Designing a Super-Peer Network

Beverly Yang   Hector Garcia-Molina
{byang, hector}@cs.stanford.edu
Computer Science Department, Stanford University

Abstract

A *super-peer* is a node in a peer-to-peer network that operates both as a server to a set of clients, and as an equal in a network of super-peers. Super-peer networks strike a balance between the inherent efficiency of centralized search, and the autonomy, load balancing and robustness to attacks provided by distributed search. Furthermore, they take advantage of the heterogeneity of capabilities (e.g., bandwidth, processing power) across peers, which recent studies have shown to be enormous. Hence, new and old P2P systems like KaZaA and Gnutella are adopting super-peers in their design.

Despite their growing popularity, the behavior of super-peer networks is not well understood. For example, what are the potential drawbacks of super-peer networks? How can super-peers be made more reliable? How many clients should a super-peer take on to maximize efficiency? In this paper we examine super-peer networks in detail, gaining an understanding of their fundamental characteristics and performance tradeoffs. We also present practical guidelines and a general procedure for the design of an efficient super-peer network.

1 Introduction

Peer-to-peer (P2P) systems have recently become a popular medium through which to share huge amounts of data. Because P2P systems distribute the main costs of sharing data – disk space for storing files and bandwidth for transferring them – across the peers in the network, they have been able to scale without the need for powerful, expensive servers. In addition to the ability to pool together and harness large amounts of resources, the strengths of existing P2P systems (e.g., [6, 7, 17, 11]) include self-organization, load-balancing, adaptation, and fault tolerance. Because of these qualities, much research has been focused on understanding the issues surrounding these systems and improving their performance (e.g., [5, 12, 21]).

There are several types of P2P systems that reflect varying degrees of centralization. In *pure* systems such as Gnutella [7] and Freenet [6], all peers have equal roles and responsibilities in all aspects: query, download, etc. In a *hybrid* system such as Napster [17], search is performed over a centralized directory, but download still occurs in a P2P fashion – hence, peers are equal in download only. *Super-peer networks* such as KaZaA [11] (one of the most popular file-sharing system today) present a cross between pure and hybrid systems. A *super-peer* is a node that acts as a centralized server to a subset of clients. Clients submit queries to their super-peer and receive results from it, as in a hybrid system. However, super-peers are also connected to each other as peers in a pure system are, routing messages over this overlay network, and submitting and answering queries on behalf of their clients and themselves. Hence, super-peers are equal in terms of search, and all peers (including clients) are equal in terms of download. A "super-peer network" is simply a P2P network consisting of these super-peers and their clients.

Although P2P systems have many strengths, each type of system also has its own weaknesses. Pure P2P systems tend to be inefficient; for example, current search in Gnutella consists of flooding the network with query messages. Much existing research has focused on improving the search protocol, as discussed in Section 2. Another important source of inefficiency is bottlenecks caused by the very limited capabilities of some peers. For example, the Gnutella network experienced deteriorated performance – e.g., slower response time, fewer available resources – when the size of the network surged in August 2000. One study [24] found these problems were caused by peers connected by dial-up modems becoming saturated by the increased load, dying, and fragmenting the network by their departure. Peers on modems were dying because all peers in Gnutella are given equal roles and responsibilities, regardless of capability. However, studies such as [22] have shown considerable heterogeneity (e.g., up to 3 orders of magnitude difference in bandwidth) among the capabilities of participating peers. The obvious conclusion is that an efficient system should take advantage of this heterogeneity, assigning greater responsibility to those who are more capable of handling it.

Hybrid systems also have their shortcomings. While centralized search is generally more efficient than distributed search in terms of aggregate cost, the cost incurred on the single node housing the centralized index is very high. Unless the index is distributed across several nodes, this single node becomes a performance and scalability bottleneck. Hybrid systems are also more vulnerable to attack, as there are few highly-visible targets that would bring down the entire system if they failed.

Because a super-peer network combines elements of both pure and hybrid systems, it has the potential to combine the efficiency of a centralized search with the autonomy, load balancing and robustness to attacks provided by distributed search. For example, since super-peers act as centralized servers to their clients, they can handle queries more efficiently than each individual client could. However, since there are relatively many super-peers in a system, no single super-peer need handle a very large load, nor will one peer become a bottleneck or single point of failure for the entire system (though it may become a bottleneck for its clients, as described in Section 3).

For the reasons outlined above, super-peer networks clearly have potential; however, their design involves performance tradeoffs and questions that are currently not well understood. For example, what is a good ratio of clients to super-peers? Do super-peers actually make search more efficient (e.g., lower cost, faster response times), or do they simply make the system more stable? How much more work will super-peers handle compared to clients? Compared to peers in a pure system? How should super-peers connect to each other – can recommendations be made for the topology of the super-peer network? Since super-peers introduce a single-point of failure for its clients, are there ways to make them more reliable?

In this paper, our goal is to develop practical guidelines on how to design super-peer networks, answering questions such as those presented above. In particular, our main contributions are:

- We present several "rules of thumb" summarizing the main tradeoffs in super-peer networks (Section 5.1).
- We formulate a procedure for the global design of a super-peer network, and illustrate how it improves the performance of existing systems (Section 5.2).
- We give guidelines for local decision making to achieve a globally efficient network in an *automatic*, *adaptive* manner (Section 5.3).
- We introduce "k-redundancy", a new variant of super-peer design, and show that is improves both reliability and performance of the super-peer network.

By carefully studying super-peer networks and presenting our results here, our goal is to provide a better understanding of these networks that can lead to improved systems. The design of current file-sharing networks using super-peers is driven mostly by intuition; through analytical characterization and analysis of super-peer networks, we aim to provide the science behind the intuition.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 gives a formal description of the search problem, and outlines the different parameters describing a super-peer network. Section 4 describes the framework for analysis used to generate our results, and Section 5 presents these results in the form of guidelines.

2 Related Work

There are several existing studies on the performance of hybrid and pure P2P systems. Reference [25] compares the performance of hybrid systems with different replication and server organization schemes. Several measurement studies over Gnutella, a pure system, include [1] and [22]. These studies conclude that an effective system must 1) prevent "freeloading", where some nodes take from the community without contributing, and 2) distribute work to peers according to their capabilities. In regards to the first point, systems such as MojoNation [15] and ongoing research (e.g., [3, 9]) seek to develop incentives for users to contribute. In regards to the second point, reference [14] proposes a pure system in which nodes can direct the flow of messages away from themselves, if they are overloaded, and towards higher-capacity nodes. Super-peer networks also address this point.

Much research has also been focused on improving search efficiency by designing good search protocols; for example, Chord [23], Pastry [20], CAN [19], and Tapestry [27] in the specific context of supporting point queries, and [4, 26] in the context of supporting more expressive queries (e.g., keyword query with regular expressions). Each of these search protocols can be applied to super-peer networks, as the use of super-peers and the choice of routing protocol are orthogonal issues.

3 Problem Description

To describe how a super-peer network functions, we will first give background on pure P2P networks, and then describe what changes when peers in the pure system are replaced by super-peers and clients.

3.1 Pure peer-to-peer networks

In a P2P system, users submit queries and receive results (such as actual data, or pointers to data) in return. Data shared in a P2P system can be of any type; in most cases users share files. Queries can also take any appropriate form given the type of data shared. For example, in a file-sharing system, queries might be unique identifiers, or keywords with regular expressions. Each node has a collection of files or data to share.

Two nodes that maintain an open connection, or *edge*, between themselves are called *neighbors*. The number of neighbors a node has is called its *outdegree*. Messages are routed along these open connections only. If a message needs to travel between two nodes that are not neighbors, it will travel over multiple edges. The length of the path traveled by a message is known as its *hop* count.

When a user submits a query, her node becomes the query *source*. In the baseline search technique used by Gnutella, the source node will send the query to all of its neighbors. Other routing protocols such as those described in [4, 26] may send the query to a select subset of neighbors, for efficiency. When a node receives a query, it will process it over its local collection. If any results are found, it will send a single Response message back to the source. The total result set for a query is the bag union of results from every node that processes the query. The node may also forward the query to its neighbors. In the baseline Gnutella search, query messages are given a *time to live* (TTL) that specifies how many hops the message may take. When a node receives a query, it decrements the TTL, and if the TTL is greater than 0, it forwards the query to all its neighbors. The number of nodes that process the query is known as the *reach* of the query.

In some systems such as Gnutella, the location of the source is not known to the responding node. In this case, the Response message will be forwarded back along the reverse path of the query message, which ultimately leads back to the source. In the case where the source location is known, the responder can open a temporary connection to the source and transfer results directly. While the first method uses more aggregate bandwidth than the second, it will not bombard the source with connection requests, as will the second method, and it provides additional benefits such as anonymity for the query source. Hence, in this paper, we assume the first method is used.

3.2 Super-peer networks

A super-peer network operates exactly like a pure P2P network, except that every "node" in the previous description is actually a super-peer, and each super-peer is connected to a set of clients. Clients are connected to a single super-peer only. Figure 1a illustrates what the topology of a super-peer network might look like. We call a super-peer and its

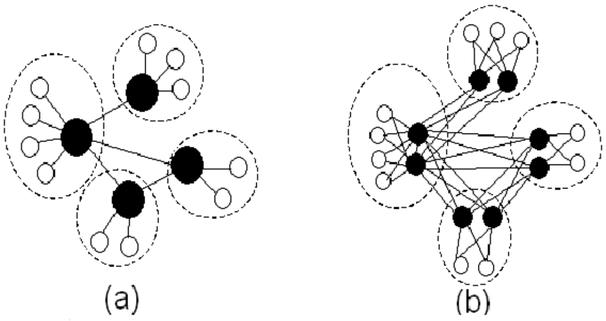

Figure 1: Illustration of a super-peer network (a) with no redundancy, (b) with 2-redundancy. Black nodes represent super-peers, white nodes represent clients. Clusters are marked by the dashed lines.

clients a *cluster*, where *cluster size* is the number of nodes in the cluster, including the super-peer itself. A pure P2P network is actually a "degenerate" super-peer network where cluster size is 1 – every node is a super-peer with no clients.

When a super-peer receives a query from a neighbor, it will process the query on its clients' behalf, rather than forwarding the query to its clients. In order to process the query for its clients, a super-peer keeps an index over its clients' data. This index must hold sufficient information to answer all queries. For example, if the shared data are files and queries are keyword searches over the file title, then the super-peer may keep inverted lists over the titles of files owned by its clients. If the super-peer finds any results, it will return one Response message. This Response message contains the results, and the address of each client whose collection produced a result.

In order for the super-peer to maintain this index, when a client joins the system, it will send metadata over its collection to its super-peer, and the super-peer will add this metadata to its index. When a client leaves, its super-peer will remove its metadata from the index. If a client ever updates its data (e.g., insertion, deletion or modification of an item), it will send this update to the super-peer as well. Hence, super-peer networks introduce two basic actions in addition to query: *joins* (with an associated *leave*), and *updates*.

When a client wishes to submit a query to the network, it will send the query to its super-peer only. The super-peer will then submit the query to its neighbors as if it were its own query, and forward any Response messages it receives back to the client. Outside of the cluster, a client's query is indistinguishable from a super-peer's query.

Since clients are shielded from all query processing and traffic, weak peers can be made into clients, while the core of the system can run efficiently on a network of powerful super-peers. Hence, as mentioned earlier, super-peer networks use the heterogeneity of peers to their advantage. Also, as we will see in Section 5, the overhead of maintaining an index at the super-peer is small in comparison to the savings in query cost this centralized index allows.

Super-peer redundancy. Although clusters are efficient, a super-peer becomes a single point of failure for its cluster, and a potential bottleneck. When the super-peer fails or simply leaves, all its clients become temporarily disconnected until they can find a new super-peer to connect to.

To provide reliability to the cluster and decrease the load on the super-peer, we introduce redundancy into the design of the super-peer. We say that a super-peer is *k-redundant* if there are k nodes sharing the super-peer load, forming a single "virtual" super-peer. Every node in the virtual super-peer is a *partner* with equal responsibilities: each partner is connected to every client and has a full index of the clients' data, as well as the data of other partners. Clients send queries to each partner in a round-robin fashion[1]; similarly, incoming queries from neighbors are distributed across partners equally. Hence, the incoming query rate on each partner is a factor of k less than on a single super-peer with no redundancy, though the cost of processing each query is higher due to the larger index.

A k-redundant super-peer has much greater availability and reliability than a single super-peer. Since all partners can respond to queries, if one partner fails, the others may continue to service clients and neighbors until a new partner can be found. The probability that all partners will fail before any failed partner can be replaced is much lower than the probability of a single super-peer failing.

However, super-peer redundancy comes at a cost. In order for each partner to have a full index with which to answer queries, a client must send metadata to each of these partners when it joins. Hence, the aggregate cost of a client join action is k times greater than before. Also, neighbors must be connected to each one of the partners, so that any partner may receive messages from any neighbor. Assuming that every super-peer in the network is k-redundant, the number of open connections amongst super-peers increases by a factor of k^2. Because the number of open connections increases so quickly as k increases, in this paper we will only consider the case where $k = 2$. Henceforth, we will use the term "super-peer redundancy" to refer to the 2-redundant case only. Figure 1b illustrates a super-peer network topology with redundancy.

At first glance, super-peer redundancy seems to trade off reliability for cost. Cost-wise (disregarding the loss in reliability), a more effective policy might be to simply make each partner into a super-peer with half the clients – that is, have twice the number of clusters at the half the original size and no redundancy. In this way, the individual query load on each super-peer will be halved as with 2-redundancy, and the index will be half the size. However, in Section 5.1, we will see how super-peer redundancy actually has the surprising effect of *reducing* load on each super-peer, in addition to providing greater reliability.

Topology of the super-peer network. Gnutella is the only open P2P system for which topology information is known. In Gnutella, the overlay network formed by the peers follows a power-law, meaning the frequency f_d of an outdegree d is proportional to d^α, where α is some constant. The power-law naturally occurs because altruistic and powerful peers voluntarily accept a greater number of neighbors. (We will see in Section 5.1 how a greater outdegree results in greater load).

From crawls of the Gnutella network performed in June 2001, we found the average outdegree of the network to be 3.1. In a super-peer network, however, we believe the average outdegree will be much higher, since super-peers have greater load capacity than an average peer. Because it is difficult to predict the average outdegree of a super-peer

[1] Other load-balancing techniques can be used; we choose round-robin for minimum overhead.

Name	Default	Description
Graph Type	Power	The type of network, which may be strongly connected or power-law
Graph Size	10000	The number of peers in the network
Cluster Size	10	The number of nodes per cluster
Redundancy	No	A boolean value specifying whether or not super-peer redundancy is used
Avg. Outdegree	3.1	The average outdegree of a super-peer
TTL	7	The time-to-live of a query message
Query Rate	$9.26 \cdot 10^{-3}$	The expected number of queries per user per second
Update Rate	$1.85 \cdot 10^{-2}$	The expected number of updates per user per second

Table 1: Configuration parameters, and default values

network, we will assume that every super-peer will be given a "suggested" outdegree from some global source (e.g., as part of the protocol). We assume the actual outdegrees will vary according to a power-law with this "suggested" outdegree as the average, since some super-peers will be more able and willing to accept a large outdegree than others.

4 Evaluation Model

We will compare the performance of super-peer networks in a file-sharing application based on two types of metrics: *load*, and *quality of results*.

Load is defined as the amount of work an entity must do per unit of time. Load is measured along three resource types: *incoming bandwidth*, *outgoing bandwidth*, and *processing power*. Bandwidth is measured in bits per second (bps), processing power in cycles per second (Hz). Because load varies over time, we will be using mean-value analysis, described in further detail in the next section. We treat incoming and outgoing bandwidth as separate resources because their availability is often asymmetric: many types of connections (e.g., cable modem) allow greater downstream bandwidth than upstream. As a result, upstream bandwidth may become a bottleneck even if downstream bandwidth is abundant.

Some systems are efficient overall, while other systems may be less efficient, but put a lower load on individual super-peers. Hence, we will look at both *individual* load, the load of a single node, as well as *aggregate* load, the sum of the loads of all nodes in the system.

We measure quality of results by the *number of results* returned per query. Other metrics for quality of results often includes relevance of results and response time. While our performance model does not capture absolute response time, relative response times can be deduced through our results, seen in Section 5.1. Because relevance of results is subjective and application specific, we do not use this metric.

4.1 Performance Evaluation

We will be comparing the performance of different *configurations* of systems, where a configuration is defined by a set of parameters, listed in Table 1. Configuration parameters describe both the topology of the network, as well as user behavior. We will describe these parameters in further detail as they appear later in the section.

Action	Bandwidth Cost (Bytes)	Processing Cost (Units)
Send Query	82 + query length	$.44 + .003 \cdot$ query length
Recv. Query	82 + query length	$.57 + .004 \cdot$ query length
Process Query	0	$14 + 1.1 \cdot$ # results
Send Response	$80 + 28 \cdot$ # addr $+76 \cdot$ #results	$.21 + .31 \cdot$ # addr $+.2 \cdot$ #results
Recv Response	$80 + 28 \cdot$ # addr $+76 \cdot$ #results	$.26 + .41 \cdot$ # addr $+.3 \cdot$ #results
Send Join	$80 + 72 \cdot$ # files	$.44 + .2 \cdot$ # files
Recv. Join	$80 + 72 \cdot$ # files	$.56 + .3 \cdot$ # files
Process Join	0	$14 + 10.5 \cdot$ # files
Send Update	152	.6
Recv. Update	152	.8
Process Update	0	30
Packet Multiplex	0	$.01\cdot$ # open connections

Figure 2: Costs of atomic actions

Description	Value
Expected length of query string	12 B
Average size of result record	76 B
Average size of metadata for a single file	72 B
Average number of queries per user per second	$9.26 \cdot 10^{-3}$

Figure 3: General Statistics

There are 4 steps in the analysis of a configuration:

Step 1: Generating an instance. The configuration parameters listed in Table 1 describe the desired topology of the network. First, we calculate the number of clusters as $n = \frac{\mathit{GraphSize}}{\mathit{ClusterSize}}$. We then generate a topology of n nodes based on the *type* of graph specified. We consider two types of networks: *strongly connected*, and *power-law*. We study strongly connected networks as a best-case scenario for the number of results (reach covers every node, so all possible results will be returned), and for bandwidth efficiency (no Response messages will be forwarded, so bandwidth is conserved). We study power-law networks because they reflect the real topology of the Gnutella network. Strongly connected networks are straightforward to generate. Power-law networks are generated according to the *PLOD* algorithm presented in [18].

Each node in the generated graph corresponds to a single cluster. We transform each node into a single super-peer or "virtual" super-peer if there is redundancy. We then add C clients to each super-peer, where C follows the normal distribution $N(\mu_c, .2\mu_c)$, and where μ_c is the mean cluster size defined as $\mu_c = \mathit{ClusterSize} - 1$ if there is no redundancy and $\mu_c = \mathit{ClusterSize} - 2$ if there is. To each peer in the network, both super-peers and clients, we assign a number of files and a lifespan according to the distribution of files and lifespans measured by [22] over Gnutella.

Note that the actual distribution of cluster size depends on how clients discover super-peers. The "bootstrapping" problem of how a joining node discovers currently connected nodes is an open problem in pure and super-peer networks, with many potential solutions. For example, in Gnutella, many peers use "pong servers"[2] that keep a list of currently connected nodes in the network, and give the joining peer the IP address of a randomly chosen connected node. Such a service might instead provide the peer with the IP address of a randomly chosen super-peer, in a super-peer network. Regardless of the exact method used, which is orthogonal to this study, we expect any well-constructed discovery method to be fair, or at least random. Hence, we use the normal distribution to describe the distribution of cluster sizes.

[2]Two popular pong servers are gnutellahosts.com, originally run by Clip2 [24], or router.limewire.com, run by LimeWire [13]

Step 2: Calculating expected cost of actions. There are three "macro" actions in our cost model: query, join and update. Each of these actions is composed of smaller "atomic" actions for which costs are given in Table 2. There are two types of cost measured: bandwidth, and processing power. In terms of bandwidth, the cost of an action is the number of bytes being transferred. We define the size of a message by the Gnutella protocol where applicable. For example, query messages in Gnutella include a 22-byte Gnutella header, a 2 byte field for flags, and a null-terminated query string. Total size of a query message, including Ethernet and TCP/IP headers, is therefore 82 + query string length. Some values, such as the size of a metadata record, are not specified by the protocol, but are a function of the type of data being shared. These values, listed in Table 3, were gathered through observation of the Gnutella network over a 1-month period, described in [26].

The processing costs of actions are given in coarse units, and were determined by measurements taken on a Pentium III 930 MHz processor running Linux kernel version 2.2. Processing costs will vary between machines and implementations, so the values seen in Table 2 are meant to be representative, rather than exact. A unit is defined to be the cost of sending and receiving a Gnutella message with no payload, which was measured to be roughly 7200 cycles on the measurement machine. When displaying figures in Section 5, we will convert these coarse units to cycles using this conversion ratio.

The packet multiplex cost is a per-message cost reflecting the growing operating system overhead of handling incoming and outgoing packets as the number of open connections increases. Please see Appendix A for a discussion of this cost and its derivation.

As an example of how to calculate the cost of a "macro" action, consider the cost of a client joining the system. From the client perspective, the action consists of the startup cost of sending a Join message, and for each file in its collection, the cost of sending the metadata for that file to the super-peer. Suppose the client has x files and m open connections. Outgoing bandwidth for the client is therefore $80 + 72 \cdot x$, incoming bandwidth is 0, and processing cost is $.44 + .2 \cdot x + .01 \cdot m$. From the perspective of the super-peer, the client join action consists of the startup cost of receiving and processing the Join message, and then for each file owned by the client, the cost of receiving the metadata and adding the metadata to its index. In this example, we see how an action can involve multiple nodes, and how cost is dependent on the instance of the system (e.g., how many files the client owns).

Updates and leaves, like joins, are a straightforward interaction between a client and super-peer, or just the super-peer itself. Queries, however, are much more complicated since they involve a large number of nodes, and depend heavily on the topology of the network instance. Here, we describe how we count the actions taken for a query. For our evaluations, we assume the use of the simple baseline search used in Gnutella (described in Section 3). However, other protocols such as those described in [26] may also be used on a super-peer network, resulting in overall performance gain, but similar tradeoffs between configurations.

We use a breadth-first traversal over the network to determine which nodes receive the query, where the source of the traversal is the query source S, and the depth is equal to the TTL of the query message. Any response message will then travel along the reverse path of the query, meaning it will travel up the predecessor graph of the breadth-first traversal until it reaches the source S. Every node along this path must sustain the cost of forwarding this message. Note that messages in a real network may not always propagate in an exact breadth-first fashion, as latencies will vary across connections. However, since we are focusing on cost rather than response time, and since query messages are very small (average 94 bytes), breadth-first traversal remains a reasonable approximation of query propagation for our purposes.

To determine how many results a super-peer T returns, we use the query model developed in [25]. Though this query model was developed for hybrid file-sharing systems, it is still applicable to the super-peer file-sharing systems we are studying. Given the number of files in the super-peer's index, which is dependent on the particular instance I generated in step 1, we can use this model to determine $E[N_T|I]$, the expected number of results returned, and $E[K_T|I]$, the expected number of T's clients whose collections produced results. Note that since the cost of the query is a linear function of $(N_T|I)$ and $(K_T|I)$, and load is a linear function of the cost of queries, we can use these expected values to calculate expected load. Please see Appendix B for how we calculate $E[N_T|I]$ and $E[K_T|I]$ using the query model.

Step 3: Calculating load from actions. In step 2 we calculate expected values for C_{qST}, C_{jST}, and C_{uST}, the cost of a query, join and update, respectively, when the action is initiated by node S and incurred on node T, for every pair of nodes S and T in the network instance. S and T may be super-peers or clients.

For each type of action, we need to know the rate at which the action occurs. Default values for these rates are provided in Table 1. Query rate is taken from the general statistics listed in Table 3. Join rate is determined on a per-node basis. On average, if the size of the network is stable, when a node leaves the network, another node is joining elsewhere. Hence, the rate at which nodes join the system is the inverse of the length of time they remain logged in. Update rate is obtained indirectly, since it is impossible to observe through experiments how frequently users updated their collections. We first assume that most online updates occur as a result of a user downloading a file. We then use the rate of downloads observed in [25] for the OpenNap system as our update rate. Because the cost of updates is low relative to the cost of queries and joins, the overall performance of the system is not sensitive to the value of the update rate.

Given the cost and rate of each type of action, we can now calculate the expected load on an individual node T, for a network instance I:

$$E[M_T|I] = \sum_{S \in \text{network}} E[C_{qST}|I] \cdot E[F_{qS}] + E[C_{jST}|I] \cdot E[F_{jS}|I] + E[C_{uST}|I] \cdot E[F_{uS}] \quad (1)$$

F_{qS} is defined as the number of queries submitted by S in a second, so $E[F_{qS}]$ is simply the query rate per user listed in Table 1, for all S. $F_{jS}|I$ and F_{uS} are similarly defined.

We can also calculate the expected number of results per query originated by node S:

$$E[R_S|I] = \sum_{T \in \text{network}} E[N_T|I] \quad (2)$$

Often we will want to calculate the expected load or results per query on nodes that belong to some set Q defined by a condition: for example, Q may be the set of all nodes that are super-peers, or the set of all super-peers with 2 neighbors. The expected load M_Q of all such nodes is defined as:

$$E[M_Q|I] = \frac{\sum_{n \in Q} E[M_n|I]}{|Q|} \tag{3}$$

Finally, aggregate load is defined as:

$$E[\overline{M}|I] = \sum_{n \in \text{network}} E[M_n|I] \tag{4}$$

Step 4: Repeated Trials. We run analysis over several instances of a configuration and average $E[M|I]$ over these trials to calculate $E[E[M|I]] = E[M]$, the value by which we compare different configurations. We also calculate 95% confidence intervals for $E[M|I]$.

5 Results

In this section we present the results of our evaluations on a wide range of configurations. Because there are many different scenarios and factors to consider, we do not attempt to report on all the results here. Instead, from all our results we distill a few important "rules of thumb" to follow while designing a P2P topology, and present these rules to the reader, supported with examples from our experiments. We then formulate a general procedure that incorporates the rules and produces an efficient topology. Finally, we discuss how an individual node without a global view of the system might make local decisions to form a globally efficient network.

Recall that all results in the following section are expected values. All figures show the expected value of costs, along with vertical bars to denote 95% confidence intervals for $E[\text{value}|instance]$ where appropriate. All figures use the default parameters listed in Table 1, unless otherwise specified. Please refer to Appendix C for additional results.

5.1 Rules of Thumb

The four rules of thumb we gathered from our experiments are as follows:

1. Increasing cluster size decreases aggregate load, but increases individual load.
2. Super-peer redundancy is good.
3. Maximize outdegree of super-peers.
4. Minimize TTL.

Let us now examine the causes, implications and details of each.

#1 Increasing cluster size decreases aggregate load, but increases individual load. In terms of cluster size, there is a clear tradeoff between aggregate and individual load. Figure 4 shows the aggregate bandwidth required by

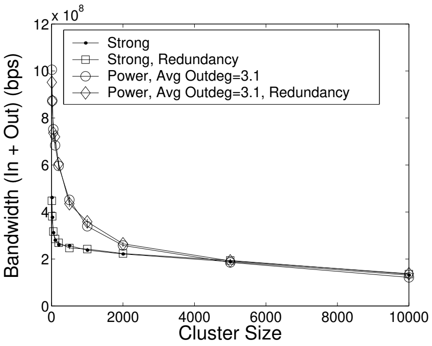

Figure 4: Aggregate load decreases with cluster size

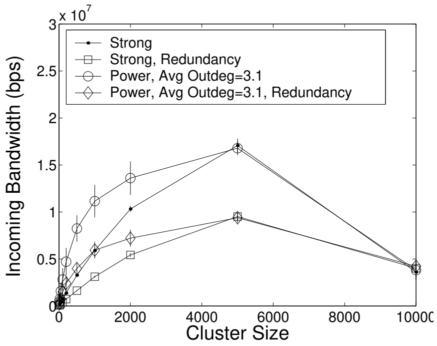

Figure 5: Individual load increases with cluster size, with few exceptions

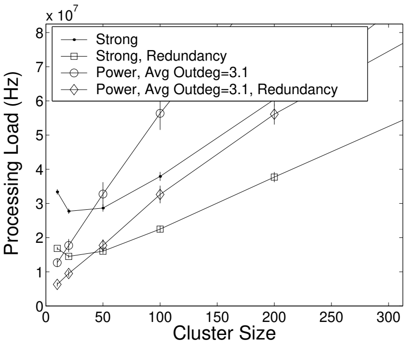

Figure 6: Individual processing load

two systems as cluster size is varied. One system has a strongly connected topology with TTL=1, shown as a best-case scenario for bandwidth. The other system has a power-law topology with average outdegree of 3.1 and TTL=7, parameters reflective of the Gnutella topology. For now, ignore the curves for super-peer redundancy. In both systems, the expected number of results is the same for all cluster sizes.

Both curves in Figure 4 show that aggregate load decreases dramatically at first as cluster size increases. Aggregate load then experiences a “knee” around cluster size of 200 in the strong network and 1000 in the power-law network, after which it decreases gradually. Intuitively, the fewer the super-peers in a system (i.e., the larger the cluster size), the less communication overhead between super-peers there will be. At the extreme where there is a single super-peer (e.g., cluster size is 10000 in Figure 4), queries are sent directly from clients to a single “server”, and results are sent directly from server to client. At the other extreme (e.g., cluster size is 1), the same number of results are being returned as in the first case, but in addition, there is the cost of sending queries to every super-peer, a startup cost for every super-peer as they process the query, and the overhead of additional packet headers for individual query responses. In fact, it is these extra costs, which grow inversely proportionally to the number of super-peers in the system, that cause the knee to occur in Figure 4.

Though large cluster size is ideal for aggregate load, it has the opposite effect on individual super-peer load, with a few exceptions. Figure 5 shows the individual incoming bandwidth required of super-peers for the same two systems as before, as cluster size is varied. With one exception, we see that individual load grows rapidly with the growth of cluster size. For example, in the strongly connected system, a super-peer with 100 clients has almost twice the load as a super-peer with 50.

The exception we see in this figure, occurring when there is a single super-peer (cluster size is 10000), only occurs for incoming bandwidth, and not for outgoing bandwidth or processing load. The explanation for this exception is as follows: Most incoming bandwidth load for a super-peer arises from forwarding query results from other super-peers to a client. If a cluster consists of a fraction f of all nodes in the network (i.e., $f = \frac{\text{cluster size}}{\text{number users}}$ in our example), then the super-peer for that cluster will be submitting roughly f of all queries on behalf of its clients. Because files in its own index constitute roughly f of all responses to a query, $1 - f$ of all results will be sent from other clusters to this super-peer. Therefore, total expected incoming results is proportional to $f \cdot (1 - f)$, which has a maximum at $f = \frac{1}{2}$ (i.e., at a cluster size of 5000 in Figure 5), and two minimums at $f = 0$ and $f = 1$. Incoming bandwidth of a super-peer when cluster size is 10000 is thus much lower than when cluster size is 5000. This exception is important to keep in mind if incoming (downstream) bandwidth is the bottleneck, and we are considering large cluster sizes. Thus, a system designer should be careful to avoid cluster sizes around a tenth to a half of the total number of peers (i.e., where load exceeds load at *ClusterSize* = *GraphSize*).

Another exception to the rule can be found looking at processing load when outdegree is large. Figure 6 shows individual processing load as a function of cluster size, for the same two systems as before. Note that the processing load of a super-peer in the strongly connected graph actually increases as cluster size becomes very small. The reason for this behavior is the overhead of handling network traffic when the number of connections is large (see Appendix A for a discussion of this overhead). In a strongly connected graph, the number of connections is equal to the cluster size plus $\frac{\text{number users}}{\text{cluster size}}$, which reaches a maximum at very large and very small cluster sizes. The overhead of many open connections is high when cluster size is large, but relative to the cost of other actions (such as query processing), it is small. On the other hand, when cluster size is small, the majority of the processing load is due to this overhead. In most operating systems, the default number of open connections is limited, meaning overhead issues aside, having 1000+ open connections is not feasible. However, in the cases where it is feasible, this network overhead is an important consideration when selecting small cluster sizes. We will see how the number of open connections becomes a design consideration in Section 5.2.

What is a good tradeoff between aggregate and individual load? The answer to this question is specific to the application and situation. In general, individual good is probably more important than overall good, since users are less likely to altruistically be a super-peer if cost is very high. However, if all users belong to a single organization – for example, they are running on a corporate LAN – then aggregate load becomes quite important, as all costs are ultimately incurred on a single entity. In addition, a very important factor to consider is the availability and reliability of the system. With large cluster sizes and few super-peers, there are a few points of failure, which has a worse effect on availability should these super-peers leave, fail or be attacked. By keeping cluster sizes small, there are no easy targets for malicious users, and failure of a super-peer leaves just a few clients temporarily unconnected. In general, then, it would be good to choose a cluster size that is small enough to keep a reasonable individual load and provide reliability to the system, but large enough to avoid the knee in aggregate load when cluster size is small.

#2 Super-peer redundancy is good. Going back to Figure 4 and looking at the curves for super-peer redundancy, we see that introducing redundancy has no significant affect on aggregate bandwidth for both systems. In addition, Figure 5 shows that redundancy does decrease individual super-peer load significantly. For example, when cluster size is 100 in the strongly connected system, super-peer redundancy increases aggregate load by about 2.5%, but decreases individual load in each partner by 48% – driving it down to the individual load of a non-redundant super-peer when

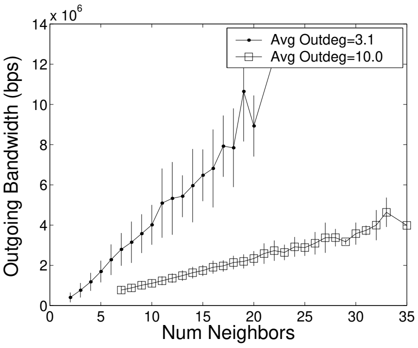

Figure 7: When all super-peers increase outdegree, individual and aggregate load decreases

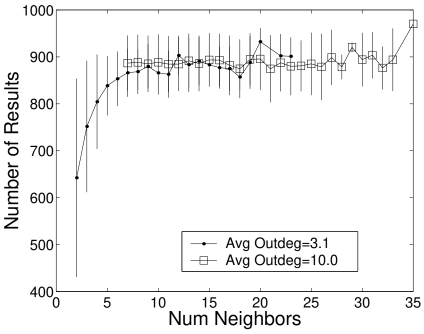

Figure 8: With a low average outdegree, many super-peers do not receive the full number of results

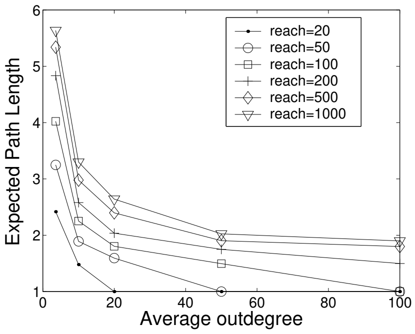

Figure 9: Estimating expected path length given reach and outdegree

cluster size 40. This effect is surprising, given the overhead of maintaining larger indices. The savings in cost come mostly from the decrease in total per-message overhead of sending results. Therefore, in terms of bandwidth, super-peer redundancy gives us better performance than both a system with no redundancy and the same cluster size, and a system with no redundancy and half the cluster size. Super-peer redundancy gives us the "best of both worlds" – the good aggregate load of a large cluster size, and the good individual load of a smaller cluster size – in addition to increased reliability.

However, super-peer redundancy involves a tradeoff between individual and aggregate load in terms of processing power. In the strongly connected system discussed earlier where cluster size is 100, aggregate processing load increases by roughly 17% when super-peer redundancy is introduced, while individual processing load decreases by about 41%. (Note that since the number of super-peers/partners doubles, aggregate load can increase though individual load decreases). Therefore, in a system such as Gnutella, super-peer redundancy is definitely recommended when individual load is the bottleneck, but should be applied with care if aggregate processing load is a concern. Please see Appendix C for a discussion on super-peer redundancy in different configurations.

#3 Maximize outdegree of super-peers. In the current Gnutella network, weak peers with few resources will connect to few neighbors, while altruistic powerful peers connect to many neighbors. This arrangement seems reasonable, since weak peers should not take on more load than they can handle. However, our experiments clearly show that increasing outdegree can actually *reduce* individual load, provided all peers do so. Figure 7 shows a histogram of individual outgoing bandwidth loads for super-peers as a function of outdegree for two systems with power-law topologies: one where average outdegree is 3.1 (reflective of the Gnutella topology), and one where average outdegree is 10. Both systems have a cluster size of 20, though these results are the same for any cluster size. Figure 8 is like Figure 7, except it shows a histogram of expected number of results. As these figures present histograms, vertical bars denote one standard deviation, rather than confidence intervals.

First, note that while super-peers with very few (2 or 3) neighbors in the first topology (average outdegree of 3.1) do have a slightly lighter load than any super-peers in the second topology (average outdegree of 10), they also receive fewer results. For example, in Figure 7, when average outdegree equals 3.1, the expected load of a super-peer with 3 neighbors is roughly $7.6 \cdot 10^5$ bps, while Figure 8 shows that this node receives roughly 750 expected results per query. In contrast, when average outdegree equals 10, the expected load of a super-peer with 7 neighbors is slightly higher, at $7.7 \cdot 10^5$ bps, but the expected number of results it receives is much higher, at roughly 890. Hence, the "gain" in load comes with a cost in user satisfaction. Once a super-peer in the first topology has enough neighbors (7) to attain full results, it has a load that is higher than that of most peers in the second topology! Super-peers in a system with higher outdegree therefore have a much lower load for a given number of results.

We can also see that super-peers with the highest outdegree in the first topology must support an extremely heavy load, while the loads of all peers in the second topology remain in the same moderate range. In this respect, the second topology can be said to be more "fair". Furthermore, fewer concentrated points of heavy load in the first topology translate to fewer points of failure, and therefore less reliability.

In terms of aggregate load, our results (shown in Appendix D) show an improvement of over 31% in bandwidth and a slightly lower processing load is attained by increasing outdegree from 3.1 to 10.

Increasing average outdegree improves performance because it reduces the *expected path length* of query responses. The expected path length (EPL) is simply the expected number of hops taken by a query response message on its path back to the query source. Each hop taken consumes incoming bandwidth from the receiver, outgoing bandwidth from the sender, and processing power from both; hence, the shorter the EPL, the more efficient a query is. Furthermore, since each hop takes time, EPL is also a rough measure of the *average response time* of a query – a shorter EPL implies lower response times. For the systems shown in Figure 7, increasing average outdegree from 3.1 to 10 decreases EPL from 5.4 to 3.

For these benefits to apply, however, *all* super-peers must agree to increase their outdegree. Consider the system where average outdegree is 3.1 in Figure 7. If a single peer with 4 neighbors decides to increase its outdegree to 9, its load will increase from $1.18 \cdot 10^6$ to $3.58 \cdot 10^6$ bps, a 303% increase. If all super-peers increased their outdegree such that average outdegree is 10, then that super-peer will actually experience a 14% reduction in load, to $1.01 \cdot 10^6$ bps.

In summary, if every super-peer agrees to increase outdegree, every super-peer's load will either decrease dramatically, or increase slightly accompanied by an increase in number of results. If only a few super-peers increase their outdegree, however, those few nodes will suffer a tremendous increase in load. It is important, therefore, that increasing outdegree be a uniform decision.

As a caveat to this rule in extreme cases, it is possible for the outdegree to be *too* large, in which case performance is negatively affected. Please refer to Appendix E for a description of this caveat.

#4 Minimize TTL. For a given outdegree and topology, reach is a function of TTL. Since the expected number of results is proportional to expected reach, a designer may choose the *desired reach* based on the desired number of results, and then select the TTL that will produce this reach. We will discuss how to select the TTL later in the section.

(1)	Select the desired reach r.
(2)	Set TTL=1
(3)	Decrease cluster size until desired individual load is attained. – if bandwidth load cannot be attained, decrease r, as no configuration can be more bandwidth-efficient than TTL=1. – if individual load is too high, apply super-peer redundancy, and/or decrease r.
(4)	If average outdegree is too high, increment TTL by 1, and go back to step (3)
(5)	Decrease average outdegree if doing so does not affect EPL, and reach r can still be attained.

Figure 10: Global Design Procedure

However, if the desired reach covers every super-peer in the network, then an infinite number of TTLs produce the correct reach, since if TTL=x allows all nodes to be reached, TTL=$x+1$ will have the same reach. In such a case, it is important to determine the minimum TTL needed to reach all nodes, and use that TTL. For example, a system with average outdegree=20 and TTL=4 uses roughly $7.75 \cdot 10^8$ bps in aggregate incoming bandwidth for a full reach. However, that same system uses only $6.30 \cdot 10^8$ bps, or 19% less aggregate incoming bandwidth, if TTL is reduced to 3 (which still allows a full reach). Similar improvements are seen for outgoing bandwidth and processing load.

The cause of this difference is the cost of sending unnecessary query messages when TTL > 3. Once queries have reached every node, any additional query message will be *redundant* – that is, it will have traveled over a cycle in the network, and arrive at a node that has already seen the query before. Any redundant query will be dropped by the receiver. Although query messages are not large, if every node sends a redundant query to each of its neighbors, the resources consumed can add up to a significant amount.

When reach covers every node in the system, the simplest way to choose the best TTL is for individual super-peers to monitor the response messages it receives. If a super-peer rarely or never receives responses from beyond x hops away, then it can decrease its TTL to x without affecting its reach. When the desired reach covers just a subset of all nodes, however, finding a TTL to produce the desired reach is more difficult. Such a decision should be made globally, and not locally, since otherwise super-peers have an incentive to set the TTL of their queries unnecessarily high: to get more results at the expense of other nodes' loads.

The correct global TTL can be obtained by predicting the EPL for the desired reach and average outdegree, and then rounding up. Figure 9 shows the experimentally-determined EPL for a number of scenarios. Along the x-axis we vary the average outdegree of the topology (assumed to be power-law), and the different curves show different desired reaches. For example, when average outdegree is 20 and the desired reach is 500 nodes, Figure 9 tells us the EPL is roughly 2.5. Hence, TTL should be set to 3. Please refer to Appendix F for further details in predicting a global TTL.

5.2 Procedure for Global Design

Putting all these rules together, we now present a general procedure for the global design of a P2P topology, shown in Figure 10. The global design procedure is intended for a system administrator or designer, and its goal is to suggest an efficient system configuration given constraints and properties specified by the designer. These constraints include the maximum load and open connections allowed on an individual super-peer, and the maximum aggregate load on the system, if desired. The properties include the number of users in the network and the desired reach. (As mentioned before, the desired reach can be chosen according to the desired number of results per query, since the two are proportional.) Although we can not prove the procedure yields the "best" topology, empirical evidence from analysis shows it usually returns a topology for which improvements can not be made without violating the given constraints.

The design procedure may be applied during *initial* design time, and it can be applied *incrementally* or *adaptively* as the system is running. For initial design, the desired properties and constraints of the system, based on user behavior and peer characteristics, must be known at design time. Such information may be difficult to obtain accurately before the system runs. Hence, in addition to the initial design phase, a centralized decision-maker can run with the system and periodically re-evaluate the design procedure based on observation and feedback from the super-peers. For example, users joining the Morpheus [16] P2P network must first contact a centralized server which then directs them to a super-peer. Though little is known about what this server does in the proprietary Morpheus system, it is feasible that this server, which is clearly in contact with the super-peers and has access to global information, can tune the topology based on the current workload, and send directives to the super-peers to achieve this topology. The next section discusses local decision making in the absence of design-time information and a centralized decision-maker.

Let us illustrate how the procedure works by using it to refine the topology used by today's Gnutella system. As mentioned in Section 4, analysis of network crawls show that the average outdegree for Gnutella is 3.1. At the time of this writing, the current Gnutella network size has been reported to vary between 10000 to 30000 users [13], so we choose 20000 to be size of the network in our analysis. Cluster size is 1, since there are few or no super-peers, and analysis shows that the reach of such a topology with TTL=7 is roughly 3000 nodes out of 20000 total.

First, let us specify the desired constraints and properties of our system. To compare the results of the procedure with the current Gnutella system, we set reach to be 3000 peers (equivalently, $\frac{3000}{\text{cluster size}}$ super-peers), and the size of the network to be 20000 peers. For individual loads, let us limit our expected upstream and downstream bandwidth load to 100 Kbps each way, our expected processing load to 10 MHz, and our expected number of open connections to 100. Let us also assume for now that to keep the peer program simple, we do not want super-peer redundancy. Note that in specifying load limits, it is important to choose a limit that is far below the actual capabilities of the peer, for several reasons. First, actual load may exceed expected load during bursts of activity. Second, the expected load is for *search* only, and not for download, chat, and other activities in the system. Finally, users will probably not want all their resources to be devoted to the system, but just a small fraction.

Now, let us run the procedure over our specifications. Reach has already been set to 3000 (step 1). Next, we set TTL=1, so that EPL=1 for most efficient bandwidth usage (step 2). Now we must choose a small enough cluster size such that our requirements for individual load are met (step 3). Under our constraints, analysis shows the largest supportable cluster size is roughly 20. However, to achieve the desired reach, average outdegree must be 150 (in order

	Incoming Bandwidth (bps)	Outgoing Bandwidth (bps)	Processing Power (Hz)	Number Results	EPL
Today	$9.08 \cdot 10^8$	$9.09 \cdot 10^8$	$6.88 \cdot 10^{10}$	269	6.5
New	$1.50 \cdot 10^8$	$1.90 \cdot 10^8$	$0.917 \cdot 10^{10}$	270	1.9
New w/ Red.	$1.56 \cdot 10^8$	$1.85 \cdot 10^8$	$1.01 \cdot 10^{10}$	269	1.9

Figure 11: Aggregate load comparison between today's system, and the new design

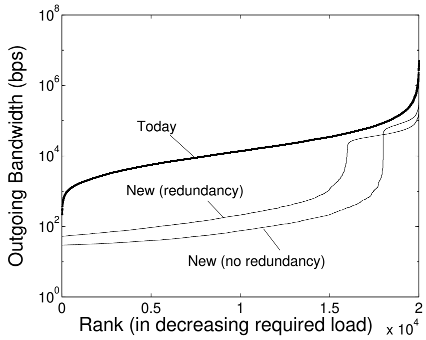

Figure 12: New design has much lower loads

to reach $\frac{3000}{20} = 150$ super-peers), leading to a total of 169 open connections (an additional 19 connections for clients) – far exceeding our limit. Following the procedure (step 4), we increase TTL to 2, and select a new cluster size of 10 using analysis, which meets all our load requirements. To achieve our desired reach, each super-peer must have about 18 neighbors (since expected reach will be bounded above by roughly $18^2 + 18 = 342$). Looking at Figure 9, we see that decreasing outdegree affects our reach, so we will not change the outdegree (step 5). Total average outdegree is therefore $9 + 19 = 28$.

How does this new topology, generated by our procedure, compare with the old one? Figure 11 lists the aggregate loads of the two topologies, along with number of results. The new topology achieves over 79% improvement over the old in all expected load aspects, while maintaining slightly higher expected quality of results. Furthermore, though we do not capture response time in our model, the average response time in the new topology is probably much better than in the old, because EPL is much shorter. Figure 11 also lists the aggregate cost of the new topology with super-peer redundancy. As we can see, aggregate cost is barely affected by the introduction of redundancy.

Individual load likewise shows dramatic improvement. Figure 12 shows the outgoing bandwidth load of every node (super-peers and clients) in the system, for all three topologies. These values were taken from a single representative instance of each topology configuration. Comparing the new topology with and without redundancy, we see that the average super-peer load (that is, the top 20% ranked nodes in the topology with redundancy, and the top 10% ranked nodes in the topology without) is 41% lower with redundancy than without. As a tradeoff, average client load is 2 to 3 times greater with redundancy. However, since client loads are already very low – on the order of 100 bps – this increase will be barely perceptible to the clients.

Comparing the new topology without redundancy to today's topology, Figure 12 shows that for the lowest 90% of loads in both system, individual load is one to two orders of magnitude lower in the new topology than in the old. This 90% corresponds to the nodes that are clients in the new topology. Among the highest 10% of loads, we still see significant improvement in the new topology, ranging from 40% improvement at the "neck" occurring at about the 90th percentile, to a whole order of magnitude for the top .1% heaviest loads. Individual users will therefore have a much better user experience with these reduced loads – including those who agree to become super-peers, especially if super-peer redundancy is used.

5.3 Local Decisions

In the case where constraints and properties of the system can not be accurately specified at design time, and in the case where a centralized decision maker is not desirable or simply not available at the moment, super-peers should be able to make local decisions that will tend towards a globally efficient topology. Here, we will qualitatively discuss rules for local decision making, based on the global rules from Section 5.1.

In the following discussion, we assume that super-peers set a limit to the load they are willing to handle, and that they are equally willing to accept any load that does not exceed this limit (i.e., they will never refuse to increase their load unless it exceeds their predefined limit). The guidelines we will discuss are actions that will help *other* super-peers at the expense of the super-peer who follows them; hence, our "limited altruism" assumption allows us to suppose that every super-peer is willing to incur these expenses on themselves, which will ultimately result in less load for everyone. (Ongoing research in other contexts (e.g. [3, 9]) is exploring how to detect whether every peer in a P2P system is correctly playing their part, and punish the freeloaders and malicious nodes.)

I. A super-peer should always accept new clients. Given that the client must be served by some super-peer in the network, it generally does not make sense for a super-peer to refuse the client. If a super-peer finds that its cluster size is getting too large to handle (i.e., load frequently exceeds the limit), it should select a capable client from its cluster to become a partner as a redundant super-peer. Alternatively, it can select a client from its cluster to become a new super-peer, and the cluster should be split into two. In this manner, the number of super-peers can adapt for an increasing global workload. Likewise, if a cluster becomes too small (i.e., load remains far below the limit), the super-peer may try to find another small cluster, and coalesce the clusters. There is no set rule as to how small a cluster should be before the super-peer tries to coalesce, as there are always other uses for the extra bandwidth (e.g., accepting a new neighbor). Following these guidelines, according to global rule #1, clusters will tend to be as large as possible while respecting individual limits, thereby minimizing aggregate load.

II. A super-peer should increase its outdegree, as long as its cluster is not growing and it has enough resources to spare. By rule #3, increasing outdegree will decrease EPL, which is beneficial to the network as a whole. Increasing outdegree when TTL is small will increase the quality of results for the super-peer and its clients, at the cost of a proportionally higher load. However, if TTL is high (in particular, the TTL of the query messages forwarded by this super-peer), the super-peer should be cautious about accepting a new neighbor, because it may find itself forwarding a large number of messages for other nodes.

As we saw earlier, increasing outdegree is only effective if everyone participates. If a super-peer finds that it does not have enough resources to support more than a few neighbors, it should consider dropping some clients to free up resources, or "resign" to become a client itself.

III. A super-peer should decrease its TTL, as long as it does not affect its reach. For the case where the desired reach is not every super-peer in the system, a decrease in TTL should only happen after an increase in outdegree, in order to maintain the same reach. For the case where a node wants to reach all super-peers, according to rule #4, decreasing TTL may not affect reach, and should therefore always be done. A super-peer can tell if its reach has been affected by whether or not it receives fewer results.

6 Conclusion

Super-peer networks are effective because they combine the efficiency of the centralized client-server model with the autonomy, load balancing and robustness of distributed search. They also take advantage of heterogeneity of capabilities across peers. Because well-designed super-peer networks promise large performance improvements for P2P systems, it is important to base design decisions on a solid understanding of the system's behavior, rather than on the guesswork currently being used. In this paper, we address this need by providing a scientific foundation on which to build a good understanding of super-peer network performance. In particular, we consider redundancy in super-peer design and topology variations, and carefully analyze super-peer network performance. From our results, we have been able to extract a few rules of thumb that capture the essential tradeoffs, a global design procedure, and local decision-making recommendations to adaptively and automatically achieve a globally efficient system.

Acknowledgements. We would like to thank Steve Gribble's group at the University of Washington for the use of their Gnutella measurement data, and Kelly Truelove and Ziad El Kurjie of Clip2 for use of their crawl data.

References

[1] Eytan Adar and Bernardo A. Huberman. Free Riding on Gnutella. http://www.firstmonday.dk/issues/issue5_10/-adar/index.html, September 2000.

[2] G. Banga and J. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proc. of USENIX*, June 1998.

[3] B. Cooper, M. Bawa, N. Daswani, and H. Garcia-Molina. Protecting the pipe from malicious peers. Technical report, Stanford University, 2002. Available at http://dbpubs.stanford.edu/pub/2002-3.

[4] A. Crespo and H. Garcia-Molina. Routing indices for peer-to-peer systems. In *Proc. of the 28th Intl. Conf. on Distributed Computing Systems*, July 2002.

[5] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[6] Freenet website. http://freenet.sourceforge.net.

[7] Gnutella website. http://www.gnutella.com.

[8] Archived Clip2 website. http://web.archive.org/web/20010201081800/dss.clip2.com/.

[9] P. Goelle, K. Keyton-Brown, and I. Mironov. Incentives for sharing in peer-to-peer networks. In *Proc. of ACM Conference on Electronic Commerce*, October 2001.

[10] R. Gooch. I/o event handling under linux. http://www.atnf.csiro.au/ rgooch/linux/docs/io-events.html, 2001.

[11] KaZaA website. http://www.kazaa.com.

[12] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Thea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. ASPLOS*, pages 190–201, November 2000.

[13] Limewire website. http://www.limewire.com.

[14] Q. Lv, S. Ratnasamy, and S. Shenker. Can heterogeneity make gnutella scalable? In *Proc. of 1st International Workshop on Peer-to-Peer Systems*, March 2002.

[15] MojoNation website. http://www.mojonation.net.

[16] Morpheus website. http://www.musiccity.com.

[17] Napster website. http://www.napster.com.

[18] C. Palmer and J. Steffan. Generating network topologies that obey power laws. In *Proc. of GLOBECOM 2000*, November 2000.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, August 2001.

[20] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.

[21] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[22] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of the Multimedia Computing and Networking*, January 2002.

[23] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, August 2001.

[24] K. Truelove. To the bandwidth barrier and beyond. http://web.archive.org/web/20010107185800/dss.clip2.com/gnutella.html, 2000.

[25] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *Proc. of the 27th Intl. Conf. on Very Large Databases*, September 2001.

[26] B. Yang and H. Garcia-Molina. Improving efficiency of peer-to-peer search. In *Proc. of the 28th Intl. Conf. on Distributed Computing Systems*, July 2002.

[27] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001. Available at http://www.cs.berkeley.edu/ ravenben/publications/CSD-01-1141.pdf.

A The Overhead of Open Connections

As the number of open connections increases, the operating system overhead of handling messages increases as well. We model the cost of this overhead as follows.

First, we assume a super-peer is designed as an *event-driven* server, where a single thread services all connections using the `select` system call to detect events on these connections. Event-driven systems are often preferred over thread-per-connection designs because of simplicity of design, and lower context-switching and synchronization overhead, especially on single-processor machines. In many operating systems (e.g., Digital UNIX, 4.3BSD, Linux), the implementation of `select` does a linear scan over an array of file descriptors, where there is one descriptor per open connection. The cost of scanning a single file descriptor in this array has been reported to be 3μs on a Pentium 100 machine running Linux 2.1.x in [10], which translates to about .04 units in our scale.

Now, reference [2] shows that as the number of open connections increases, the cost of a `select` call also increases linearly until the number of connections exceeds roughly 1000, after which the cost levels off. This levelling off is due to the amortization of events across calls to `select`. That is, as the number of connections increases, the time spent in the `select` call increases, so that more events will be reported per call. However, the load of events in these experiments is much higher than the load experienced by a super-peer in the default configuration specified in Table 1. Hence, a super-peer would probably continue to see a linear cost growth for much larger numbers of connections. Furthermore, none of the results reported in this paper consider the case where the number of open connections exceeds 1000, so for the range of connections we consider, the linear cost growth will be present.

A linear growth of overhead, however, does not imply that there will be a single call to `select` per message. Let us assume that in the case where the number of open connections is very large (approx. 1000), TTL=1. In this case, roughly 1 out of 4 incoming/outgoing messages are incoming query messages, which take the longest to process. We will then assume that once a query has been processed, an average of 4 more events will be discovered by the next call to `select`, so that the cost of the call will be amortized over 4 messages. Hence, additional overhead per open connection per message is $\frac{.04}{4} = .01$ units.

B Calculation of $E[N_T|I]$ and $E[K_T|I]$

The query model described in [25] defines two probability functions: $g(i)$, which gives us the probability that a random submitted query is equal to query q_i, and $f(i)$, which gives us the probability that a random file will be a match (a result) for query q_i. We will use the same distributions experimentally determined for the OpenNap system in [25] for our default configuration, since OpenNap is also a file-sharing P2P system, and there is no available data from live super-peer networks.

Let x_i be the number of files owned by client i of super-peer T, and let T have c clients, and own x_T files. The total number of files indexed by T is $x_{tot} = x_T + \sum_{i=1}^{c} x_i$. The query model assumes that the probability that a

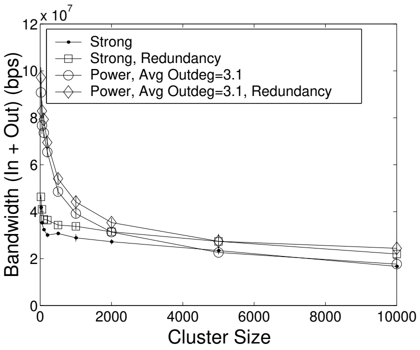

Figure A-13: Aggregate load with lower query rate

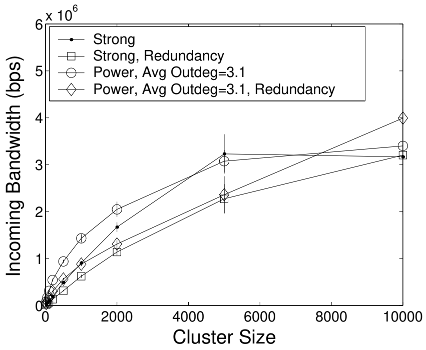

Figure A-14: Individual load of super-peers with lower query rate

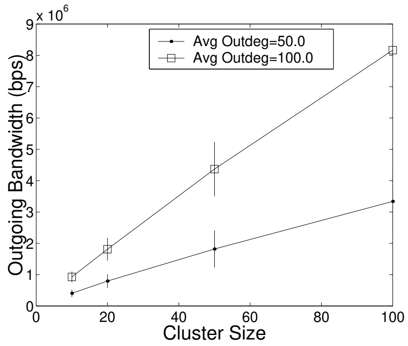

Figure A-15: A modification to Rule #3: higher outdegrees can be harmful

given file will be returned as a match is independent of whether other files in the same collection are returned. With this assumption, the number of files returned from a collection of size n for a query of selection power p is distributed according to `binomial(n,p)`, and the expected value is therefore $n \cdot p$. Hence,

$$E[N_T|I] = \sum_{j=0}^{\infty} g(j) \cdot \big(f(j) \cdot x_{tot}\big) \tag{5}$$

Now, let Q_i be the identifier variable where $Q_i = 1$ iff client i returns at least one result. $P(Q_i|I) = 1 - P(\text{client } i \text{ returns no results}) = 1 - (1-p)^{x_i}$, where p is the selection power of the current query. Hence,

$$E[K_T|I] = \sum_{i=1}^{c} E(Q_i|I) = \sum_{i=1}^{c}\sum_{j=0}^{\infty} g(j) \cdot \left(1 - \big(1 - f(j)\big)^{x_i}\right) \tag{6}$$

C System Performance with Different Action Rates

For the query and join rates observed for Gnutella, the ratio of queries to joins is roughly 10 – that is, a user will submit an average of 10 queries during each session. This ratio is important because it determines the *relative load* incurred by each type of action. Relative load of queries and joins is important because it affects the relative performance of different configurations, depending on how "join-intensive" these configurations are. For example, the cost of joins is doubled with super-peer redundancy, while the cost of queries is halved. If joins are expensive relative to queries, this tradeoff may no longer be good.

Let us consider new configurations where the relative load of queries to joins is low. In particular, we will change the query rate listed in Table 1 to be $9.26 \cdot 10^{-3}$, such that the ratio of queries to joins is now roughly 1. We could have also decreased the relative load of queries by making the cost of queries less expensive; for example, modeling a system in which all queries are for rare items and therefore receive few results. The effects of both cases are similar.

Figure A-13 shows the aggregate bandwidth of the same four systems shown in Figure 4 (Section 5.1), but with the lower query rate. We observe two effects: first, the aggregate load still decreases as cluster size decreases, but to a lesser degree than what we observed in Figure 4. Load decreases less rapidly because the savings in query

Avg Outdegree	Incoming Bandwidth (bps)	Outgoing Bandwidth (bps)	Processing Power (Hz)
3.1	$3.51 \cdot 10^8$	$3.49 \cdot 10^8$	$6.06 \cdot 10^9$
10.0	$2.67 \cdot 10^8$	$2.65 \cdot 10^8$	$6.05 \cdot 10^9$

Table 2: Aggregate load comparison between an average outdegree of 3.1, and an average outdegree of 10.0

communication overhead allowed by larger clusters is now small in comparison to the load of joins. Second, we observe that super-peer redundancy causes aggregate load to increase by a greater amount than before. For example, when cluster size is 100 in the strongly connected system, aggregate bandwidth increases by 14% with redundancy.

Figure A-14 shows the individual incoming bandwidth of super-peers for the same four systems shown in Figure 5 (Section 5.1), but with the lower query rate. First, we observe that individual load now reaches a maximum at *ClusterSize* = *GraphSize*, since load is now dominated by joins. Second, note that super-peer redundancy does not decrease load as effectively as before (though the improvements are still good). When cluster size is 100 in the strongly connected network, individual incoming bandwidth load decreases by roughly 30% with redundancy, while processing load decreases by 18%. Recall that super-peer redundancy trades decreased query load for increased join load, which is not as helpful when the relative rate of joins is high. However, bear in mind that super-peer redundancy is still equally effective in improving reliability.

In summary, though the benefits of various cluster sizes and super-peer redundancy still apply, they may vary in degree based on the relative cost and rates of queries to joins.

D Additional Results

Table 2 lists the aggregate load for two topologies: the first with an average outdegree of 3.1, and the second with an average outdegree of 10. Both topologies have a cluster size of 100, since the first topology yields a smaller expected number of results for any smaller cluster size.

E Caveat to Rule 3

Looking at Figure 9 in Section 5.1 again, we see that in some cases, increasing outdegree does not significantly affect EPL. For example, when desired reach is 500 nodes, increasing outdegree from 50 to 100 only decreases EPL by .14. In these cases, having a larger outdegree will not only be useless in decreasing EPL, it will actually *hurt* performance by increasing the number of redundant queries. Figure A-15 illustrates this point, showing the individual load of a super-peer with various cluster sizes, for an average outdegree of 50 and 100, and TTL=2. In each case, the desired reach is set to be the total number of super-peers, which is $\frac{\text{number users}}{\text{cluster size}}$. For all cluster sizes shown, systems with an average outdegree of 50 outperform systems with an average outdegree of 100 because the EPL is roughly the same in both cases, while redundant queries increase with outdegree. Therefore, as a modification to rule #3, outdegree should be increased, so long as doing so decreases EPL. Once EPL ceases to decrease, outdegree should not be increased any further.

This caveat has implications in both the global design procedure (note step 5 in Figure 10) and local decision-making. In terms of local decisions, it is difficult for a super-peer to detect whether it has “too many” neighbors. One possible method to detect this case is for the super-peer to increase its outdegree, and then see if it starts receiving more query responses. If the number of responses does not increase significantly, the connection should be dropped.

F Details in Predicting Global TTL

There are two useful details to keep in mind while predicting a global TTL, using the method described in rule #4 of Section 5.1. First, setting TTL too close to the EPL will cause the actual reach to be lower than the desired value, since some path lengths will be greater than the expected path length. For example, when average outdegree is 10 and reach is 500, EPL is 3.0. However, setting TTL to 3 results in a reach of roughly 400 nodes, rather than 500.

Second, note that a close approximation for EPL in a topology with an average outdegree d is $\log_d(reach)$. Such an approximation is most accurate for a tree topology where the query source is the root. In a graph topology, the approximation becomes a lower bound, as the “effective outdegree” is lower than the actual outdegree due to cycles. This approximation is useful for finding a good TTL without running experiments like the ones used to generate Figure 9.