes and several other aggregate statistics, such as the number of query matches sent and received. At the conclusion of the simulation, nodes connect back to the manager a final time and send their logs over the network. Once a test node has sent its log, it clears the virtual user's files that were created for the test run and exits. The manager collects the logs, outputs them into a comma-separated value (CSV) file and calculates useful statistics such as query recall. Since the manager knows about every user that participated in the simulation (and by extension, every file and query on the network), it can determine the total number of query matches that could have occurred during the simulation.

4.3.5 Bootstrapping

The process of actually bootstrapping the test involves pushing out the Kudzu software to PlanetLab and then starting the software on as many of PlanetLab's 1000 machines as can be harnessed. We set a single machine up to use as a server for the Kudzu software (distinct from the machine being used as the manager to lessen the bandwidth load). Of course, in order for either the manager or any test nodes to download the software and begin execution, the command to start them must be sent from somewhere. The most straightforward way to issue commands to PlanetLab machines is through SSH connections. Unfortunately, given the number of machines we try to contact (roughly 1000), trying to open all SSH connections from one machine is a fairly fruitless endeavor due to the required upstream bandwidth of doing so. Thus, rather than using a single machine, we distributed the workload of sending the SSH commands to about 10 control machines. The master control program (a small Perl script) opens up SSH connections to each of the control machines and assigns each of them responsibility for an equal share of the PlanetLab machine list. Each control machine then opens SSH connections and issues both the software download and Kudzu start commands to every PlanetLab machine in their assigned portion, pausing briefly after each assignment to avoid overloading the software server from the hundreds of concurrent requests. Once each command is issued, the Kudzu test program begins executing and follows the procedure described above.

4.4 Summary

The process of implementing Kudzu was primarily an exercise in designing a robust and efficient system of communication between peers in the network. The final solution of using protocol buffers provided a useful amount of abstraction while remaining low-level enough to keep communication overhead to a minimum. Some messages (for example, a ping message) are transmitted across the network using as little as 5 bytes of data – a one byte header, a two byte type field, and a two byte message id.

Implementing the test harness was mostly an issue of accounting for both the large size and general unreliability of PlanetLab. Since PlanetLab contains many unreliable machines, our simulations had to try to keep tests as consistent as possible in an environment in which machines are constantly acting unexpectedly or not responding at all. Furthermore, we needed to deal with the challenges of coordinating hundred of machines from a single manager.

Chapter 5

Evaluation

In order to evaluate the effectiveness of our design and implementation choices, we conducted extensive tests of a Kudzu network using our client by running our test framework on PlanetLab. Of PlanetLab's roughly 1000 machines, we were able to harness roughly half in our tests, which we found to be sufficiently large to give useful results. We present the results of our experiments here.

5.1 Evaluation Metrics

The first consideration in designing our experiments was deciding what we wanted to evaluate during our tests. We decided on three primary aspects of the network that we were interested in measuring: bandwidth utilization, query recall, and download speeds.

5.1.1 Bandwidth Utilization

Since the primary scalability bottleneck in a fully decentralized network like Kudzu is bandwidth, we wanted to gather realistic data on the amount of network traffic actually used by Kudzu. Since we had an actual implementation of Kudzu to experiment with, measuring bandwidth usage was a straightforward matter of totaling all incoming or outgoing messages at each node and then aggregating this information at the completion of a test. For each message sent or received across the network, we recorded both the message type and the message byte count. At the end of the simulation, for each message type $t \in \{ping, backconnect, query_request, query_response, \ldots\}$ and direction $d \in \{received, sent\}$, each node returned to the simulation manager the total number of t messages in direction d and the average byte size of the message group. We were then free to gather any statistics we wished from this data corpus.

With the notable exception of block transfer messages, most messages in a Kudzu network can be expected to be fairly small (under 100 bytes) and able to fit inside a single TCP packet. Thus, to calculate that actual number of bytes transmitted across the wire, we add a standard 20 byte TCP header and a 20 byte IP header onto the size of each Kudzu message. Due to the small size of most Kudzu messages, these headers have a significant impact on the network's total bandwidth usage.

5.1.2 Query Recall

Since one of the primary goals of Kudzu was an effective method of autonomously organizing the network, we needed a metric corresponding to how effective a given organization policy actually was. Let $N = \{n_1, n_2, \ldots, n_k\}$ be a network of k nodes where a node $n_i = \{F_i, Q_i\}$ is comprised of a set of files and queries. We define **query recall** or simply **recall** as the number of possible query matches observed during a test over the total number of possible matches in N. More formally, for a network of predefined users N, starting network configuration C (including the maximum TTL setting), and organization policy P, we have the following query recall R:

$$R = \frac{matchesObserved(N, C, P)}{matchesPossible(N)}$$

We assume that N in configuration C is connected; in practical terms, this means that every query in $Q = \bigcup_i Q_i$ has the potential to be matched against every file in $F = \bigcup_i F_i$. Since we know the entire network N before we begin the simulation, we can calculate $matchesPossible(N)$ offline by simply checking every node's query against every other node's fileset and tallying the number of matches. In theory, we could write a network simulator to calculate $matchesObserved(N, C, P)$ offline as well by simulating network activity according to N, C, and P and measure the results. However, due to the variety of timing issues and unexpected network events that may be encountered during an actual test, we opted to calculate this value from live experiments.

We define a single match as a matching of a single file to a single query, *not* a single node's matching of a single query. This means that if a node n has 20 files that match a given query q, n will report 20 matches to q, even though those 20 results are returned in a single message. The rationale for this is that query matches that return many results are more important than those that return few results; a peer's query that returns 50 results from a single node is probably more useful than one that returns only 10 results from two nodes. Another issue we needed to consider was query duplicates, due to the fact that in the dataset, nodes often issued the same query several times in rapid succession. We opted to allow the duplicate queries (and thus allow their results to be tallied multiple times) on the basis of two notions: one, that queries that are issued many times are probably more important to the user than those that are not, and two, that organization policies acting on query matches returned or queries issued may cause the network topology to change between duplicate queries, causing different results to be returned from new queries than from earlier identical queries.

During the actual simulation, each node tallied the matches it received from other nodes and the matches it sent to other nodes. At the conclusion of the simulation, these two sets were returned to the simulation manager. Under perfect conditions, the sum total of all matches sent and received across the entire network should logically be equivalent, since every match registered as sent will subsequently be registered as received. In practice, of course, this is rarely the case – some PlanetLab nodes cannot reach others, and timing issues often mean that some nodes complete the simulation and exit before matches found by lagging nodes can be received. To calculate the value of $matchesObserved(N, C, P)$, we first gather all sets of received and sent matches and add one match for every matching pair of a received and sent match. Once this is done, we are left with a set of

unpaired sent matches and a set of unpaired received matches (the latter may occur when nodes fail to report back to the manager at all after the simulation). We opted to add the number of unpaired sent matches to the total number of matches, though this generally adds less than 5% to the total.

5.1.3 Download Speeds

Finally, we wished to measure file download speeds, primarily to serve as a comparison to speeds observed in the same transfers when conducted through BitTorrent. This is simply an issue of setting up a download swarm and timing the nodes downloading the file (along with the average transfer speeds). A cumulative distribution function of download completion times serves well as a comparison of the same peers downloading the same file on either a Kudzu or a BitTorrent network. To ensure that the two networks were on a relatively even playing field, we set Kudzu's chunk size and block size to BitTorrent's defaults of 512 KB and 16 KB, respectively. Though we were unable to determine BitTorrent's default request pipelining policy, we ran Kudzu with a fixed pipelining setting of 10.

5.2 Dataset Peer Selection

An important consideration for the bandwidth and query recall tests was how to select the roughly 500 peers to simulate from the 3500 users in our dataset [12]. One problem we quickly encountered was much of the dataset was quite sparse; many users were sharing few files and most issued a very few number of queries. We suspect that this is due in large part not the users themselves but to the method in which the dataset was gathered; though the dataset spanned a period of 3 months, most of the users captured during that time were probably only active during a tiny fraction of that period. We initially selected our simulation peers randomly, but this resulted in an average of only 3 to 4 queries per simulated node; though our simulations took far less than 3 months to run, this still proved to be insufficient data. To compensate, for each peer p with f files and q queries, we assigned p a score of $(f + 100 * q)$ and then ranked all peers by their scores (we decided on a query factor of 100 since our original average number of files was roughly 100 times the average number of queries). We then imposed a minimum of 50 files and 5 queries per node and selected our simulation peers from the resulting ordering. Approximately 400 of the 3500 peers fit these criteria; once these 400 were assigned, we removed the minimum file and query cutoffs and selected the final (roughly 100) peers according only to the ranking score.

One reasonable concern is that the handpicking of our users from the dataset skewed our results unrealistically – after all, all of the users in the dataset were equally 'real', so only using a particular subset seems potentially harmful. Our rationale for this stems from the fact that in all likelihood, most of the users in the dataset only resided on the network briefly before leaving. The prevalence of short-lived, fairly inactive users is harmful to our experiments because our dataset does not have peer lifetime information. As a result, each user we select for the simulation resides on the network for the entire duration of the simulation, even though the actual user was probably observed on the network only briefly. This has the effect of greatly thinning out the amount of traffic in a given

simulation run relative to the amount of traffic actually observed when the dataset was captured. Selecting the most active nodes is an imperfect solution to this problem but serves to compensate for this undesirable thinning effect. Furthermore, while the users we select is skewed towards those most active, since we do not modify any data on a per-user basis, each user itself remains a source of fully realistic data.

Our timed tests were performed with a time factor of 100,000, resulting in a raw simulation time of roughly 2 minutes – this simulation time was actually substantially smaller than the time required to contact all peers, assign files and queries, and perform other simulation setup needed. Each test run took roughly 20 minutes from the time the central manager was started to the time the results were received.

5.3 Bandwidth Motivation

Recall from our discussion of TTL in Chapter 3 that we assumed the network is organized like a tree in which every hop would reach $(c - 1)$ new peers, where c is the number of connections maintained per peer. Of course, a real network does not form as a tree, but will instead contain many cycles. Cycles are beneficial in that they provide important redundancy in the network; this keeps the network connected even as peers come and ensure that queries are not dependent on single peers to reach other parts of the network. However, they can also result in wasted bandwidth by allowing a single query to be routed to a single recipient node multiple times. While filtering out these duplicate queries is straightforward at the destination node, every duplicate query received is a waste of bandwidth.

We initially explored what would happen if we did not restrict TTL at all but instead allowed queries to propagate throughout the entire network. By our earlier analysis, we already expect uncapping TTL to be highly bandwidth intensive and probably unscalable, as each new query on the network results in an increase in the amount of traffic every node has to handle. However, we did not take into account the wasted bandwidth derived from duplicate queries. In an early experiment, we decided to measure the impact of these duplicates. We started a network of several hundred nodes and had each node send randomly generated queries at frequent intervals, then periodically sampled the number of times a single query was received by a particular node. This gives us a ratio of the number of new, unique queries received to the number of duplicate queries received. The results of this test are shown in Figure 5.1.

We see that while the unique query rate was fairly volatile, during most of the test it stayed around 25%. This means that on average, each query was received 4 times by each node, resulting in 3 wasted query messages for each useful message. Furthermore, note that this 75% inefficiency accumulates on top of the already exponential network bandwidth incurred by having every node see every query. Soon after observing these results, we ceased working with complete query propagation and instead turned to studying the impact of particular TTL settings on a network.

Our initial bandwidth test aimed to verify that the total bandwidth used by the network would be exponential in the max TTL setting. We ran the dataset simulation on PlanetLab once for each max TTL value through 10 and calculated the aggregate bandwidth used by the network. This

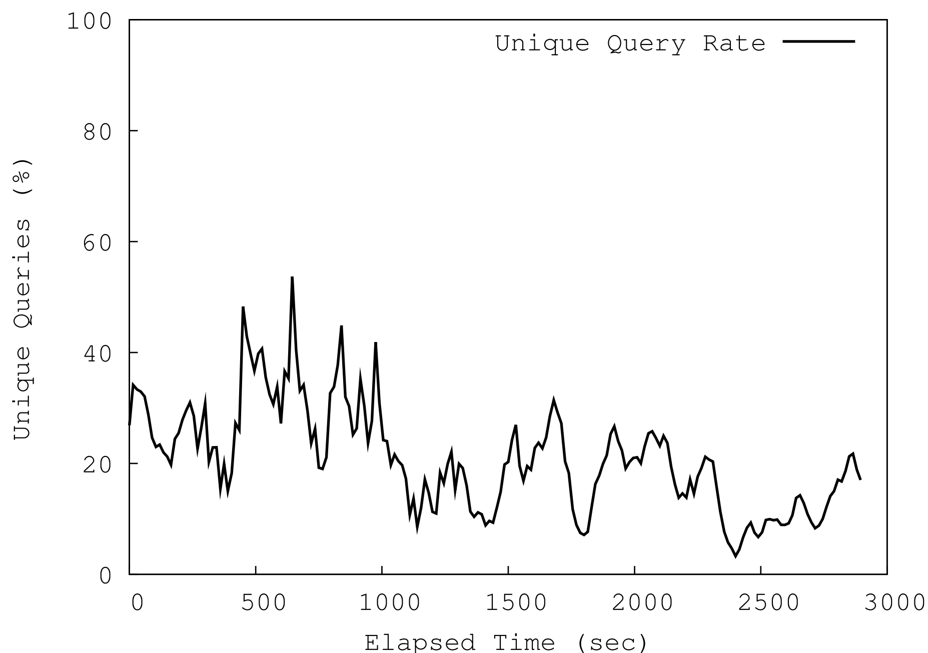

Figure 5.1: Unique query ratios in a network with uncapped TTL.

includes query requests, query responses, and any other messages exchanged on the network. Note that it does *not* include any downloads, since for these tests we did not actually initiate any file transfers when matches were received. Our results are shown in Figure 5.2. A random network organization was used with a minimum connection setting of 3 and a maximum connection setting of 4. These values were chosen to provide a full range of minimal network coverage to near-complete network coverage as the max TTL increased to 10. Furthermore, these values are typical real-world settings – the original Gnutella employed 4 connections per peer.

We see from the curve that bandwidth usage increases significantly more than linearly in the TTL; its exponential tendency is particularly pronounced up to TTL 6. More variation is present at higher TTL values, though this likely has to do with the size of the network – with 3 to 4 connections per node, some queries start reaching most of the nodes in the network around TTL 7 and may stop propagating in less than the maximum number of hops. However, the aggregate bandwidth continues to increase steadily along with TTL. This confirms our hypotheses about the role of TTL in network scalability; that is, increasing the TTL enough for queries to cover an entire network is an unscalable proposition as we allow the network to grow. Though the absolute bandwidth usage in our test is fairly modest, this is subject both to the fact that our network is only modestly sized by current standards and the fact that each of our simulation nodes averaged only 10 to 15 queries over the entire simulation. In a real network in which users are constantly joining to issue queries, higher query rates are quite likely.

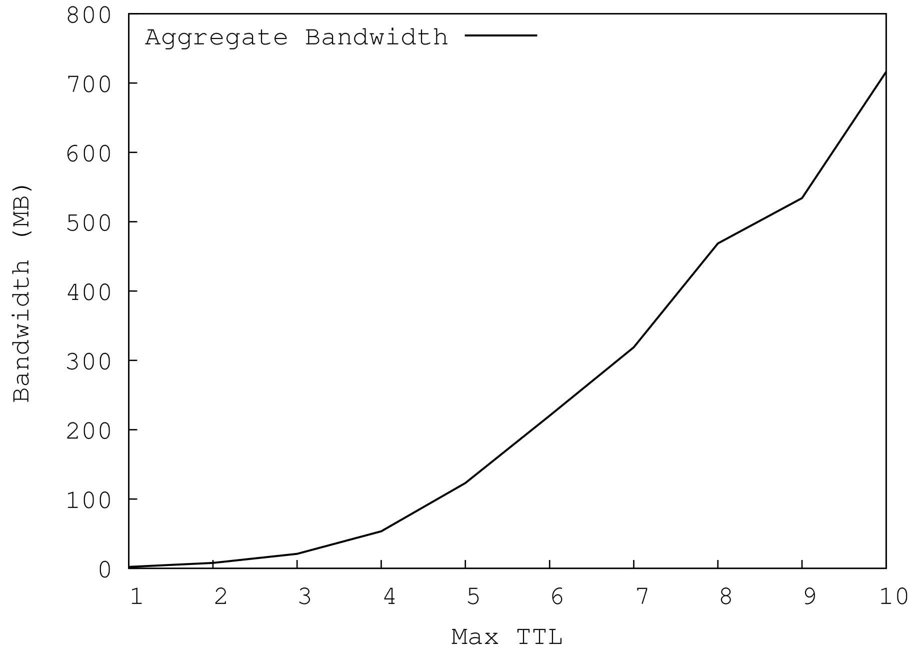

Figure 5.2: Aggregate bandwidth usage across a range of max TTL values.

5.4 Organization Strategies

Given the link between TTL and bandwidth usage, the goal is to maximize query recall while minimizing the TTL (and thus bandwidth usage as well). We investigated the effectiveness of four different organization policies, which we detail here (see Section 3.3.1 for a description of the general policy types). Recall that we refer to the minimum number of peer connections as MIN and the maximum number as MAX. For all of our tests, we set MIN to 3 and MAX to 4.

- A fixed policy with random organization. For this organization, the manager assigned each peer in the simulation at least MIN and no more than MAX other peers to connect to. The selection process consisted of randomly picking two peers from the pool of peers with less than MAX assigned connections and pairing them, then repeating until all peers had at least MIN connections or no further pairings were possible. This process was entirely executed on the central manager, which simply informed the simulation nodes of their connections once all pairings were complete. Note that in a real network, peers usually join through a small set of public nodes, resulting in a non-random network. Thus, this policy is unrealistic in practice.

- A naive organization policy with a single entry node. This is effectively the simplest possible realistic network, as everyone joins the network through a single publicized entry node and then finds other peers through that peer without any particular selection criteria. Connections are chosen to be established or disconnected randomly so as to maintain between MIN and MAX connections per node.

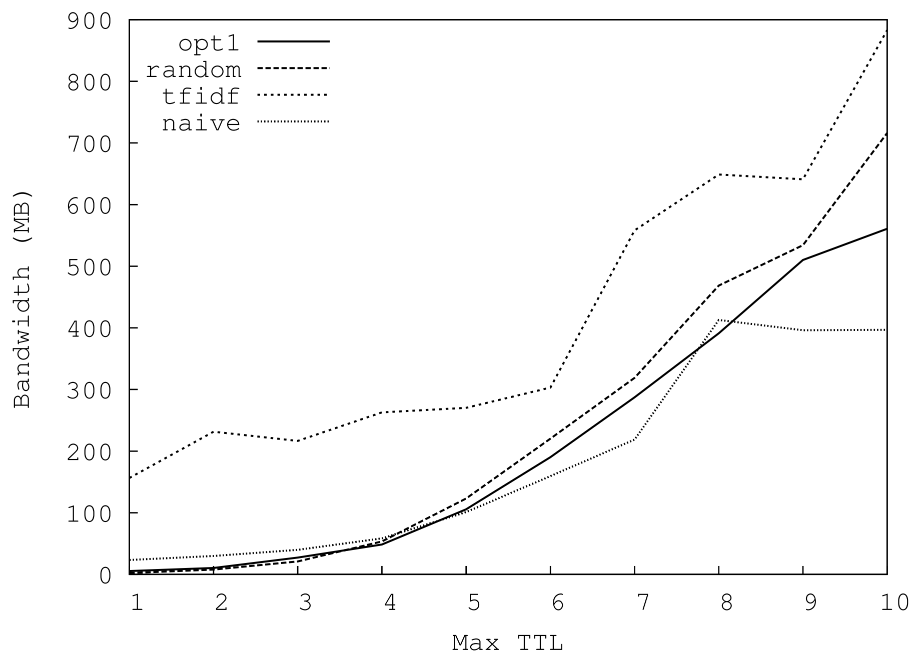

Figure 5.3: Aggregate bandwidth usage versus max TTL for each of the four organization strategies.

- An optimal policy within a single hop (we refer to this policy as OPT1). This is a fixed policy in which peer connections are chosen to maximize the number of matches that will be received within a single hop. Before the test begins, the manager calculates all matches between node pairs, then ranks the pairs according to the number of matches. Connections to assign are then chosen in order (within the constraints of MIN and MAX) in the same manner as the first strategy. This is clearly an impossible policy to implement in a live deployment, since it requires knowledge of queries not yet issued – however, it shows the gains of organizing optimally among a node's direct neighbors.

- The TFIDF policy described in Chapter 3. This is our second realistic policy – nodes enter through a publicized entry node and then begin exploring the network, choosing connections so as to maximize the sum TFIDF score of the node's peers. No manager or central intervention is required to implement this policy.

5.4.1 Policy Bandwidth Use

We first calculated the amount of bandwidth used by each of the four strategies to determine the baseline relationship of TTL and bandwidth use. The aggregate bandwidth used in each of the policies over 10 runs of increasing TTL values is given in Figure 5.3.

Although the general exponential trend is evident across all four of the organization schemes, the most striking difference is the extra bandwidth used by TFIDF organization. This is a result of the

bandwidth required in transferring the file stores required to calculate TFIDF values. Recall that a node's file store is the set of all of the words in its filenames and their associated frequencies. Given that we sorted virtual users partially by the number of files they contained, it is unsurprising that many of their files stores were large. Consequently, constantly transferring large filestores across the network during exploration expended a considerable amount of bandwidth.

However, the bandwidth used by this exploration does not disqualify TFIDF outright. Exploration occurs at a fixed rate per node and has no impact on the exploration conducted by other nodes; in other words, exploration adds only a constant amount of bandwidth per node, or an aggregate linear increase in the size of the network. Our data corroborates this; the difference in aggregate bandwidth between TFIDF and the others at TTL 1 is roughly the same as it is at TTL 10 (and those in between). There is a fair amount of variability owing to the randomized nature of network exploration, but we would expect the percentage of the network's overhead consumed by exploration to decrease as the network is scaled up in size, owing to the increasing dominance of query traffic.

Naive organization consumes a small but noticeable amount of bandwidth more than random or OPT1 at lower TTLs. Similar to TFIDF, this is due to the node exploration that the naive policy requires in order to find new connections. However, once peers are located, file stores do not need to be exchanged, explaining why the overhead incurred by naive is much less than that incurred by TFIDF. At higher TTLs, however, naive organization begins to see *less* bandwidth usage; this is explained by network fragmentation, which we address more fully in the following section.

5.5 Query Recall Tests

For the same set of tests, we calculated the query recall for each of the four organization strategies across the range of TTL values. The results are shown in Figure 5.4. For OPT1, random, and TFIDF, we see the expected trend towards 100% recall as TTL increases. We found that the average number of connections per node in each of the tests tended strongly towards MIN rather than MAX. Thus, assuming that we have an average of 3 connections per node, with a TTL of 9 we know that queries can reach up to $3 * 2^8 = 768$ nodes; since we have roughly 500 nodes, we expect that this should be sufficient to reach most of the network (taking into account cycles, PlanetLab, and so forth). The unreliability of PlanetLab means that we cannot expect to actually obtain 100% recall, but the fact that recall passes 90% at TTL 10 indicates the trend towards perfect recall.

OPT1, as we expect, does well at TTL 1 (and continues doing well up until TTL 4) relative to the other strategies. Given that it is premised entirely on 1-hop locality, it is understandable that its advantage disappears at higher TTLs. Even at low TTLs, however, It does not do as well as one might expect. Since it is optimal within 1 hop, we can conclude that query responses were generally spread across many nodes rather than localized to only a few. This is because with approximately MIN connections, only about 10% of potential query responders could be placed within a single hop. This result suggests that a reasonable TTL is always necessary to avoid crippling query recall even with an arbitrarily powerful organization scheme. However, this result is also due to the fact that we picked the virtual users with the heaviest load – in earlier tests of randomly selected virtual users, we observed recall as high as 20% for OPT1. It is not surprising that a sparser network allows

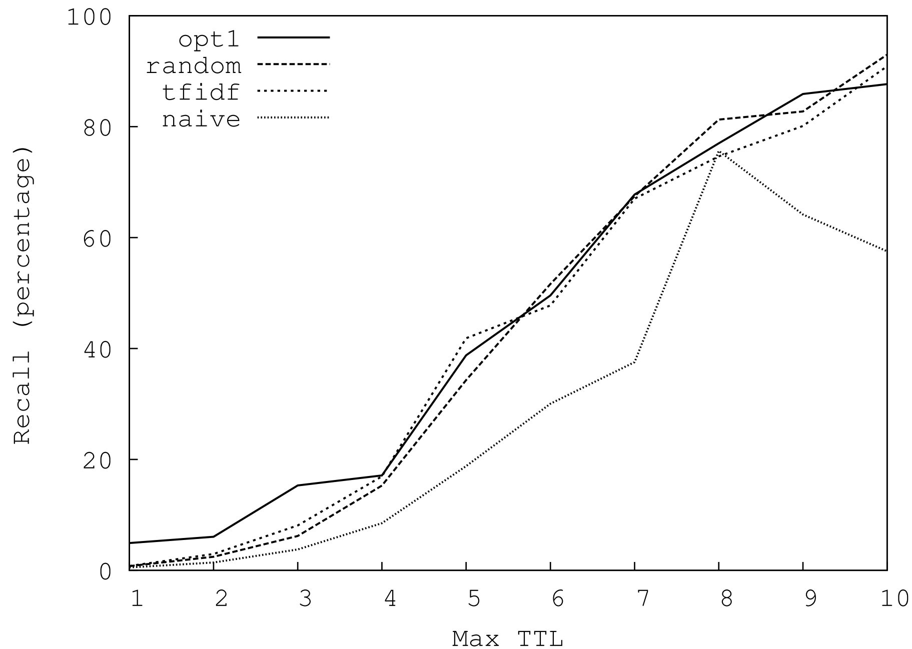

Figure 5.4: Query recall versus max TTL for each of the four organization strategies.

a greater percentage of the total queries to be found with the same number of connections.

TFIDF and random organization perform comparably throughout. While it is unfortunate that TFIDF does not appear to lend a significant benefit over random organization, it is important to note that the random organization scheme cannot easily be implemented in a live network. More specifically, since the users in a real network are not known before they join, ensuring a random distribution of connections is difficult. Our TFIDF scheme, however, does not assume any unrealistic capabilities and could be applied to any live Kudzu network.

The most interesting result of this graph, however, is the extremely poor overall performance of naive organization combined with the spike at TTL 8. At first glance, it is surprising that naive performs worse than random, much less significantly worse. Explaining this discrepancy was the next part of our investigation, which we discuss below.

5.5.1 Network Organization

Since our strategies were designed to organize the nodes in the network so as to maximize recall, it made sense to examine the actual structure that the networks took using the various organization schemes. We were doubly interested in examining this structure after seeing the performance discrepancy between naive organization and random organization, from which we initially expected similar performance.

To determine the structure of the network at a given point in time, we issue a special traversal message from a start node to all its neighbors, which each then forward the message along like a

usual query. However, unlike a query, we impose no TTL on the number of hops and send responses from every recipient node containing a list of the node's current connections. Once all responses have arrived at the initial node, it has enough information to reconstruct the entire network. Note that in the case of the dynamic organization schemes (naive and TFIDF), this will not be an exact snapshot, since links may change partway into the traversal, but the traversal time is short enough to give a good approximation.

What we found after investigating the behavior of a naively organized network was the presence of two factors contributing to poor recall:

1. **Fragmentation.** One issue was that pieces of the network would sometimes break off from the main network and form a smaller separate network. This is obviously highly detrimental to query recall, as queries from each of the subnetworks cannot reach each other, resulting in many potential matches that can never be fulfilled. Fragmentation is likely in the naive scheme due to the fact that the entry node does not maintain any more connections than each of the other nodes. A series of new nodes arriving at the same time will request new peers from the entry node, and these new peers will be selected from the same small set of peers that are still connected to the entry node. Later-arriving peers will then arrive at the entry node, causing its existing connections to be reshuffled. This reshuffling may cause a small group of highly interconnected nodes to break off if their only connections to the primary network are through the entry node (whose connections are highly volatile due to new arrivals).

2. **High network diameter.** The fragmentation that we observed was generally fairly limited. The larger problem we found was that the diameter of the network (that is, the maximum number of hops from any node to any other node) was quite high in the naive strategy. With a high network diameter, the TTL required to reach most of the other nodes in the network will increase, resulting in poor recall. The cause of a high network diameter may be understood as a less severe version of fragmentation: highly clustered sets of nodes effectively waste many of their connections making cycles to other nodes in the same cluster rather than connecting to other parts of the network. Even if the cluster remains weakly connected to the rest of the network, queries entering the cluster will spend several hops without encountering many new nodes.

One of the networks measured during a naive test run is shown in Figure 5.5. Though this particular snapshot did not reveal any disconnected components, the network is clearly unbalanced and contains several tightly clustered pieces with few connections to other parts of the network; one such cluster is indicated. Note that for the five primary nodes in the cluster, queries with a max TTL of 3 will reach only 9 other nodes. Assuming that nodes are evenly distributed between having 3 and 4 connections, we'd expect queries in the general case to reach $3.5 * 2.5^2 = 21.9$ or roughly 20 other nodes; thus, this isolated cluster is clearly functioning at a deficit relative to the TTL.

The conclusion we draw from our recall results is that the naive organization strategy results in a poorly distributed, unbalanced network. This means that at the same TTL as a different scheme, queries reach fewer nodes in the network. The spike observed at TTL 8 is presumably a fluke; the

Figure 5.5: Network topology resulting from naive organization. Note the weakly connected cluster in the upper right.

most likely cause is that some small set of auspicious connections formed bridges to otherwise remote portions of the network, alleviating the high network diameter observed in the other nine runs and resulting in comparable performance to the other organization strategies. Given that this behavior only occurred once out of ten runs, however, it seems that a severely unbalanced network is the most likely outcome. The reduced bandwidth observed in the naive strategy is also explained by an unbalanced network. Since queries encounter fewer unique nodes, more cycles occur, thus resulting in discarded rather than forwarded queries as nodes are visited multiple times.

Exploration Strategies

Given that the overall cause of poor naive recall was an unbalanced network, we investigated how a simple organization scheme could be made more balanced. When a node needs new connections, it requests new peers from one of its neighbors. Optimally, the new peers that are returned are selected randomly and uniformly from the set of all possible peers (as is the case in the random organization strategy). Given that nodes have incomplete information in a true network, this optimal behavior is not possible. However, a node can approximate optimal behavior fairly well if it modifies the way in which it returns new peers. Under the original scheme, peers are returned from the set of currently connected peers. If a peer conducts its own exploration and returns peers from a list much larger than just its present connections, the overall peer connection distribution will be much more evenly distributed. We summarize the two exploration strategies as follows:

- **Passive exploration**. When a node n requests new peers from another node m, return randomly selected peers from the set of peers currently connected to m.
- **Active exploration**. Perform periodic exploration of the network by contacting a known peer and adding all of its connections to a growing list of known peers. When a node n requests new peers from another node m, return randomly selected peers from the peer list compiled by m.

We ran tests of naive organization with both passive and active exploration and took snapshots of the resulting network topologies. Snapshots for passive and active exploration are given in Figure 5.6 and Figure 5.7, respectively, with all nodes in the network arranged in a ring.

Passive exploration clearly results in a significantly more unbalanced network than active exploration; Figure 5.6 exhibits both significantly larger coverage gaps in the ring topology and groups of nodes with high degrees of interconnectedness. Some of these features are noted in Figure 5.8.

Given that attempting to randomly balance the network has such a striking effect on naive organization, a natural question to ask is whether it would have a similar effect on other organization strategies. Note that the TFIDF strategy already performs what is effectively active exploration to build a peer ranking, with the added aspect of file store requests to peers added to the ranking. However, our TFIDF strategy still returns new peers to requesting nodes using the passive method; the full ranking list is used only by the compiling node itself to decide on its connections. Thus, the distinction between passive and active exploration can be applied to TFIDF organization as well as to the naive organization. To see what effect this would have on TFIDF organization, we reran the

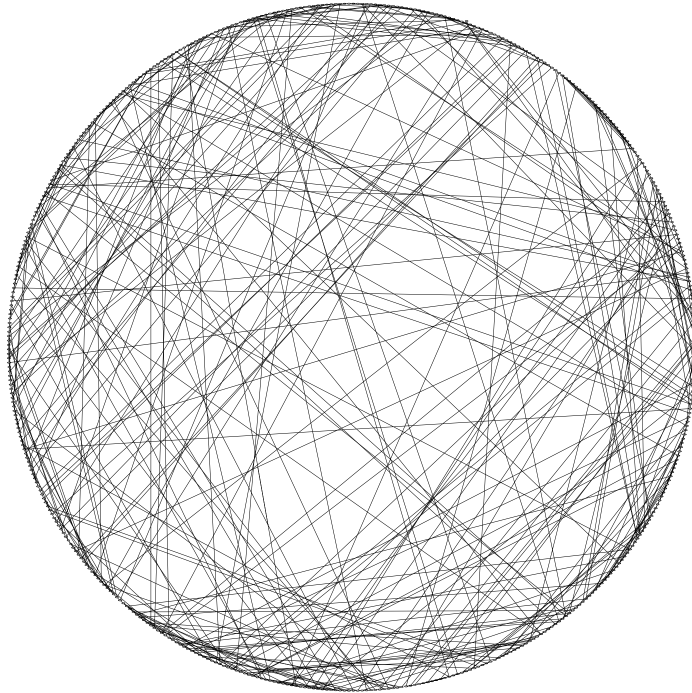

Figure 5.6: Circular network topology resulting from naive organization with passive exploration.

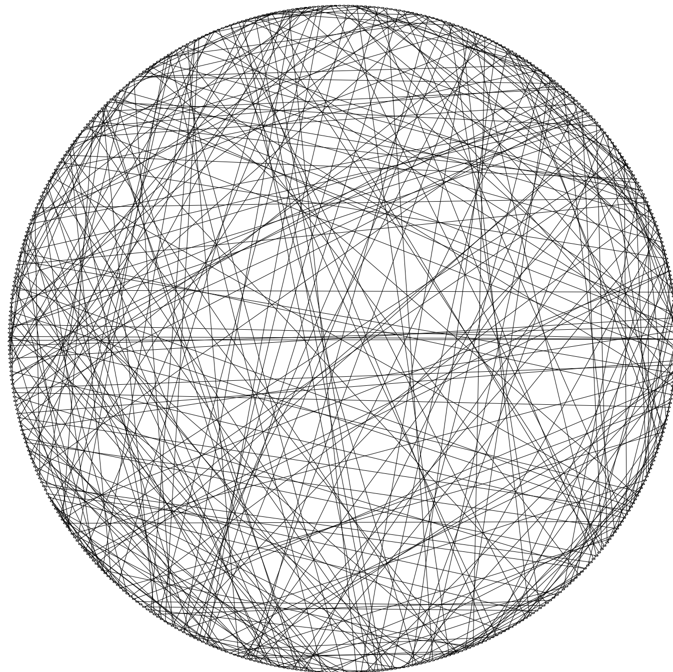

Figure 5.7: Circular network topology resulting from naive organization with active exploration.

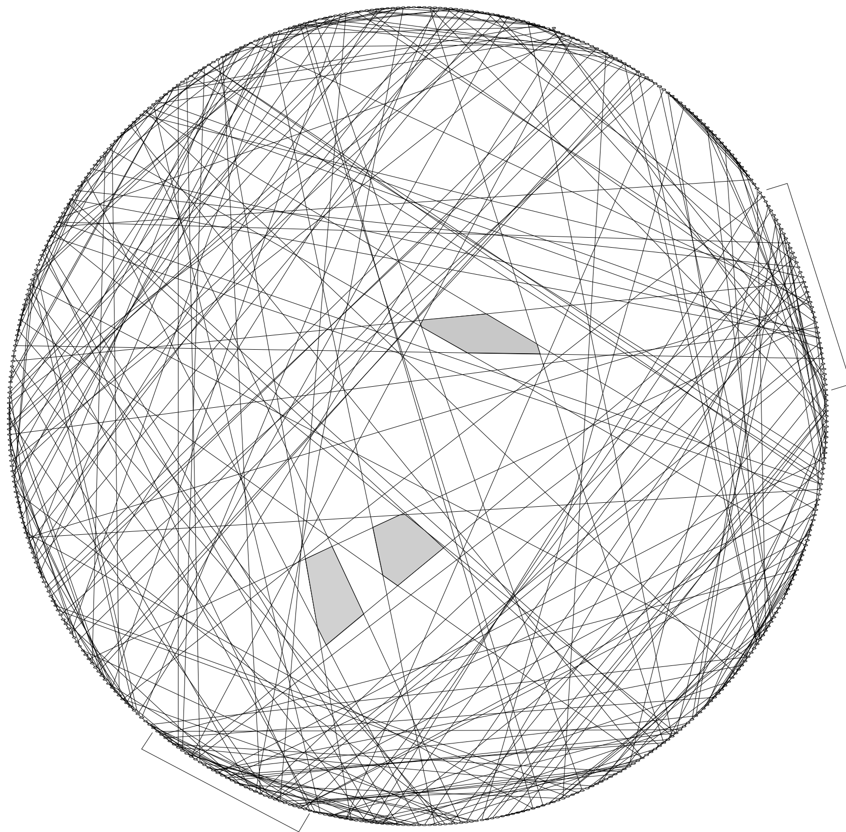

Figure 5.8: Naive organization with passive exploration and noted coverage gaps (shaded regions) and highly interconnected node groups (demarcated by lines).

same series of topology tests as before with TFIDF rather than naive organization. Snapshots are shown for passive and active exploration in Figure 5.9 and Figure 5.10, respectively.

Both resulting topologies are somewhat unbalanced, especially when compared with naive organization with active exploration. In this case, however, an unbalanced network indicates not that the organization is ineffectual but that TFIDF is accomplishing its goal; namely, unbalancing the network in such a way that recall is improved (or, at least, left unharmed) but forming clusters of nodes with high TFIDF scores to each other. Although the recall results from TFIDF were not markedly higher than random, these results suggest that TFIDF is, in fact, accomplishing its intended goal to some degree.

To empirically verify these conclusions, we reran the full set of bandwidth and recall tests on naive organization with active exploration. This fifth line is plotted alongside the existing four for both aggregate bandwidth (Figure 5.11) and query recall (Figure 5.12). We see that aggregate bandwidth falls in line with random and OPT1 organization and does not exhibit the flatline behavior at high TTLs present in passive naive organization. Active exploration does expend a small amount of additional bandwidth over passive even at low TTLs, however; this is understandable, given that active has to perform a constant amount of exploration per node. Since this exploration does not need to transfer file stores, however, the expenditure is much less than in TFIDF organization.

Recall exhibits similar trends. The deficiencies in passive naive almost entirely disappear and the resulting recall performance is on par with the three non-naive organization strategies. While still falling slightly below TFIDF at low TTLs, performing active exploration appears to make naive exploration as viable as TFIDF exploration.

Performing active exploration versus passive exploration in TFIDF appeared to have little effect; though we do not plot a sixth line here, there was minimal change between our original TFIDF results and those with active exploration. At first glance, these results may seem to mark naive organization with active exploration as the organization scheme of choice, given its similar performance to TFIDF without the bandwidth overhead of transferring file stores. However, this ignores the tradeoffs of performing passive vs. active exploration besides the small bandwidth overhead of active exploration. In particular, if a peer p has a peer p_2 in its list of known peers but is not actually connected to p_2, then p has no guarantee that p_2 is still online. For a peer p_3 requesting new peers from p, either p may return stale information to p_3 or p will have to manually check that p_2 is online by establishing a new connection and exchanging a message (introducing extra latency and bandwidth into the original peer request). If passive exploration is used, however, all returned peers are guaranteed to be valid. TFIDF organization may use passive exploration without harming recall; naive organization, on the other hand, is effectively forced to use active exploration.

Another significant benefit of TFIDF (or, for that matter, any adaptive organization scheme) is its implicit incentive model that benefits peers who remain online even when not exchanging queries by finding more useful connections through continuous exploration and TFIDF ranking. As we tune TFIDF or explore other adaptive organization schemes that are more effective, the incentive to users to remain online only increases. Thus, we conclude that naive (with active exploration) and TFIDF organization both have tradeoffs and neither is a clear winner over the other. A brief summary of the benefits and limitations of the organization strategies we evaluated is given in Table 5.1.

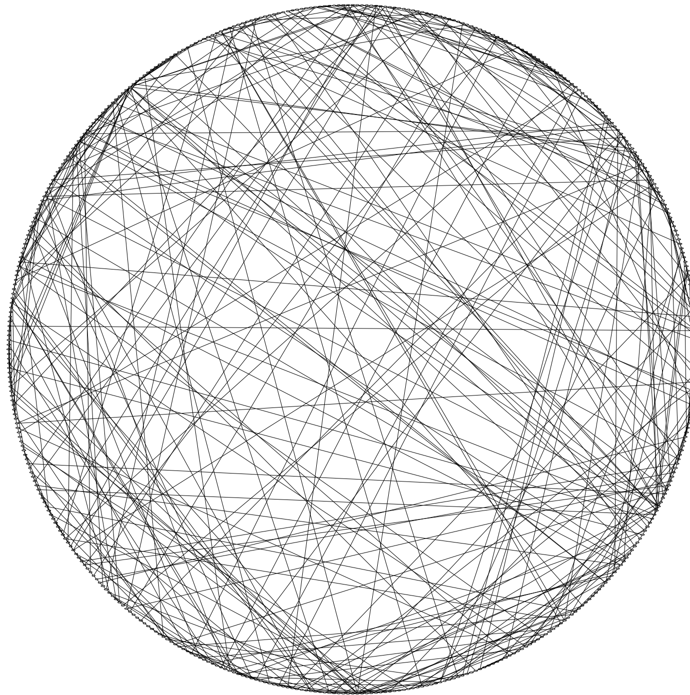

Figure 5.9: Circular network topology resulting from TFIDF organization with passive exploration.

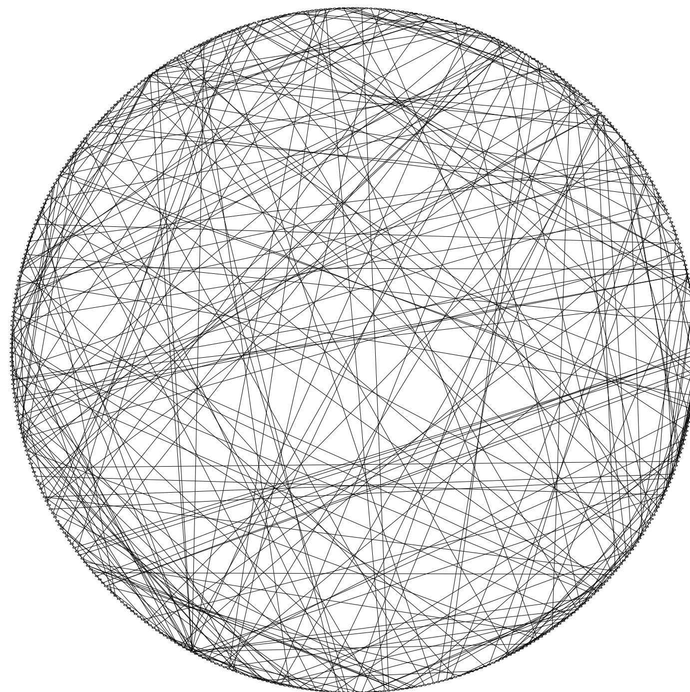

Figure 5.10: Circular network topology resulting from TFIDF organization with active exploration.

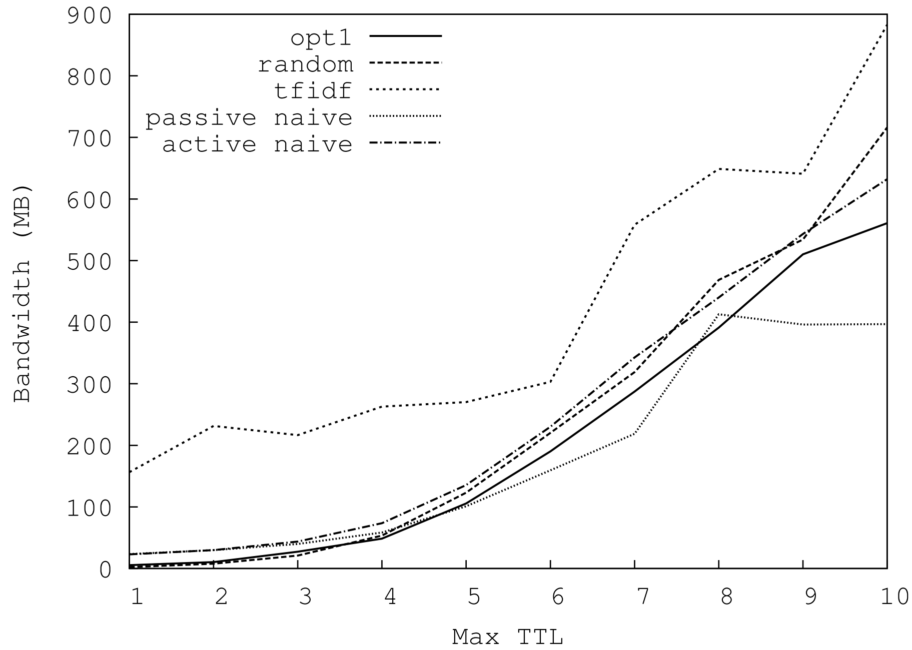

Figure 5.11: Aggregate bandwidth usage versus max TTL including naive with active exploration.

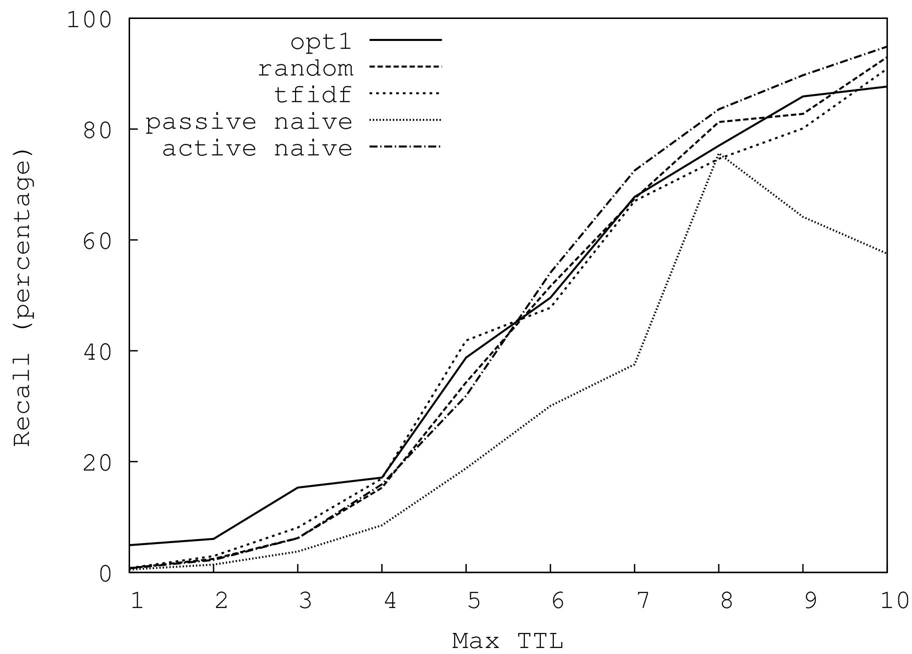

Figure 5.12: Query recall versus max TTL including naive with active exploration.

	Benefits	Limitations
OPT1	Good at low TTLs	Not realistic (requires oracle)
Random	Perfectly balanced	Not realistic (requires oracle)
Naive	Low bandwidth	Must perform active exploration
TFIDF	Passive exploration, implicit incentive model	Bandwidth cost of file stores

Table 5.1: Overview of benefits and limitations of our four organization strategies.

5.6 Download Tests

Finally, we turn briefly to download tests using our client. Our primary objective in measuring download speeds was to demonstrate that the tracker used by a BitTorrent network is not necessary to facilitate high speed downloads involving more than 2 participants (that is, a swarm versus a one-to-one download).

To test download speeds, we selected three initial seeds and placed the same 75 megabyte file on each of them. We then selected roughly 25 other machines scattered across PlanetLab to participate in the file download. We ran the file download on the same set of machines for both Kudzu and BitTorrent. For Kudzu, the three initial seeds started a Kudzu network and the 25 downloaders then connected to the network, issued a query matching the test file, and began downloading as soon as the first response was received. For BitTorrent, we set up a tracker file using a high-capacity public tracker hosted by The Pirate Bay and had the three seeds first connect through the torrent file, followed by the 25 downloaders. We used the mainline Linux BitTorrent client, version 4.4.0.

In both cases, the file was divided into 512 KB chunks, each of which was further divided into 16 KB pieces. We show the cumulative distribution function of the download completion across the entire swarm in Figure 5.13.

A similar trend is evident for both networks; one of the downloaders finishes very quickly, followed by a gap of several minutes after which most of the downloaders complete within a short span. The three seeders were located at the same site as one of the downloaders; this explains why one of the downloaders completes much more rapidly than the others in both networks.

The bulk of the downloaders complete roughly 25% faster under BitTorrent than under Kudzu. However, this is easily attributable to both parameter tuning and the chunk selection algorithm. In our Kudzu client, we simply set parameters (message timeouts, chunk update intervals, and the like) to reasonable values and did not experiment, since absolute performance was not a primary concern once we were within a reasonable distance of BitTorrent. More importantly, however, is BitTorrent's mechanism for selecting chunks to download. BitTorrent peers download random chunks for the first few rounds, then switch to favoring chunks downloaded by the fewest number of peers in the swarm. This has the effect of disseminating poorly replicated chunks first, thereby allowing more peers to upload. At present, Kudzu simply chooses randomly for the duration of the download. We observed that Kudzu was hampered by this during the download – chunks fully uploaded by the

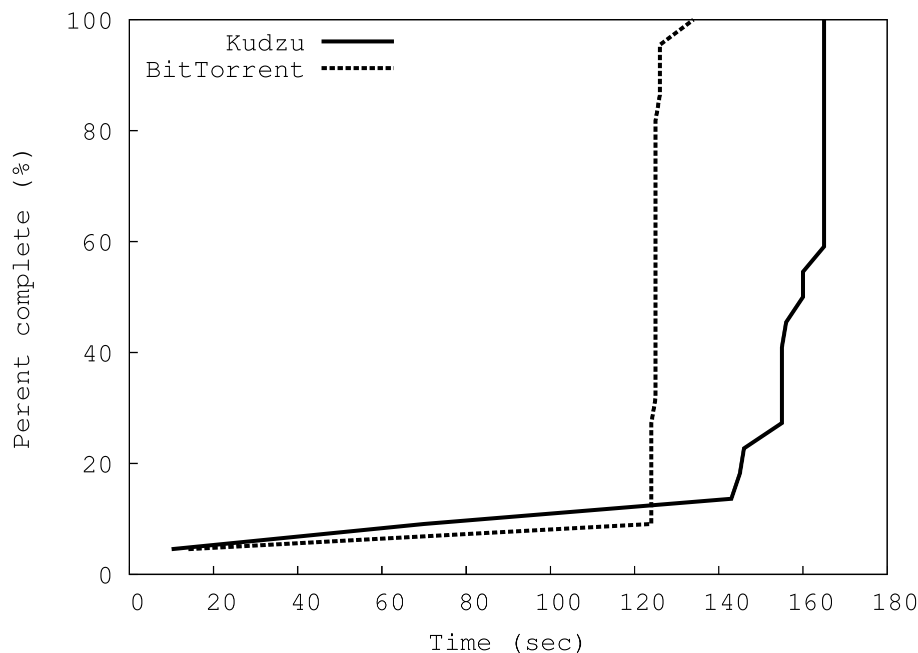

Figure 5.13: Download completion CDFs for Kudzu and BitTorrent.

initial seeds would quickly spread to the rest of the swarm (resulting in burst download speeds across all downloaders), but then download speeds would quickly slow down as the chunks available to download from the non-seeds were exhausted. We anticipate a sizable speed increase once we add BitTorrent's chunk selection to Kudzu; at present, we simply have not implemented the necessary bookkeeping. Another aspect of BitTorrent that we have not yet implemented is dynamic connection management. BitTorrent keeps roughly 4 peer connections open per download and periodically tries to improve download speeds by switching to connections providing higher average speeds. Kudzu, at present, simply connects to everyone in the swarm. This results in many more connections (some of which may not actually facilitate a significant amount of transferred data), which may be slowing the network down relative to BitTorrent.

Another measure we were interested in was the amount of overhead imposed by Kudzu to facilitate the download relative to the amount of actual data transferred between peers. This overhead includes exchanging chunks available to upload, pings to other members of the swarm, and exchanges of peer lists within the swarm. We tallied the total bandwidth used during the download tests and found that the binary download blocks themselves comprised over 98% of the total bandwidth used during the test. This means that the total overhead of the network comprised less than 2% of its bandwidth. This result suggests that the bandwidth overhead of conducting swarm downloads in a purely P2P fashion (that is, without a tracker) does not present a significant scalability issue.

5.7 Summary

Our experimentation with Kudzu led us to several main conclusions, both about Kudzu itself and about general decentralized P2P systems of its kind. We summarize these findings as follows:

- The max TTL setting is highly important to the aggregate bandwidth consumption of the network on its host network (presumably the Internet). Moreover, setting a high max TTL or uncapping TTL entirely is not necessary to achieve good query recall.
- Maintaining a balanced network is extremely important to maintaining query recall in the absence of a more sophisticated organizational strategy. An unbalanced network often leads to a high network diameter and disconnected components, both which can seriously hamper query recall.
- Intelligent peer selection via TFIDF was not as beneficial as we thought relative to a purely random organization. However, clusters formed by TFIDF seem to offset the harm caused by random imbalances in the network.
- BitTorrent-style centralized trackers are not needed to achieve high performance swarm-based downloads; a fully decentralized P2P system such as Kudzu can achieve similar performance without incurring high overhead.

Chapter 6

Conclusion

6.1 Future Work

While Kudzu is a fully functional P2P file transfer system in its own right, there are some important aspects of P2P systems that we did not consider in Kudzu's design. We discuss some of these issues here and how they may be incorporated into future versions of Kudzu. We are also continuing to explore more powerful network organization strategies than the ones evaluated thus far, some of which we detail here.

6.1.1 Organization with Machine Learning Classifiers

Recall from Section 3.3.5 that we discussed an approach to organizing the network using machine learning techniques to learn how to separate good neighbors from bad ones. We believe that an approach such as this has potential in real-world systems based both on simulation data from [5]. However, implementing such a heavyweight organizational approach in a real setting presents several challenges:

- Gathering training data requires a reliable way to label a set of training peers. The obvious way to do this (by first interacting with potential peers) is problematic because it means that the cost of bootstrapping the organization process is very high, and during the process the node will see no organizational benefits. Furthermore, in real P2P systems many users join the network, issue a few queries, and then quickly disconnect – for these users, a long-term organizational strategy is likely to have little benefit.

- Training SVM classifiers and performing feature selection are quite computationally expensive. Training the classifier is likely to take a significant amount of time; as such, one issue is choosing when to retrain the classifier as new data is received from interactions on the network. One example approach that could be used is to retrain the classifier overnight, but this encounters the the problem of short peer longevity discussed above.

However, even with these problems we believe this is an approach worth investigating. The example binary classification task we presented was a fairly simple problem formulation; there is no reason why we need to classify peers only as good or bad, nor do our features need to be restricted to the binary $\{in_file_store, not_in_file_store\}$. Instead, more sophisticated non-binary features could be used; in fact, machine learning techniques are often combined with measures such as TF-IDF (e.g., [10]) in order to improve classification performance. Once the implementation and peer longevity issues inherent to an approach of this type are resolved, there are many types of classifiers that could be employed in intelligently organizing the structure of the network.

6.1.2 Incentive Model and Adversaries

Real-world P2P can suffer both from peers that consume resources without contributing to the network (leeches) and from malicious peers that operate outside of the established protocol either for personal benefit or to simply disrupt the network (adversarial peers). Considering selfish peers and adversaries was beyond the scope of our work thus far, and we simply assumed in design and testing that nodes always acted according to the rules laid down. We noted earlier that conducting network exploration to find suitable neighbors provides an incentive to remain online; at present, however, nodes could simply refuse to upload file blocks or drop all incoming queries without being penalized. We are investigating ways to add an effective incentive model to Kudzu without imposing any centralization – some incentive models (e.g., [22]) rely on trusted third parties to manage incentives, which adds potential weaknesses to the system. An incentive model generally deals with most types of adversarial issues as well, since peers that do not abide by the established rules and conventions will find themselves either limited or blacklisted completely by other peers in the network.

6.1.3 Testing Environment

One of our goals was to evaluate Kudzu under testing conditions as realistic as possible. While we feel that our testing methodology was an improvement on most testing procedures that have been used before, there are several ways in which it could be improved:

- Our largest tests spanned roughly 500-550 nodes, which was as large a subset of PlanetLab as we could harness at once. Unfortunately, current BitTorrent and Gnutella networks often comprise tens of thousands of nodes simultaneously, which obviously causes more stress on the network and more rigorously tests a network's scalability. PlanetLab is the largest synthetic testbed for testing P2P applications easily available to researchers today, however, so scaling our evaluations beyond this scale at the present time would most likely require harnessing actual user machines.

- The Kudzu networks we evaluated were effectively static – though peer connections could change as a result of our organization policies, with only a few exceptions due to PlanetLab's unreliability, nodes that participated in each test participated in the entire test. In live P2P networks, nodes are constantly joining and leaving the network. This high level of node churn presents potential difficultly in effective network organization because the target 'optimal'

organization is in a constant state of flux. Furthermore, from an evaluation perspective, allowing significant node churn complicates deciding what constitutes a possible match in the system – keeping the peers in the network fixed allowed us to easily precompute all possible matches to decide how close to optimal the network was, but this is an unrealistic target in a real setting. Finally, the dataset that we use does not contain the information needed to replicate the actual churn that occurred (peer arrival and departure times). However, deciding how to resolve these problems and incorporate node churn into tests would provide added credibility to our evaluation results.

6.1.4 New Datasets

The Goh dataset [12] we used for our experiments was useful, but we encountered several problems stemming from our data. One was the overall lack of data per user – while the number of users was quite adequate for our purposes, most of those users showed little activity. Given our limited number of simulation machines (and thus simulated users), a dataset with more per-user data could improve our experiments, possibly captured over a longer period or tracing particular users across multiple sessions. A related issue is the lack of uptime data; that is, data per user indicating when the user arrived on the network and disconnected. This type of information is probably quite difficult to obtain in an automated fashion, but would nevertheless facilitate the addition of node churn to our tests. This would also allow us to simulate more users by replacing a disconnected user with a newly connected user on a single PlanetLab machine, thereby simulating multiple users per machine over the course of the entire simulation. To these ends, we are investigating other datasets for use in future evaluations.

6.1.5 Anonymity and Privacy

One final aspect of P2P systems we have not considered is the degree to which the activity of nodes is shielded from other nodes (who they may be interacting with). The present version of Kudzu includes the requester address in every query; thus, every query that is made effectively exposes the behavior of the user to the entire network. This is generally an undesirable property. The original version of Gnutella attempted to correct for this by forwarding query results back through the network along the path the query arrived rather than making a direct connection back to the requester. This approach meant that nodes did not know whose query they were viewing or responding to. While generally functional, this approach not only imposed a much greater bandwidth overhead but created problems when nodes somewhere along the intermediate route disconnected from the network, breaking the chain back to the query requester. For these reasons, this approach was ultimately scrapped and changed back to the simpler direct connection method that we employ in Kudzu. However, privacy and anonymity are still important concerns in a P2P network that we may investigate for future use in Kudzu.

6.2 Summary of Contributions

This thesis presented Kudzu, a fully decentralized P2P file transfer system that employs intelligent network organization to reduce bandwidth costs and improve query recall. Kudzu provides both the flooding, keyword-search querying behaviors of Gnutella and the fast swarm-based downloads of BitTorrent by overlaying download swarms on top of the main network through which queries propagate. We leverage the correlation between node's files and queries to choose peers that are good candidates for future interaction and demonstrate that this approach has potential to greatly improve decentralized P2P networks by lowering the percentage of the network through which queries need to propagate.

In addition, we presented a distributed test harness for running live tests of P2P systems such as Kudzu on real user data. This test framework replays user data on a real network in order to evaluate the performance of the system under real-world settings. We employed this framework to run tests of Kudzu on PlanetLab, and our experiments demonstrated the efficacy of both our network organization and download behaviors. Our tests also show that our system imposes only a modest real-world bandwidth cost under realistic usage patterns.

We now briefly revisit the goals that we set for Kudzu in Section 3.1. The network is completely decentralized and relies on nothing to function correctly besides the nodes themselves, as we intended. Our network organization allows us to limit the maximum query TTL to small values, and our tests on PlanetLab running large networks suggest that Kudzu is highly scalable. The system provides both full-featured keyword searches and high performance download performance, as we desired. Finally, we have demonstrated the real-world viability of our system by implementing and evaluating it under realistic network and usage conditions. Our experiences with Kudzu have demonstrated the importance of network organization (even on a rudimentary level), as well as the feasibility of fully decentralized P2P systems to accomplish the same functions as less-decentralized systems in use today. Given these findings, we anticipate that fully decentralized systems will see increasingly widespread use in the future.

Bibliography

[1] Adar, E., and Huberman, B. Free riding on gnutella. *First Monday 5* (2000).

[2] Ante, S. E. Inside napster. *Business Week* (August 2000).

[3] Bangeman, E. Bittorrent use soars as mpaa fights on against p2p sites. http://arstechnica.com/news.ars/post/20080417-bittorrent-use-soars-as-mpaa-fights-on-against-p2p-sites.html, retrieved 22 April 2009.

[4] Beverly, R. An architecture for scalable p2p networks that respects user incentives. Submission to Symposium on Networked Systems Design and Implementation.

[5] Beverly, R., and Afergan, M. Machine learning for efficient neighbor selection in unstructured p2p networks. In *SysML '07: Proceedings of the 2nd USENIX workshop on tackling computer systems problems with machine learning techniques* (2007), pp. 1–6.

[6] Carchiolo, V., Malgeri, M., Mangioni, G., and Nicosia, V. Social behaviours applied to p2p systems: An efficient algorithm for resources organisation. In *2nd International Workshop on Collaborative P2P Information Systems* (2006).

[7] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. Making gnutella-like p2p systems scalable. In *SIGCOMM '03: Proceedings of the 2003 conference on applications, technologies, architectures, and protocols for computer communications* (2003), pp. 407–418.

[8] Cohen, B. Incentives build robustness in bittorrent. http://www.bittorrent.org/bittorrentecon.pdf, retrieved 22 April 2009, 2003.

[9] Deutsch, P. Rfc 1950 - zlib compressed data format specification. http://tools.ietf.org/html/rfc1950, retrieved 1 May 2009.

[10] Forman, G. Bns feature scaling: An improved representation over tf-idf for svm text classification. In *Conference on Information and Knowledge Management* (2008).

[11] Frankel, J., and Pepper, T. Gnutella protocol specification v0.4. http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf, retrieved 22 April 2009.

[12] Goh, S. T., Kalnis, P., Bakiras, S., and Tan, K.-L. Real datasets for file-sharing peer-to-peer systems. In *DASFAA* (2005), pp. 201–213.

[13] Google Code contributors. Protocol buffer benchmarks. `http://code.google.com/p/thrift-protobuf-compare/wiki/Benchmarking`, retrieved 20 April 2009.

[14] Google Inc. Encoding - protocol buffers. `http://code.google.com/apis/protocolbuffers/docs/encoding.html`, retrieved 1 May 2009.

[15] Google Inc. Protocol buffers. `http://code.google.com/apis/protocolbuffers/`, retrieved 22 April 2009.

[16] Liang, J., Kumar, R., and Ross, K. Understanding kazaa. `http://cis.poly.edu/~ross/papers/UnderstandingKaZaA.pdf`, retrieved 22 April 2009, 2004.

[17] Locher, T., Moor, P., Schmid, S., and Wattenhofer, R. Free riding in bittorrent is cheap. In *Proceedings of HotNets V* (2006).

[18] Loewenstern, A. Bittorrent dht protocol (draft). `http://www.bittorrent.org/beps/bep_0005.html`, retrieved 22 April 2009.

[19] Loo, B. T., Huebsch, R., Stoica, I., and Hellerstein, J. M. The case for a hybrid p2p search infrastructure. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems* (2004).

[20] MPAA. Swedish authorities sink pirate bay. `http://www.mpaa.org/press_releases/2006_05_31.pdf`, retrieved 22 April 2009.

[21] Peterson, L., Bavier, A., Fiuczynski, M., and Muir, S. Experiences building planetlab. In *Proceedings of the 7th symposium on operating systems design and implementation* (2006), pp. 351–366.

[22] Peterson, R. S., and Sirer, E. G. Antfarm: Efficient content distribution with managed swarms. In *USENIX Symposium on Networked Systems Design and Implementation* (2009).

[23] Pouwelse, J., Garbacki, P., Wang, J., Bakker, A., Yang, J., Iosup, A., Epema, D., Reinders, M., van Steen, M., and Sips, H. Tribler: A social-based peer-to-peer system. *Concurrency and Computation: Practice and Experience 20* (February 2008), 127–138.

[24] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications* (2001), pp. 161–172.

[25] RITTER, J. Why gnutella can't scale. no, really. http://www.darkridge.com/~jpr5/doc/gnutella.html, retrieved 1 May 2009, February 2001.

[26] ROHRS, C. Keyword matching. http://wiki.limewire.org/index.php?title=Keyword_Matching, retrieved 17 April 2009.

[27] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Lecture Notes in Computer Science* (2001), pp. 329–350.

[28] SALTON, G., AND BUCKLEY, C. Term-weighting approaches in automatic text retrieval. In *Information Processing and Management* (1988), pp. 513–523.

[29] SANDVINE INC. 2008 analysis of traffic demographics in north-american broadband networks. http://www.sandvine.com/general/documents/Traffic_Demographics_NA_Broadband_Networks.pdf, retrieved 22 April 2009, June 2008.

[30] SINGLA, A., AND ROHRS, C. Ultrapeers: Another step towards gnutella scalability. http://www.limewire.com/developer/Ultrapeers.html, retrieved 1 May 2009.

[31] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *IEEE/ACM Transactions on Networking* (2001), pp. 149–160.

[32] SUN MICROSYSTEMS, INC. Remote method invocation. http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp, retrieved 22 April 2009.

[33] ZENNSTROM, N., FRIIS, J., AND TALLINN, J. The fasttrack protocol. http://cvs.berlios.de/cgi-bin/viewcvs.cgi/gift-fasttrack/giFT-FastTrack/PROTOCOL?view=markup&content-type=text\%2Fvnd.viewcvs-markup&revision=HEAD, retrieved 15 December 2008.

[34] ZHU, Y., YANG, X., AND HU, Y. Making search efficient on gnutella-like p2p systems. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium* (2005).