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

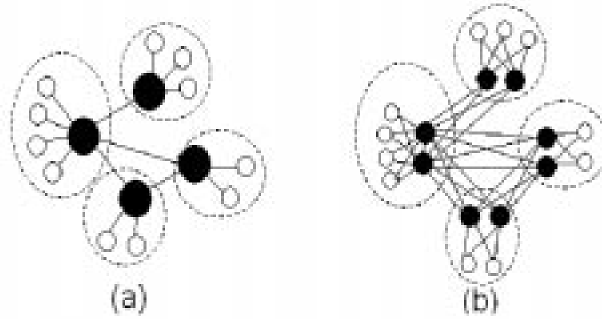


Figure 1: Illustration of a super-peer network (a) with no redundancy, (b) with 2-redundancy. Black nodes represent super-peers, white nodes represent clients. Clusters are marked by the dashed lines.

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 $n = \frac{\textit{GraphSize}}{\textit{ClusterSize}}$. 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 $\mu_c = \textit{ClusterSize} - 1$ if there is no redundancy and $\mu_c = \textit{ClusterSize} - 2$ 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.

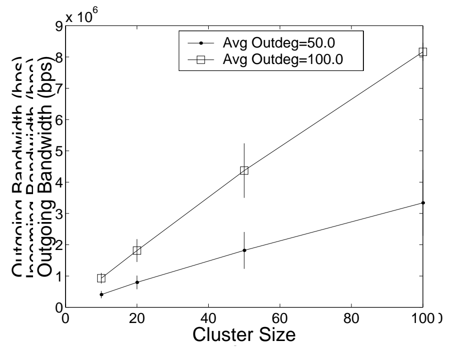
5.1 Rules of Thumb

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


x 10^6
Avg Outdeg=50.0
Avg Outdeg=100.0
Outgoing Bandwidth (bps)
Cluster Size
0
20
40
60
80
100
1
2
3
4
5
6
7
8
9