Structured Superpeers: Leveraging Heterogeneity to Provide Constant-Time Lookup

Alper Tugay Mızrak, Yuchung Cheng, Vineet Kumar and Stefan Savage
Department of Computer Science & Engineering
University of California, San Diego
{amizrak, ycheng, vineet, savage}@cs.ucsd.edu

Abstract

Peer-to-peer (P2P) systems are typically divided into those that centralize lookup functionality in a single location and those that distribute the lookup operation across the set of participating hosts. The former approach can offer constant time lookup latency, but is more expensive to scale and suffers from single points of failure. In contrast, the fully distributed approach is easier to scale and can be more resilient to failures, but the lookup latency scales as a function of the total number of participants. While the research community has made great progress in improving the latency of distributed lookup, these systems, exemplified by Chord[17], typically require $O(logN)$ hops to locate an object in a system with N hosts.

In this paper, we explore the costs and benefits of a new hybrid approach that partially distributes *lookup information among a dynamically adjusted set of high-capacity "superpeers". This design exploits the resource heterogeneity inherent in existing P2P systems to provide many of the advantages of a centralized system, even while avoiding most of the problems associated with such systems. Lookup is performed using superpeers in constant-time, and the system performs well even in the event of simultaneous superpeer failures. Finally, while our gain in performance is potentially at the expense of scalability, we will show that a straightforward implementation should be able to scale to over one million peers with reasonable lookup rates.*

1 Introduction

Peer-to-peer (P2P) systems are designed to distribute functionality and resources among a large number of independent hosts. The promise of this design is that highly distributed state is easier to scale, is more resilient to failure and supports greater administrative autonomy. However, the costs of distribution can be significant as well and consequently the research community has focused its attention on developing efficient *distributed hash table* (DHT) algorithms for routing requests to individual peers. The best of these algorithms, exemplified by systems such as Chord [17], Pastry [12], and Tapestry [20], are able to locate any item after querying no more than $O(logN)$ individual peers. While this is no small feat, it still imposes a significant performance penalty for many purposes. For example, Cox et al. show that while Chord can be used to serve Domain Name Service (DNS) records, the latency incurred is increased by an order of magnitude [1]. They further observe that "the problem becomes worse as the peer-to-peer network grows" since a large system with 1,000,000 nodes might require each lookup to traverse 20 nodes on the overlay network before locating each item of interest.

This latency is a direct consequence of homogeneously distributing index state among peers. If each peer maintains unique state necessary for a successful lookup, then the cost of a query *must* grow with the number of peers. In systems like Chord, this means that some queries *must* take $O(logN)$ steps to be resolved. The opposite extreme, illustrated by Napster [9], is to centralize the index in a single location. While this allows lookups to be completed with a single message exchange, the central index then becomes a potential bottleneck and single point of failure. We believe that neither of these architectures is ideally matched to the capabilities of existing networks of peers and it is time to revisit the design tradeoffs made in modern structured P2P systems.

Measurements of deployed P2P systems all reveal significant heterogeneity in the capabilities and activities of their members [13, 11, 10, 14]. While most peers are short-lived and have minimal bandwidth, a small fraction typically remains connected for extended periods and have significant storage, memory and bandwidth resources. This population heterogeneity suggests that performance may be significantly improved by assigning index state unequally – picking a design point somewhere *between* fully centralized and fully distributed.

This paper describes the design of a structured P2P system that explores this tradeoff. Our approach delivers constant-time $O(1)$ lookup by assigning additional state to these high-capacity peers, or *superpeers*, present in many peer-to-peer systems. Moreover, we argue that this approach can easily support 1,000,000 peers – a scaling point that represents an interesting class of peer-to-peer systems. In the remainder of this paper we briefly discuss related work, describe our system architecture, and then use a combination of analytic and simulated results to explore the scaling issues of this design.

2 Related work

Several popular unstructured P2P applications such as FastTrack [3] and Gnutella [6, 18, 15] have explored using heterogeneity to improve search performance. These systems forward queries to high-capacity superpeers (named SuperNodes, Hubs, UltraPeers, Reflectors, etc. depending on the system) selected based on their capability to process large numbers of requests. A superpeer acts as a local search engine, building an index of the files being shared by each peer connected to it and proxying search requests on behalf of these peers by flooding requests to other superpeers. These systems significantly outperform pure flooding systems by preventing low-performance hosts from transiting requests, but they still rely on flooding at the superpeer-level.

Structured P2P systems typically have not leveraged heterogeneity in their algorithms. A recent exception is Garces-Erice et al.'s two-tier hierarchical lookup design that groups nearby peers based on network latency and communicates between groups using a superpeer layer [5]. To find a peer that is responsible for a key, the top tier overlay network routes among the superpeers to first determine the group responsible for the key; the responsible group then uses an intra-group overlay network to determine the specific peer that is responsible for the key. The authors propose using two different DHT algorithms at each level of the hierarchy, a $O(logN)$ algorithm that maintains $O(logN)$ state among superpeers and an $O(1)$ algorithm that maintains $O(N)$ state within a group. The resulting design completes lookups in $O(logM)$ time where M is the number of superpeers.

Several very recent P2P designs have also offered dramatic reductions in lookup latency. The Kelips system [7] divides peers into groups by subdividing the identification key space. Peers are aware of the files stored on every group member *and* the identity of at least one member of every other group. Consequently, Kelips can achieve a constant-time lookup by either locating an object within its own group or passing the query on to a member of the appropriate group. Both Kelips and our design achieve constant-time lookup (in terms of hop count). However, Kelips does not utilize peer heterogeneity at all. Every Kelips peer is required to store $O(\sqrt{N})$ state, while in our design, only the $O(\sqrt{N})$ superpeers are required to maintain $O(\sqrt{N})$ more storage. To put this in context, a 1,000,000 node P2P system implemented in Kelips would require a total state expenditure of 1 billion entries, while our design requires only 8 million entries.

The Coral [4] system takes a different approach. It focuses on improving the latency of lookup queries rather than reducing hop count. To do this, Coral introduces a hierarchical lookup strategy that divides a single Chord-style ring into three distinct rings called clusters. A Level-2 cluster consists of peers that are mostly within 30msec of one another, while a Level-1 cluster consists of Level-2 clusters within a larger 100msec range. Finally, the Level-0 cluster represents the original Chord ring, which has no latency guarantee. A Coral peer only joins an acceptable cluster. That is, one in which the latency to 90% of the nodes is within the cluster's diameter. If a node cannot find such a cluster, it forms its own. Upon joining a cluster, each new node insert itself into its higher-level clusters, keyed under the IP addresses of its gateway routers, discovered using the `traceroute` tool. A new node, searching for a low-level acceptable cluster, can lookup topologically close peers by using its gateway router addresses as lookup keys. Consequently, lookups within a cluster or between Level1 and Level-2 clusters are expected to be fast, while the overall hop count is still $O(\log N)$. Compared with our approach, Coral does not leverage the heterogeneity among the participants. Since each Coral cluster is in fact a Chord ring, our structured superpeer design can further speed up lookup, by assigning powerful Coral peers to be superpeers. Hence, we conclude that our design is indeed another optimization of the Coral system.

Finally, several recently proposed DHT schemes have demonstrated how to provide $O(\log N)$ hops lookup with constant state per node. Kaashoesk and Karger [8], present *Koorde*, a simple degree-optimal distributed hash table. Koorde is based on the combination of Chord DHT and de-Bruijn graphs [2]. They have shown that if each peer maintains state for only 2 nodes, $O(\log N)$ hops lookup is possible. Increasing the state requirement at each node, Koorde can provide higher fault tolerance and moreover, with a state cost of $O(\log N)$, a lookup can be resolved in $O(\frac{\log N}{\log \log N})$ hops communication, which they prove is optimal. Using this approach the expect hop count to satisfy a query in a 1,000,000 node network is approximately 5.

A related design, motivated by fault-tolerance, is presented by Naor and Wieder [19]. In their system, each node maintains constant state and they demonstrate that this is sufficient to satisfy queries within $O(\log N)$ hops for the random fail-stop failure model. They provide an alternative

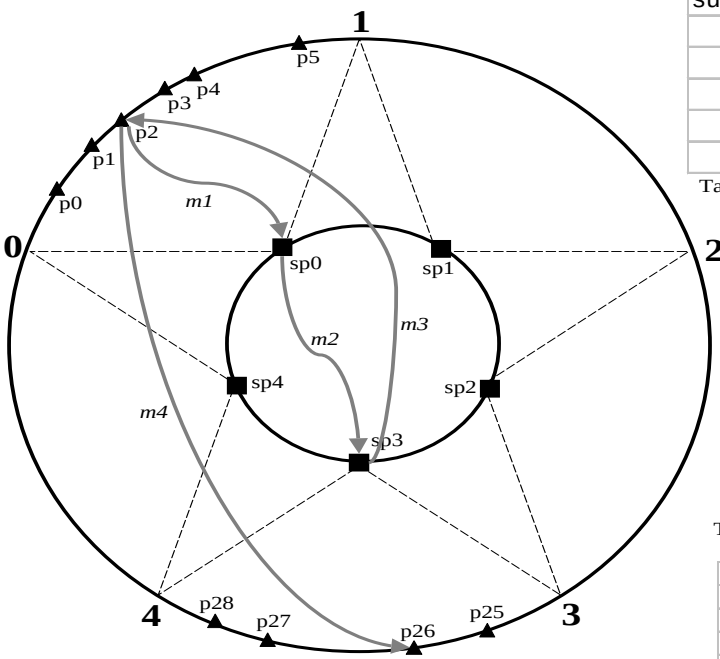

Fig.1. General system architecture

superpeer	add. range
sp0	(0 - 1]
sp1	(1 - 2]
sp2	(2 - 3]
sp3	(3 - 4]
sp4	(4 - 0]

Table 1: Superpeer table

peer	identity
p0	IP0
p1	IP1
p2	IP2
p3	IP3
p4	IP4
p5	IP5

Table 2: Peer table at sp0

peer	identity
p25	IP25
p26	IP26
p27	IP27
p28	IP28

Table 3: Peer table at sp3

lookup algorithm that allows a random subset of nodes to fail in an arbitrarily faulty manner in exchange for an increased cost of $O(\log^2 N)$ hops. Again, the lookup length can be shortened increasing the state requirement per node and their scheme has the same theoretical bounds as Koorde. Furthermore, they have shown that constant time, $O(1)$, lookup is possible with $O(N^\epsilon \log N)$ state. We believe our approach is an instance of this family. In our case, ϵ is chosen to be $1/2$ in order to implement a practically feasible system.

3 System Architecture

Figure 1 illustrates the basic architecture of our system. Given a P2P system with N peers, we place each on a circular identifier space, the "outer ring", using a DHT algorithm such as Chord. Of these N peers, M are chosen to be superpeers to create a smaller core "inner ring". We describe the process of choosing and maintaining the structure of the inner ring later. The outer ring is split into arcs and each arc is assigned to one superpeer. Each superpeer is responsible for maintaining two pieces of information: the addresses of the peers contained within its arc(e.g. $sp0$ maintains Table 2 tracking the IP addresses of all peers, $p0-p5$, on the outer ring between 0 and 1; and $sp3$ maintains Table 3) and the mapping between arcs and their responsible superpeers(e.g. each superpeer maintains a table identical to Table 1).

3.1 Lookup and Query Routing

A lookup is extremely straightforward. When a peer requests a document with key id, it sends this request directly to its superpeer. If the superpeer itself is responsible for the arc including the key id, then it locates the successor of id it its peer table and returns the result. Otherwise, it forwards the request to the superpeer who is responsible for the enclosing arc for id. In response, the responsible superpeer locates the successor peer in its peer table and the result is returned to the peer who initiated the request. In the worst case, two nodes must be contacted to resolve a lookup – one for the immediate superpeer and one for the superpeer responsible for the key – requiring three messages in total.

For example, in Figure 1, peer $p2$ requests a document with a key id value of roughly 3.5. In normal operation, $p2$ sends this request $m1$ to its superpeer $sp0$. Since $sp0$ is only responsible for the arc (0,1], it consults its superpeer table(Table 1) and finds that superpeer $sp3$ is responsible the arc enclosing id, (3,4]. It forwards the request, $m2$ to $sp3$, who in turns looks up the closest successor to id in its peer table(Table 3) and returns the result, $p26$'s IP address, to the original requestor in message $m3$. Finally, $p2$ can connect to $p26$, and issue its request for content, $m4$.

As a result, this system architecture has a constant cost,

$O(1)$. If the peer generating a lookup request has the same superpeer as the peer it is looking for, then only one connection is sufficient to resolve the request: the immediate superpeer. In the worst case lookup – where the correspondent superpeers are different – the number of nodes that must be contacted is two: one for the immediate superpeer, and one for the superpeer responsible for the key.

This design has three key technical challenges: How to bootstrap the system, how to manage failure and how to address changes in load? We address each in turn below.

3.2 Bootstrapping

We assume that the outer ring is managed using a traditional DHT algorithm, such as Chord, and an external mechanism is used to identify members of this ring so new peers may join. As the first k peers join the system, these peers are also commissioned as superpeers and placed into the inner ring as well. This provides an initial set of superpeers and a division of the identifier space, putting the system into a consistent state. As additional peers join the system, they obtain the identify of their superpeer from their immediate neighbors in the outer ring. This peer sends a join RPC to the superpeer who in turn updates the peer table for its arc.

We also assume the each new peer identifies itself to a *volunteer service*, which maintains the resources that a peer is willing and able to contribute to the system. This metric can be computed based on the lookup message processing power and/or storage capability. We assume that the *volunteer service* is provided as a black box, it might be implemented as a centralized or distributed service. This service is used by the superpeers in the inner ring to select additional superpeer candidates in response to increased load or superpeer failure.

3.3 Detecting and Managing Failure

There are two qualitatively different kinds of failures that must be tolerated: peer failures and superpeer failures.

If a peer fails, or leaves the system, the principal issue is updating the peer table at its responsible superpeer. If a peer leaves gracefully it can simply contact its superpeer directly. If the peer fails unexpectedly, this is detected by its neighbors who periodically issue local keep-alive messages (identically to the scheme described in [17]). When a peer detects that its neighbor has failed, it communicates this information to its superpeer in turn. In all cases, the superpeer simply removes the failed peer from its table.

If a superpeer fails, this presents two issues. First, all of the peers in the arc previously managed by the superpeer must be reassigned to one or more replacement superpeers. Second, all other superpeers must be informed about these replacements so they may update their arc tables.

We detect superpeer failures in a similar fashion to peer failures. Each superpeer sends periodic keep-alive messages to its neighbors in the inner ring. If a neighbor cannot be reached then the probing continues with further neighbors until the scope of the unmanaged arc can be determined. Depending on the size of this range, one or more new superpeers may be created from peers listed in the volunteer service or the range may be assigned to neighboring superpeers. In either case the changes to topology of the inner ring are distributed to all superpeers. Note that this operation is the only action that takes more than constant time to complete.

To recover the lost peer to superpeer state, each superpeer replicates the peer information at k of its inner ring neighbors, providing protection against $(k-1)$ consecutive failures (we expect a small value of $k = 2$ or $k = 3$ to suffice). Consequently, when a replacement superpeer is inserted into the inner ring, it can obtain the contents of its peer table directly and quickly from its immediate neighbors. To ensure that these replicas are consistent, each superpeer must update its neighbors with any changes to its peer table that occur. If this system fails due to many consecutive superpeer failures, a new superpeer can still reconstruct its peer info by traversing the peers in its arc from one end to another, using the successor list in the outer ring.

3.4 Load balancing

There are several reasons why a superpeer may receive more load than it can handle. Superpeers are themselves heterogeneous and may vary significantly in storage capacity or message processing power. Similarly, the popularity of content may be non-uniform and particular arcs may experience more requests than others. In these cases it may be necessary to move load among superpeers or change the number of superpeers in the system.

We assume that each superpeer knows its maximum capacity and measures the current load driven by the request rate. When a superpeer's load approaches to capacity, it may share its load with its neighbors if they have sufficient excess capacity or with a new superpeer from the volunteer service. In either case the superpeer splits its arc appropriately and reassigns pieces of this range to the neighbors accepting the load. Conversely, if a superpeer has sufficient excess capacity than it may absorb the entire load of a neighbor and return that neighbor to the volunteer service list.

To tune the load balancing behavior we define four limit parameters for each superpeer: min, max, $lower$, and $upper$. The first two represent hard limits on the capacity of the superpeer, while the later two are soft limits meant to

initiate load balancing activities long before a superpeer is overwhelmed or idle.

There are four criteria that cause a load balancing operation to occur:

- if the load of a superpeer exceeds the hard limit max
- if the load of a superpeer goes below the hard limit min
- if the cumulative load of three neighbor superpeers exceeds the soft limit $upper$ times three
- if the cumulative load of three neighbor superpeers goes below the soft limit $lower$ times three

Depending on the cumulative load of three neighbor superpeers (e.g. smaller than $3{\cdot}lower$ or bigger than $3{\cdot}upper$), the load balancing algorithm either shifts load to the neighbors, introduces a new superpeer or dismisses an existing superpeer. While each comparison and action is entirely local, the system converges in steady state so that all superpeers are loaded between their soft limits and the total number of superpeers in the system is within a constant factor of optimal.

4 Scalability Analysis

In this section we analyze the scalability of the approach we have described, including the storage overhead incurred, the lookup processing required, and the effects of maintenance traffic. We explore these issues analytically to capture steady state behavior and through simulation to explore the impact of membership dynamics on our approach.

4.1 Storage requirements

Each superpeer has to maintain a record to track every other superpeer in the inner ring (including information such as IP address, node id, address range of its arc, network connection state, etc.) and a record for each peer in its address range (including IP address, node id, network connection state, etc.) If a superpeer record consumes c_s bytes, and a peer record requires c_p bytes, then the total storage at each superpeer is:

$$S = c_s M + c_p(\frac{N}{M})$$

The storage requirements at each superpeer is minimal when $M = \sqrt{\frac{c_p N}{c_s}}$. Under this condition, the storage at each superpeer is $S = 2\sqrt{c_s c_p N}$, and the total extra storage required in the whole network is $S_{total} = c_s M^2 + c_p N$, a value that is surprisingly small.

Mapping this to a particular example, in an overlay network of $N = 10^6$ peers and $M = 10^3$ superpeers, each superpeer must maintain only 2000 additional records. For comparison, if this state were used instead to improve the performance of a Chord-based network, each peer would only receive 2 extra records in their finger table (which is unlikely to have a significant effect). Similarly, the Pastry system [12] must contact $\lceil \log_B N \rceil$ nodes to satisfy a query, given $(B-1)\lceil \log_B N \rceil$ routing entries per node – where B is the number of address bits that each peer matches in a round. To support 1,000,000 nodes with a lookup cost of only two hops, then B must be equal to 1000, and *all 1,000,000 peers* must maintain a routing table with 1998 entries. We achieve the same lookup performance with roughly the same storage requirement imposed at only *1000 superpeers*.

Therefore, for reasonably large peer-to-peer networks, the storage overhead is not a significant scalability limitation.

4.2 Lookup processing

Superpeers must be able to service all the lookups they receive from the peers for whom they are responsible. In addition, they must handle queries from other superpeers. We model the lookup process as a uniform rate of q lookups/second for each peer. At a particular superpeer, the lookup rate is $R_i = 2qN_i$, where N_i is the number of peers in its arc. If all superpeers are homogeneous, assuming a uniform distribution of peers in the key space, then the average lookup rate at a superpeer can be estimated as $R = 2q\frac{N}{M}$.

Again, consider an overlay network of $N = 10^6$ peers and $M = 10^3$ superpeers, assuming that each node uniformly issues $q = 1 \;\; lookup/sec$ for objects that are uniformly distributed throughout the system (a much higher rate that is likely in practice). Under these conditions, the average lookup rate that a superpeer must process is only $R = 2000 \; lookup/sec$. For comparison, Stoica et al's recent $i3$ system is able to process in excess of 25,000 queries per second [16].

Lookup processing does not appear to significantly limit this design either.

4.3 Maintenance traffic

As peers join or leave the system, the responsible superpeer must alert its neighboring superpeers to ensure that peer table state is replicated consistently. The number of messages per node is constant relative to the size of the system and the size of each message is bounded by the number of peers that each superpeer can handle. Another sound of maintenance traffic is the keepalive messages used to detect

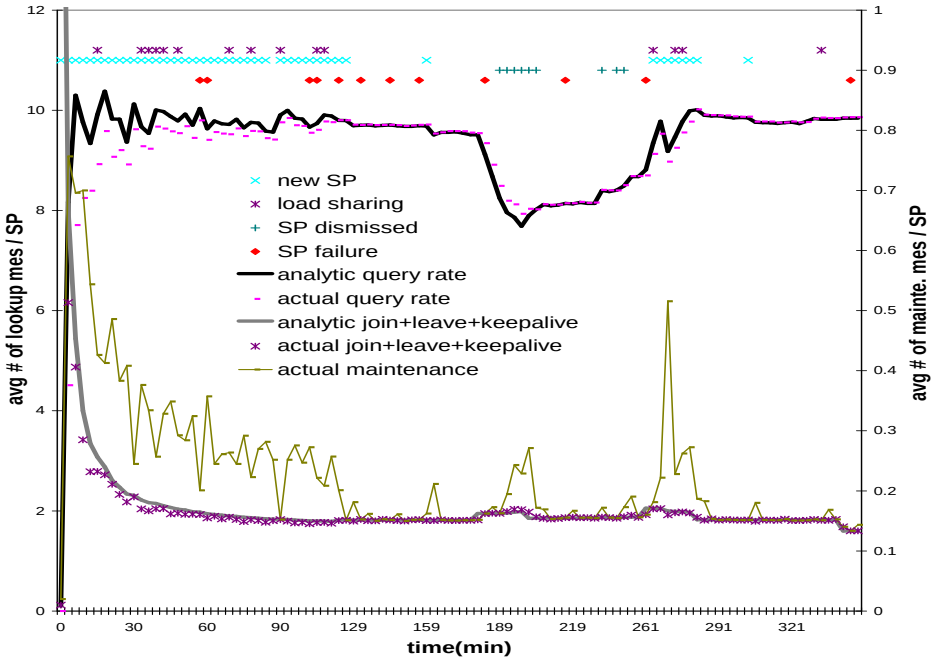

Figure 2. Average lookup and maintenance message rates

failures among superpeers. Similarly, the number of messages per node remains constant for all system sizes as does the message size. Neither of these activities are likely to impact system scalability.

In contrast, the topology changes of the inner ring are relatively expensive. If the superpeer table is changed due to a failure or a load balancing action, then this data must be distributed to all superpeers in the inner ring. While the size of each message is constant, the number of messages to be sent grows linearly with the number of superpeers. However, determining the sensitivity of this overhead is a function of load balancing dynamics that are not well described analytically. Instead, we use an event-driven simulation to explore this issue.

4.4 Simulator methodology

The simulation was performed by modifying the Chord simulator to accommodate the changes in the system architecture. In the simulation, our target environment is around 10,000 peers. The fixed parameters for the system were set as follows:

- the allowed range of peers for each superpeer $(min, lower, upper, max)$ were set to $(55, 67, 113, 125)$.
- the redundancy parameter $k = 2$.
- the lookup rate for each peer was set to q = 0.05 lookup/sec.
- each superpeer failure probability is 0.036 over an hour.
- the keep-alive period for superpeers is 30 sec.

Phase	Duration (min)	Join rate (peer/sec)	Leave rate (peer/sec)
1	120	1.5	0
2	60	1.0	1.0
3	20	0.3	3.0
4	60	1.0	1.0
5	20	3.0	0.3
6	60	1.0	1.0
7	10	0	0

Figure 3. Simulation Phases

We create a synthetic workload, listed in Figure 3, meant to explore several different operating regimes. The first phase is to bootstrap and setup a network of 10800 = 120 * 1.5 * 60 nodes. The second, fourth and sixth phases are to simulate the steady network with equal join and leave rate. The third phase is to simulate heavy leave in a 20 minutes period. The fifth phase is to simulate heavy join in a 20 minutes period. The last phase is a quiet period.

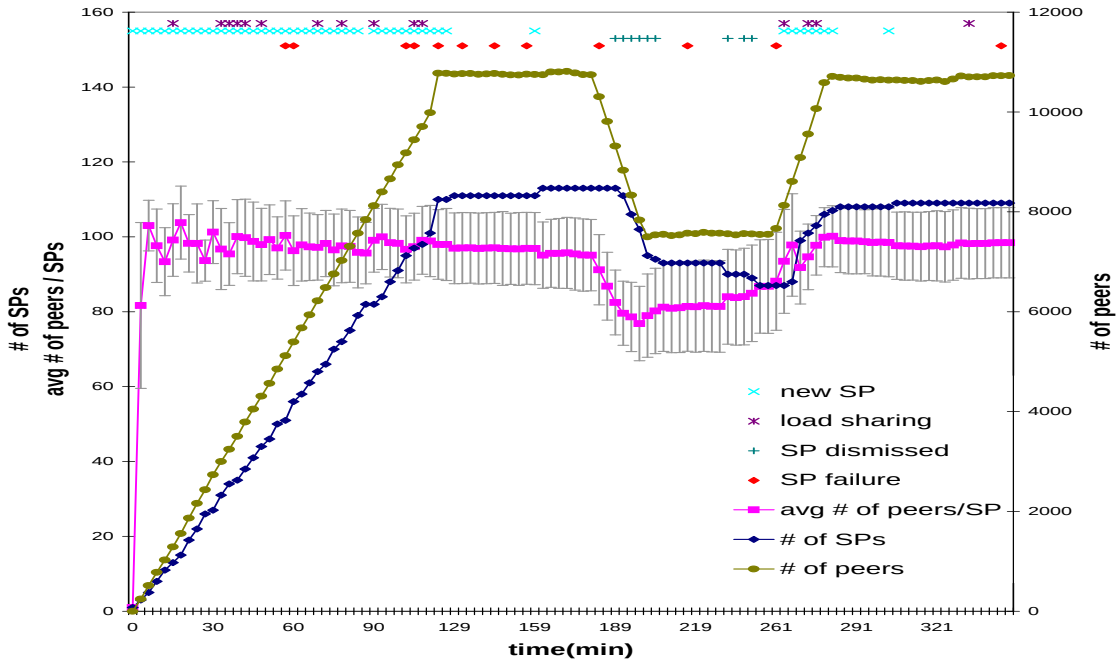

Figure 4. The # of peers, superpeers and average load/superpeer

4.5 Simulation Results

Figure 2 and Figure 4 show the simulation results. System events such as superpeer creation, load balancing actions, and superpeer failures are plotted on the top of both figures.

In Figure 2, the expected average analytic lookup rate($R = 2q\frac{N}{M}$) is drawn and the actual values are plotted over it demonstrating a strong match.

The maintenance messages at superpeers due to peer joining, peers leaving or failing, and the keepalive messages between superpeers are all predictable with the fixed parameters in the simulator. In Figure 2, the expected maintenance traffic rate per superpeer (excluding load balancing messages) is drawn below and the actual values plotted over, again showing a near-perfect match. The maintenance messages for load balancing is unpredictable, because of the nature of the load balancing algorithm and the behavior of the whole system. However all these messages constitute maintenance traffic. The actual average maintenance messages per superpeer is drawn in Figure 2. The cost of the load balancing can be computed as the difference of total maintenance traffic minus the predictable message rate. When compared with the lookup rate, it is seen that the maintenance traffic is quite insignificant.

The simulation results show that the system is highly adaptive to the peer joins or leaves and failures of both peers and superpeers. In Figure 4, the total number of peers can be seen plotted. As the number of peers changes, the number of superpeers and the load on each superpeer are tuned by the load balancing algorithm fairly well. It is seen that the number of superpeers adjusts dynamically to represent the change in load caused by the peers joining and leaving. Moreover, the load at a particular superpeer never exceeds the hard limits and generally stays between the soft limits. This indicates that the load-balancing algorithm is able to balance the load among the superpeers.

5 Conclusion

There are a wide-range of design options in building peer-to-peer systems. In a centralized system, exemplified by Napster, the obvious advantage is that of lower lookup cost, with the associated challenges of scalability and reliability. Decentralized systems, such as Chord, can offer improved scalability and reliability, but with increased lookup overhead relating to the nature of distributed hash tables.

Our approach of using superpeers provides many of the advantages of both centralized systems and distributed systems, while avoiding most of their drawbacks. Superpeer-based lookup can offer constant-time latency, while offering a configurable degree of resilience to superpeer failures. We show that the overhead involved in maintaining the structure of the superpeer ring is low as compared to lookup traffic that the peers generate. Moreover, we demonstrate a

simple superpeer load-balancing algorithm, and show that it provides an equitable and achievable load distribution. While increased centralization does ultimately limit scalability, we show that our proposed system could reasonable scale over one million peers for reasonable lookup rates. Consequently, this approach is a reasonable design choice for most realistic system deployments. Finally, we note that our approach can easily be integrated into existing P2P systems as a performance optimization to the existing lookup algorithm rather than as a replacement.

Acknowledgments

We like to thank Ion Stoica for his valuable comments on our system design and his help on modifying Chord Simulator. We also are grateful to Ranjita Bhagwan for her comments on earlier drafts of this paper.

References

[1] Russ Cox, Athicha Muthitacharoen, and Robert Morris. Serving DNS using Chord. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, Cambridge, MA, March 2002.

[2] N.G. de Bruijn. A combinatorial problem. *Kononklijke Nederlands Akademi van Wetenchappen*, 49(2):758–764, 1946.

[3] Kazaa Media Desktop. http://www.kazaa.com.

[4] Michael Freedman and David Mazieres. Sloppy hashing and self-organizing clusters. In *2st International Peer To Peer Systems Workshop (IPTPS 2003)*, Berkeley, CA, USA, February 2003.

[5] Luis Garces-Erice, Keith W. Ross, Guillaume Urvoy-Keller, and Ernst W. Biersack. Hierarchical peer-to-peer look-up services. In *submission to INFOCOM2003*, 2002.

[6] Gnutella. http://www.gnutella.com.

[7] Indranil Gupta, Kenneth Birman, Prakash Linga, Al Demers, and Robbert Van Renesse. Kelips: building an efficient and stable P2P DHT through increased memory and background overhead. In *2st International Peer To Peer Systems Workshop (IPTPS 2003)*, Berkeley, CA, USA, February 2003.

[8] Frans Kaashoek and David R. Karger. Koorde: A simple degree-optimal hash table. In *2st International Peer To Peer Systems Workshop (IPTPS 2003)*, Berkeley, CA, USA, February 2003.

[9] Napster. http://www.napster.com.

[10] Matei Ripeanu and Ian T. Foster. Mapping the gnutella network: Macroscopic properties of large-scale peer-to-peer systems. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 85–93. Springer-Verlag, 2002.

[11] Matei Ripeanu, Adriana Iamnitchi, and Ian Foster. Mapping the gnutella network. *IEEE Internet Computing*, 6(1):50–57, 2002.

[12] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[13] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[14] Shubho Sen and Jia Wang. Analyzing p2p traffic across large networks. In *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2002.

[15] Anurag Singla and Christopher Rohrs. Ultrapeers: Another step towards gnutella scalability. Whitepaper, 2002.

[16] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *In Proc. ACM SIGCOMM Conference (SIGCOMM'02), pages 73–88, August*, 2002.

[17] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[18] Kelly Truelove. Gnutella and the transient web. Whitepaper, 2002.

[19] Udi Wieder and Moni Naor. A simple fault tolerant distributed hash table. In *2st International Peer To Peer Systems Workshop (IPTPS 2003)*, Berkeley, CA, USA, February 2003.

[20] Ben Y. Zhao, John Kubiatowicz, and Anthony D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.