Kudzu: A Self-Balancing P2P File Transfer System

Sean Barker, Marius Catalin Iordan, Jeannie Albrecht
Williams College
{09sb,09mci,jra1}@williams.edu

Barath Raghavan
UC San Diego
barath@cs.ucsd.edu

Abstract

Many peer to peer file sharing systems to date rely in part upon centralized components or rigid network structure. The former approach typically sacrifices robustness for efficiency whereas the latter sacrifices efficiency for robustness. In this paper we present Kudzu, a system that is both fully decentralized and unstructured, yet outperforms today's most popular file sharing system, BitTorrent. Key to Kudzu's design is its automatic peer structure rebalancing, which obviates the usual explicit peer management protocols used by peer to peer systems. We examine the design of our network architecture and discuss its relationship to existing file transfer systems. We also provide an empirical evaluation of our current implementation's performance on a series of tests conducted on the PlanetLab testbed. We evaluate download speeds, load capabilities, and bandwidth efficiency.

1 Introduction

Nearly one decade ago, the simple yet powerful concept of peer to peer network protocol design broke into the public arena. As each generation of peer to peer systems has risen, thrived, and withered—from Napster to Gnutella to Kazaa to BitTorrent—a pattern has emerged: systems that rely upon centralization in part or in whole, despite an auxiliary peer to peer structure, thrive and gain wider adoption. BitTorrent is the best current example of this phenomenon, as a system that relies predominantly upon a centralized "tracker" to act as a directory service for hosts that wish to share files. In part, the fear of a fully decentralized approach—that of the original Gnutella protocol—is justified. To combat this, researchers developed myriad "structured" peer to peer systems which typically contain at their heart a Distributed Hash Table. However, maintenance of this structure presents its own complexities.

In this paper, we revisit the idea of peer to peer file sharing without centralized control *or* structure. In other words, we ask: is it possible to produce an efficient file sharing system using a Gnutella-like structure without any explicit maintenance overhead? We present our approach and the implementation of our system, Kudzu, in which peers incrementally and automatically select neighbors in the peer graph to maximize network performance. A key feature of Kudzu is that nodes require no explicit maintenance protocol or network probing, yet still rebalance peer connections in a manner that maximizes performance. We focus our efforts on designing a balancing procedure that works and is simple both in concept and in implementation; as a result, we do not attempt to formulate our learning approach more formally.

Specifically, Kudzu was designed according to the following core principles:

1. The system must be truly decentralized; any peer may perform every function in the network. Additionally, the system should intelligently organize itself without central information.
2. If a user queries for a file residing somewhere on the network, the query is guaranteed to find the file. Furthermore, the file must be found in a reasonable amount of time.
3. If a requested file exists on multiple peers in the network, all available peers should contribute to sending the file in order to maximize the requester's download speed.
4. The network must gracefully and seamlessly handle a rapidly shifting network topology in which peers are constantly entering and leaving the network.

We show that despite Kudzu's fully-decentralized, maintenance protocol-free approach, it still outperforms BitTorrent in file transfer tests. Furthermore, we have evaluated Kudzu on PlanetLab and offer our observations on how Kudzu's automated rebalancing can be applied to numerous existing peer to peer systems.

2 Design Overview

Kudzu is designed to operate as a loosely organized set of nodes that are individually responsible for ensuring an efficient overall network topology. Rather than imposing a predefined structure, a set of Kudzu peers constitutes a collection of node-to-node connections with high redundancy. The network is functionally homogeneous, though some nodes may become more highly connected than others due to high available bandwidth.

2.1 Node Additions

New nodes join the Kudzu cluster by contacting any node already within the network. The new node establishes a connection with that existing node and asks for a random selection of the existing node's neighbors, then connects to each node in the random selection. A node tests peer connections for liveness every few seconds by pinging each of its known peers using an empty Java RMI call. If a ping fails, the peer is removed from the list of connections. Each node must maintain at least 3 active peers; if a node has fewer than 3 peers, it requests new randomly chosen nodes from its remaining peers with which to establish connections. Nodes always accept new connections from other nodes, even if the accepting node is already at or above the minimum connection quota. This means that if a particular node is randomly chosen frequently by its peers, it may end up with a large list of peers—no maximum number of connections is enforced.

2.2 Queries

Kudzu nodes forward queries to all connected peers, who then forward the queries along to all of their peers (except the sender). Queries are asynchronous and return no direct response—they are simply forwarded to all other peers in the network. We impose no TTL, so every node in the network eventually receives every query. Furthermore, since each node is connected to at least 3 others, each query is guaranteed to propagate throughout the network using between $\log_3 n$ and $n/3$ hops, where n is the number of nodes in the network. The exact number will depend on how the network structure evolves. Due to this flooding approach, a given node is likely to receive the same query multiple times from other nodes.

To avoid infinite query propagation, each node maintains a list of the queries received recently so that such duplicate queries may be discarded. A query is considered a duplicate if the last identical query was received within some timeframe (currently set at 20 minutes). An "identical" query here refers to a request for the same filename from the same host. In Kudzu, search queries not only help nodes locate files, but also help balance the network topology. Since nodes receive some queries more than once, our aim is to minimize query propagation overlap while also propagating queries quickly and preventing mesh partitions.

Queries are instrumental to our rebalancing algorithm. To accomplish rebalancing, each node keeps count of how many times it sees each query and the identity of the node from which it received the query. If the node sees the same query more than 3 times (where 3 is also the minimum number of connections maintained by each node), it removes a connection to one of the nodes that sent one of the duplicate queries. Since nodes do not forward queries back to the node that sent them, this heuristic is guaranteed never to split the network in two. Instead, it simply serves to redistribute connections from heavily and redundantly-connected nodes to less-connected ones.

As with new connection requests, nodes always obey disconnect requests from sending nodes, even if it brings the receiving node down below its minimum connection threshold. When this occurs, the node will then simply follow the usual procedure of connecting to new peers. Thus, disregarding any other ways in which connections are established in the network, the sending of queries across the network will, with high probability, cause it to converge to a 'stable' state in which every node is connected to exactly 3 other nodes (see Section 4.4). There are several exceptions to the connection removal rule, however, which we discuss in Section 3.

2.3 File Transfers

Each node maintains an index of the files it has available to share on the network. When a node receives a query for a file in its index, it sends a reply to the requesting node (whose identity is embedded in the query) and also specifies the file's name and size. Note that nodes that have the requested file still forward the query on to all of their neighbors, allowing multiple senders to serve a file simultaneously, thereby hastening the transfer.

Each time a new response is received for a file query, the downloading node opens a new connection to the host with the file (if one does not already exist) and begins retrieving 50 KB blocks. When multiple hosts respond with the file, multiple streams are opened and different chunks are simultaneously downloaded. A key design choice is that these new connections to file-source hosts remain open after the download is complete and are treated exactly the same as other connections.

Each connection keeps track of the number of blocks that have been sent or received along the pipe since the connection was opened. This serves as a measure of the connection's utility and is used to rank connections when removing a connection as a result of excessive query propagation. This process is described in detail in Section 3.

3 Machine Learning

We construct a P2P network model based on node homogeneity and individual simple action. Kudzu establishes an evolving mesh network topology that adapts to external and internal pressures by rebalancing connections between nodes to better reflect the current needs of users.

Each node's primary concern is maintaining a relatively fixed degree of connectivity. If a node finds itself in a position where it is connected to fewer nodes than its current threshold, then it actively seeks out other nodes to connect to through querying its neighbors. The upper bound on connections is less strictly enforced and is allowed to change in accordance to previously and cur-

rently observed trends in the patterns of data transfer and query propagation.

In order to adjust the threshold and keep track of activity at the level of individual nodes, we associate a 'transfer load' metric to every node that estimates the average amount of useful data that passes through each of the node's connections at any given time. Initially, each new node starts with a load of 0 on all of its connections and each time a chunk of data (we chose 50 KB as a reasonable partition size of file data) is transmitted across the network, both the sending and receiving nodes increase the transfer load on this connection. To ensure that old data and sparse spikes in activity on a certain connection don't provide a high utility to a connection indefinitely, we decrease the metric by a factor of 2 after a carefully selected timeout value.

Initially, the connectivity minimum threshold is 3 nodes. Our approach is adaptable and can handle connection levels far above this default level. The network topology detects an imbalance through the query propagation mechanism. As each query is forwarded to all neighbors (with the exception of the one from whom it was received), well-connected nodes begin to receive many duplicate queries, prompting them to initiate connection drops. When a node decides to drop a connection, the natural approach would be eliminating the one with the lowest transfer load. However, the algorithm incorporates factors beyond simply maintaining the old threshold level for number of connections.

There are three special cases in which the node does not drop the connection, but instead modifies its threshold to accommodate for another useful connection. First, if the connection is heavily and consistently used, we keep it, since it signals that the current node is important to the rest of the network; one of our primary design goals is to aid information propagation above all else. Second, nodes that have very recently started downloading from or uploading to this node are granted immunity from being dropped for a period of time, after which the usual rules apply and the connection may be subsequently dropped. Third, newcomer nodes to a network are granted similar immunity until they have a reasonable chance to begin uploading or downloading. Thus, an incoming node will not have to bounce around the network and will be given the opportunity to establish stable connections while it searches for or begins to transfer files.

Our adaptable mechanism for maintaining a variable node connection count ensures that the network topology remains balanced in real-time and applies itself to the demands of the users at any given time. As download concentration shifts from one area of the network to another, so do the thresholds for the heavily used areas increase to match this demand. Furthermore, our rebalancing technique is completely decentralized and requires no outside maintenance or oversight. The system runs

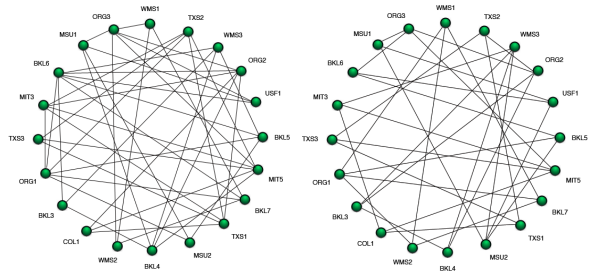

Figure 1: Typical Kudzu network topology: unbalanced (left) and balanced (right).

without supervision, evolves on its own, and adapts to changing environment conditions using only its internal algorithms in a fully autonomous fashion.

4 Evaluation

We evaluated 3 performance measures of Kudzu: download speed, peer/overlay overhead, and connection redundancy (which serves as an imperfect measure of fault tolerance). We also examined the self-adjusting aspect of the network to determine how well the network can balance itself. Each metric is discussed individually below.

4.1 Network Topology

A Kudzu network makes no assumptions about how or where nodes are going to join, so the network may be initialized with an arbitrarily unbalanced structure. To evaluate network topology on a simple network, we began with a single Kudzu node and then had twenty PlanetLab [7] nodes start and connect to the first node. We then disconnected the first node from the network, forcing the twenty remaining nodes to randomly select new neighbors. The resulting network was fairly unbalanced, as some nodes were randomly selected more often than others to replace the initial node as the third neighbor.

To test our rebalancing scheme, we then had two different nodes sequentially send 10 queries each to the rest of the network. There should be no reason to expect different results from this process than from having all nodes send queries, as any given node n will drop connections as long as at least one other node is sending queries (which will propagate with duplicates to n). The former and resulting networks as a result of rebalancing are shown in Figure 1.

We see that the original network has a high (and, we argue, excessive) degree of redundancy. Some nodes have as many as 7 connections, while others have only 3, so these inconsistent additional connections are generally undesirable. After automatic balancing as a result of queries sent across the network, the network converges to the minimum desired number of connections per node: no node has more than 4 connections. Connections are also relatively evenly distributed across the network.

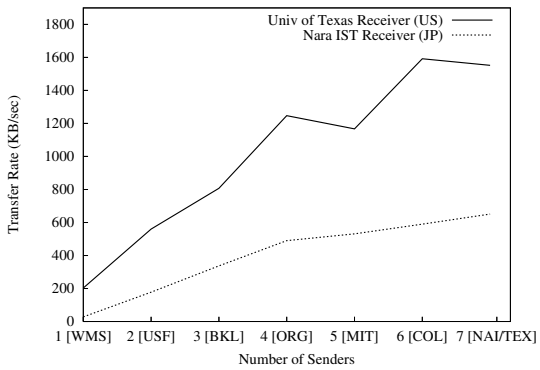

Figure 2: Total download speeds for a single file being transferred.

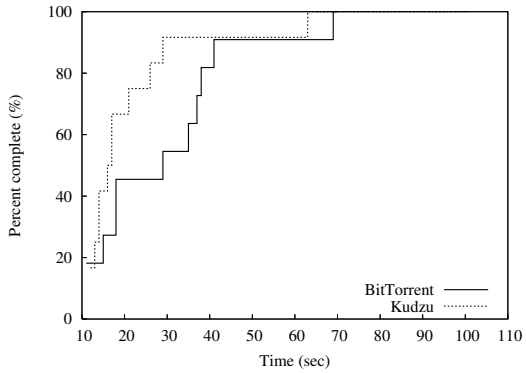

Figure 3: CDF showing percentage of nodes completing file download for Kudzu and BitTorrent.

4.2 Download Performance

To test download performance, we initialized a network of 20 nodes consisting of PlanetLab machines located at 8 sites spread across the continental United States. Tests were conducted to see how well download speeds would scale when multiple peers shared the same file with a single receiver. We ran the tests for two receivers: one from a centrally located, low-latency node (at the University of Texas) and one from a remote, high-latency node (at Nara IST in Japan). Average ping delays to the sites from Massachusetts were 60 ms and 200 ms, respectively.

In each test we transferred a 38 MB file to a single machine from a certain number of senders. We measured the average download speed over the the entire file, then added another sender and ran the test again. Up to 7 senders were evaluated in this way for each receiver. The results of our tests are displayed in Figure 2. The results demonstrate that Kudzu takes full advantage of available bandwidth. We see dramatic speed increases as the number of sending nodes increases. Furthermore, the transfer speeds attained by the Nara IST node show that even a remote node in geographic terms may see significant speed increases by downloading from multiple senders simultaneously. Our tests do not reveal any performance plateau, and it is likely that we may achieve even greater performance by adding more senders —we assume that maximal download speed is ultimately limited only by the total bandwidth available at the receiver.

To evaluate our system's transfer potential in relation to existing P2P systems, we performed a test on PlanetLab that compared download speeds on our client to a standard BitTorrent client. For both our network and BitTorrent, we seeded a 50 megabyte text file to 12 initial peers scattered around the globe and then had 12 more peers simultaneously connect to the network and download the file. The cumulative distribution function of completed downloads over three averaged runs per network are shown in Figure 3. Overall, our system compared favorably in raw download speeds, even running on top of an unoptimized, synchronous RMI implementation. We anticipate easily achievable speed gains on top of this through implementation optimizations.

4.3 Communication overhead

To simulate a realistic scenario, we created a network using as many PlanetLab nodes as possible. As before, we started a single entry node and then connected as many nodes on PlanetLab as we could contact; we were able to sustain roughly 420 of PlanetLab's 864 nodes during the majority of testing.

To evaluate the network's bandwidth overhead, we must first differentiate between 'good' bandwidth and 'bad' bandwidth. Good bandwidth refers to queries that a node receives that it has not yet received—these unique queries impart new information to the node, and are critical to the network's functionality. However, we wish to minimize duplicates. Thus, a suitable measure of bandwidth efficiency in the network is the ratio of unique to duplicate queries received.

Each node ran a test client that automatically sent 20 queries spread out in random intervals of 5 to 20 seconds. We ran two types of load tests: one where no files were present on the network (fileless test), and one where files were present (file download test). In the latter case, we selected 3 nodes and placed the same set of 10 files (each of size 10 MB) into the three node's shared directories. The three nodes were located in the United States, Europe, and Asia, and each had low relative latency. When testing with the 3 'server' nodes (though this is technically a misnomer, as they ran exactly the same software as every other node), all test clients queried for a nonexistent file 80% of the time (by querying for the local timestamp) and requested one of the 10 actual files (selected at random) 20% of the time. As per the usual Kudzu process, when a file completed downloading to a node, that file was added to the node's list of shared files and could be retransmitted to other nodes.

In both cases, every client tallied the number of unique

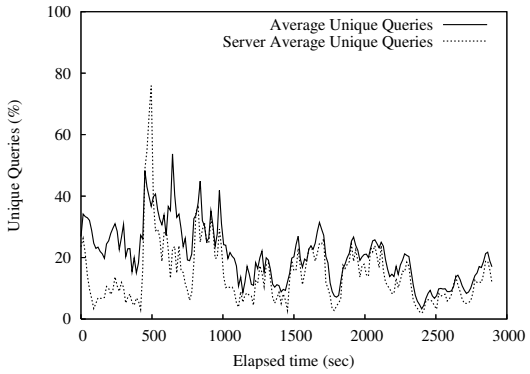

Figure 4: Unique query percentages in file download test.

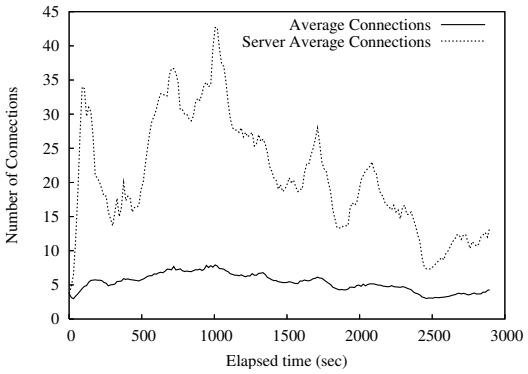

Figure 5: Number of peer connections in file load test.

and duplicate queries received. Every 15 seconds, the current counts were sent to a centralized monitor to aggregate and then reset. The monitor averaged over all values received in each 15 second window to output the final results. The file and fileless tests were run for approximately 2 hours, and approximately 15 minutes respectively.

The overall average proportions of unique queries to total queries (unique + duplicates) for the file download test and the fileless test were 20% and 40% respectively. When no files are present (and thus, no downloads occur that create new connections), the network should quickly stabilize to nearly 3 connections per node. Thus, one would expect 33% unique queries, as each query would be received once on each connection. Our result of close to 40% indicates a significant increase in efficiency as a result of the way in which the network balances—many duplicate queries are removed from the network before they have a chance to extensively propagate. In the file download test, files are constantly downloading, resulting in a greater average number of connections; this explains the lower percentage of unique queries. Results of the file download test at each 15 second time slice are graphed in Figure 4. Results are shown for both the entire network and for the three 'server' nodes alone. Again, we see worse overall performance from the server nodes as a result of their higher number of connections than the rest of the network. Given the number of connections established to the server (discussed below), it would be difficult to expect high percentages from these nodes.

4.4 Connection Redundancy

In addition to evaluating the proportion of unique queries, we also recorded the current number of connections in each data point sent to the monitor. This represents one of the cases in which we would like the network to allow certain nodes to have many connections —the server nodes should be able to maintain many connections, so that other nodes querying for files will encounter the servers quickly. The results over the 2 hour test period are displayed in Figure 5.

The server nodes experienced a sustained load of several times that of other nodes. As more nodes make file requests, the number of concurrent downloads happening over the network gradually increases to a peak, and we can see this trend in both node groups (as the ordinary nodes receive complete files and begin serving them to others). As the 20 queries sent by the test clients begin to complete and the number of uploading nodes increases, the differentiation of the server nodes decreases. If allowed to run long enough, we should see both lines intersect and then run roughly in tandem. The average number of connections in the file download test described above and the average number of connections in the fileless test was 5.3 and 3.3 connections respectively. As we predicted, in a static environment (the fileless test), the average number of connections is very close to 3, while in a fluid environment (the file download test), the average number of connections is modestly higher.

5 Future Work

From a high-level perspective, Kudzu's adaptation algorithm tries to approximate the usefulness of a given P2P link in an unstructured network by assessing transfer speed and incoming query relevance for that link. However, one could consider using a more complex decision function which combines a larger set of variables such as average uptime of the peer on the network, node failure rates, or even geographic location, all of which constitute information that is either readily available, or inexpensive to compute. This scenario maps itself quite nicely onto various machine learning algorithms (regression techniques, neural nets, SVMs) which could be used by each node to determine coefficients in a linear combination optimizing to a dependent variable representing network effectiveness.

Other improvements to Kudzu's self-adaptive behavior could most likely be made by tweaking parameters in the network (particularly with regard to timeouts and transfer rate requirements)—or even better, allowing

nodes in the network to tune parameters themselves.

Finally, there are several important issues in P2P networks that we do not address at all, such as anonymity concerns and the problems of dealing with misbehaving nodes. However, nothing in Kudzu precludes addressing such issues in the future.

6 Related Work

Kudzu's architecture was initially inspired by Gnutella's initial decentralized structure [3]. Like Gnutella, Kudzu lacks an index of available files and floods queries throughout the network. However, unlike Gnutella, we impose no TTL value on queries, allowing them to propagate. TTL's are inappropriate in an unstructured network, where "good" nodes (*e.g.*, based on geographic or network locality, overlap in interests, etc.) may find themselves on opposite sides of the network. Ultimately, Gnutella's initial architecture, combining flooding and ad hoc network neighbor selection, proved unscalable, and subsequent architectures adopted a hierarchical tree-based architecture with stable, bandwidth-plentiful super-peers near the root. In contrast, Kudzu addresses the problem without a rigid well-defined structure, but with an emergent structure formed from each node's peer-selection process.

BitTorrent [4] is the dominant P2P file sharing system today. Unlike Kudzu and other decentralized systems, BitTorrent uses centralized trackers and leaves network organization to individual users. Trackers are a single point of failure and impose a burden on the user to find them through out-of-band means. Kudzu leverages BitTorrent's "swarming" phenomenon by adopting its approach to downloading multiple pieces of a file from different peers. We are currently exploring how to adapt BitTorrent's "tit-for-tat" incentive mechanisms to Kudzu.

Recent work proposes a neighbor selection scheme that connects nodes likely to exchange data in the future [2]. In contrast, Kudzu nodes initiate queries to guide the network's organization. Instead of off-line analysis, Kudzu evaluates connection utility on-line based on a transfer-load metric that conveys the usefulness of each link in the node mesh. The technique accommodates sudden shifts in interest and search pattern allowing the structure to adapt as peers' change, obviating expensive training trials to identify overlapping data patterns in the existing network.

Kuhn et al. uses a hashtable approach mitigating the impact of churn in a P2P network given a worst-case oracle adversary [5]. Kudzu does not address churn induced by a worst-case adversary, although it is designed to adapt to churn in the underlying network.

Bernstein et al. focus on ensuring fast data transfer using the Gnutella network topology [1] by employing a Markov Decision Process approach to determine which node to select to download a file. The approach favors connections with high utility. However, Kudzu does not limit the number of possible simultaneous downloads, thus alleviating the need to select between nodes which hold a desired file. As in BitTorrent, the approach allows Kudzu to retrieve portions of a file from slower nodes.

Other previous work proposes to improve Gnutella query flooding by controlling node degree using random walks [6]; Kudzu simply floods queries under the assumption that nodes that cannot handle them are not suitable download candidates. Kudzu does recognize the importance of node degree to performance, but use alters it dynamically based on actual transfers and prunes the connection list accordingly. In contrast, Lv et al. [6] assign a fixed maximum bandwidth potential to each node and attempt to guide traffic in the network by changing connection ownership based on the gap between actual traffic and this pre-defined maximum capacity. Kudzu's design reflects the view that potential bandwidth is not relevant in real-time network conditions, as bandwidth fluctuates on short time-scales.

7 Conclusion

We have found that despite its decentralized, unstructured design, Kudzu is a robust, highly scalable file transfer protocol. By allowing each node to auto-balance its peer connections to maintain both network reliability and performance, the network reaches a dynamic equilibrium depending on the volume of data moving through its links, favoring active nodes and ensuring that queries are answered expediently and files are transferred rapidly. Our tests show that Kudzu has strong potential, as well as possibilities for future improvement. We believe that Kudzu provides a new avenue for unstructured peer to peer file transfer, one that leverages intelligent and independent adjustment of the network by each node.

References

[1] D. S. Bernstein, Z. Feng, B. N. Levine, and S. Zilberstein. Adaptive Peer Selection. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[2] R. Beverly and M. Afergan. Machine Learning for Efcient Neighbor Selection in Unstructured P2P Networks. In *Workshop on Tackling Computer Systems Problems with Machine Learning Techniques (SysML)*, 2007.

[3] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making Gnutella-like P2P Systems Scalable. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, 2003.

[4] B. Cohen. Abstract Incentives Build Robustness in BitTorrent, 2003.

[5] F. Kuhn, S. Schmid, and R. Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[6] Q. Lv, S. Ratnasamy, and S. Shenker. Can Heterogeneity Make Gnutella Scalable. In *Proceedings of the International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[7] L. L. Peterson, A. C. Bavier, M. E. Fiuczynski, and S. Muir. Experiences Building PlanetLab. In *Proceedings of the ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2006.