# Indexing Techniques for File Sharing in Scalable Peer-to-Peer Networks

(Extended Abstract)

Fred S. Annexstein, Kenneth A. Berman, Mihajlo A. Jovanovic, and Kovendhan Ponnavaikko
Department of ECECS
University of Cincinnati
Cincinnati, OH 45221 USA
fred.annexstein@uc.edu , ken.berman@uc.edu , kponnava@ececs.uc.edu

***Abstract*-File sharing is a very popular service provided by peer-to-peer (P2P) networks. In a P2P file-sharing network, users share files and issue queries to the network to find the locations of the files residing at other peer nodes. Due to factors such as large user base and use of query broadcast protocols, each node in the network receives many search queries every second. Recently network developers have incorporated proxy-enabled peers, or supernodes, which are designed to enhance scalability by providing indexing services to nodes on slower network connections. Typically, supernodes build a vector or multi-index of all the filenames of the shared files stored on other (slower) peers nodes connected to them. In this paper we consider a new model whereby the index tables of the individual nodes are merged into a single data structure stored by the supernode. We analyze this new model in relation to the standard vectorized data structure. We compare the performance of these supernode indexing algorithms and provide a theoretical analysis that is asymptotic and probabilistic in nature. However, there are several significant constant factors that the theory does not account for, and which in practice are important for designing an optimal system solution. We report herein on a series of simulation experiments which provide 1) verification of the asymptotic analysis of the formal framework, and 2) tools to determine the magnitude of the constant factors involved in the performance analysis. Our general conclusion is that when the query rate exceeds the rate of data updates, the new merged model is preferable to the vector model. However, the details of our analysis allow us to consider combinations of several parameters, and thereby enable the design of optimal indexing schemes via the incorporation of measurements of the parameters of particular applications.**

***Keywords --- peer-to-peer computing; text indexing; file sharing; algorithm analysis.***

## I. 1. INTRODUCTION

Peer-to-peer (P2P) networks make new services available to end-users by enabling their PCs to become active participants in computing processes. File sharing has been the most popular service for which P2P networks have been used in recent years. In a P2P file-sharing network, users share files and issue queries to the network to find out the locations of the files residing at other peer nodes.

Research is ongoing with a focus on the scalability issues in P2P networks [1,13,15]. The dimensions of scalability for P2P file sharing applications are numerous, including number of nodes hosted, the rate of connection and loss, the size and composition of shared file indexes, and the rate and scope of valid search queries. In the most popular P2P protocols, such as Gnutella [3, 20], nodes are responsible for responding to the queries that they receive, as well as forwarding the queries in a broadcast. The size of many P2P file sharing networks is quite large, with common reports exceeding several hundred thousand simultaneous users. Due to factors such as large user base and use of query broadcast protocols, each node in the network receives and must respond to many search queries every second. Recently, P2P network developers have incorporated proxy-enabled peers, or supernodes, which are design to enhance scalability by providing indexing services to nodes on slower network connections. In practice, supernodes are designed to index all the filenames of the shared files stored on other (slower) peers nodes connected to them [20].

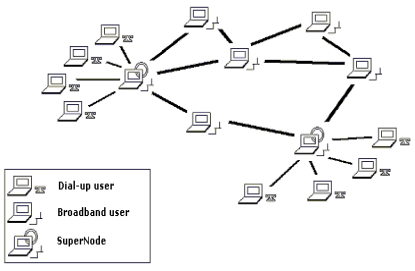


Figure 1. A P2P network using supernodes as proxies for dial-up users.

In Figure 1, we show a P2P network containing both normal host nodes and supernodes. Typically, end-users on slower connections, particularly dial-up modems, will opt to connect to public access supernodes, since they are assumed to operate on higher-speed connections. The supernode indexes the filenames of the files residing at connected hosts. The supernode subsequently responds to queries arriving from the network, and does not forward the queries to any of

Research supported in part by NSF Grant No. CCR-9877139, and an Ohio Board of Regents' Research Investment Grant.

the connected hosts. Thus the supernode shields the hosts under it from bandwidth consuming query traffic. It has been reported that the scalability and overall performance of the network significantly improves with the introduction of supernodes [20]. However, the scalability of the supernodes themselves remains an issue; since for example, the question remains how large a load can each supernode handle effectively. Let us consider this question in detail.

Note that file storage and file transfer are not an issue for supernodes. Although supernodes act as proxies for indexing purposes, they are not involved in any file transfers that may occur as a result of query hits, since these transfers occur via direct connections between hosts. The primary factors affecting the supernode performance, include the time taken to process all the queries, and the time for updating the index after connection and disconnection. In the supernode, the time it takes to insert filenames into the main index and the time it takes to delete filenames from the main index can be significant.

The indexing problem for the supernode is to maintain a dynamic index of all the filenames of all the hosts that are connected to the supernode. These dynamic multi-indexes are typically stored as a vector of index tables, which are searched sequentially with each query. In this paper we consider a new model for supernode indexing, whereby the index tables are merged into a single data structure. In this paper we analyze this new model in relation to the standard vectorized data structure. We compare the performance of supernode indexing algorithms and provide a theoretical analysis that is asymptotic and probabilistic in nature. There are, however, several constant factors that the theory does not account for, and which in practice are important for designing an optimal system solution. We report herein on a series of simulation experiments which provide 1) verification of the asymptotic analysis of the formal framework, and 2) tools to determine the magnitude of the constant factors involved in the performance analysis. For our theoretical and experimental analysis we assume Poisson distributions on the arrival rate of nodes, and an exponential distribution on the duration time that a node stays connected, as is done in [17]. These distributions are useful for modeling and analyzing a highly dynamic, yet relatively stable system. Our general conclusion is that when the query rate exceeds the rate of data updates, the new merged model is preferable to the vector model. In this paper, we give a detailed analysis that accounts for a number of variable performance factors such as number and rate of connections, size and rate of change of filenames, and query rate and query string sizes. The details of our analysis allow us to consider combinations of parameters, and thereby enable the design of optimal indexing schemes via the incorporation of measurements of the parameters of particular applications.

## II. TEXT INDEXING TECHNIQUES

Pattern matching is a classical algorithmic problem [9,10,16]. There are several significant dynamic text indexing methods discussed in the literature which enable general pattern matching, including String B-Trees and Suffix Trees [7, 12, 17], Dynamic Suffix Arrays [2, 11], and other more recent methods [4, 5, 6, 8]. With respect to the application of interest in this paper, that of pattern matching using an index of dynamic lists of filenames, the generalized suffix tree is a sufficiently optimal method. For all the performance measures of interest, i.e., for search, insertion and deletion, the time complexities, obtained by using generalized suffix trees, are independent of the size of the index. The search time is linear in terms of the length of the pattern and the number of occurrences which is best possible. The insertion and deletion times are both linearly proportional to the length of the filename to be inserted/deleted, which is again the best possible.

Suffix trees can be used to solve the exact pattern matching problem in linear time. The classic application for suffix trees is the substring problem. One is first given a text S of length m, and a string-pattern p of length n we mush either find an occurrence of p in S or determine that p is not contained in S. With the use of a suffix tree, the text is preprocessed in O(m) time; thereafter, whenever a string of length n is input the algorithm searches for it in O(n) time using that suffix tree. The O(m) preprocessing and O(n) search result for the substring problem is extremely effective for the P2P filename indexing problem, where many short sequences of requested strings will be input as queries after the suffix tree is built.

Algorithms for constructing suffix trees are described in Weiner [18] and McCreight [12]. More recently, Ukkonen [17] developed a conceptually simpler linear-time algorithm for building suffix trees. A suffix tree for an m-character text-string S is a rooted directed tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out S[i..m]. For example, the suffix tree for the string xabxac is shown in Figure 2. The path from the root to the leaf numbered 1 spells out the full string S = xabxac, while the path to the leaf numbered 5 spells out the suffix ac, which starts in position 5 of S. As stated above, the definition of a suffix tree for S does not guarantee that a suffix tree for any string actually exists. The problem is that if one suffix of S matches a prefix of another suffix of S then no suffix tree obeying the above definition is possible, since the path for the first suffix would not end at a leaf. For example, if the last character of xabxac is removed, creating string xabxa, then suffix xa is a prefix of suffix xabxa, so the path spelling out xa would not end at a leaf.

To avoid this problem, we assume (as was true in Figure 2) that the last character of S appears nowhere else in S. Then, no suffix of the resulting string can be a prefix of any other suffix. To achieve this in practice, we can add a character to the end of S that is not in the alphabet that string S is taken from. Usually the character '$' is used as the "termination" character.

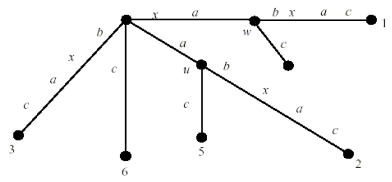

Figure 2. Suffix tree for the string "xabxac"

For our purpose of storing filenames, we use a special kind of suffix tree called the generalized suffix tree. The generalized suffix tree GST(P) for a set of strings P is the tree obtained by superimposing incrementally the suffix trees ST(p) for all the strings p in P. Two arcs are superimposed whenever their labels have a common prefix (equal suffixes can be associated with the same leaf).

While performing searches for query patterns, we must find out all the filenames that have the pattern as a substring. Due to this requirement, at each node in the generalized suffix tree, we store the file identifiers of all the filenames that have the string represented by that node as a substring.

## III. SUPERNODE DYNAMIC INDEXING SCHEMES

The frequency of connections and disconnections in a supernode can be very high. Since for every connection and disconnection, we must perform insertions into and deletions from the main index, the method used to maintain the main index plays a major role in determining the efficiency of the supernode's performance. When a node connects to the supernode, it must transfer to the supernode, the GST of all its filenames. The standard method for a supernode to add or delete a filenames index is to store a vector of pointers to the roots of the individual indexes. We identify this standard model as the *vector model VM*.

In the VM the supernode maintains a vector of GSTs, containing the GSTs of all the nodes that are connected to it. When a node connects to the supernode, its GST is transferred to the supernode and a pointer to it must be added to the vector. When the node disconnects, its tree pointer must be removed from the vector.

In this paper we consider a new model, whereby the index tables stored in the supernode are merged into a single data structure. We identify this model as the *merged tree model MTM.*

In the MTM the supernode maintains a single, primary GST. When a node connects to the supernode, the new node's GST is inserted into the primary GST of the supernode. When a node disconnects, the set of filenames associated with that node must be removed from the primary GST.

There are a number of tradeoffs when considering these two models. In the merged tree model, significantly more pre-processing of the data structure needs to be done when nodes connect and disconnect. In the vector model, little or no pre-processing is required. On the other hand, while performing searches, in the merged tree model only a single tree needs to be searched, whereas in the vector model, a collection of trees, one per connected node, must be searched. We now consider the problem of comparing the overall performances of the two schemes.

## IV. THEORETICAL ANALYSIS OF SUPERNODE INDEXING

In this section, we present a formal analysis of the performance characteristics of the two models. A large number of parameters are involved in the performance analysis, including the data set size and composition, the number of nodes and their rate of addition and deletion from the supernode, the rate and composition of queries. As noted earlier, for our analysis we assume Poisson distributions on the arrival rate of nodes, and an exponential distribution on the duration time that a node stays connected. These distributions are useful for modeling and analyzing a relatively stable system, for example it can be shown that the evolving system rapidly converges to a stable average number N of nodes connected to the supernode [17]. We focus our attention on the five parameters most relevant to performance. We assume Poisson (or exponential) distributions on all these random variables, and use the following notation to denote the parameterized mean values:

* N is the average number of connections to the supernode.

* F is the average number of files per node.

* L is the average length of a filename.

* S is the average length of a query string.

* Q is the average number of queries per unit time.

In order to compare the performances of the two schemes, we make use of a value called the response time. Formally, the response time is defined as the time taken by each scheme to perform the following two phase operations, which taken together is the sum of the processing time and the search time.

RESPONSE_TIME = Phase 1 + Phase 2, where

Phase 1: Time required to update the data structure to account for the connection of new client nodes and the disconnection of other previous client node.

Phase 2: Time required to perform and return results from a set of queries on the updated data structure.

To compute and express formally a value for RESPONSE_TIME we will normalize time units to be equal to the expected arrival rate ($\lambda = 1$) of new connections. Therefore we have that, with high probability, in Phase 1 the data structure is modified to account for O (1) node updates changes. Also, we have that, with high probability, in Phase 2 the number of queries on the updated index is O(Q). We have the following theorem that allows us to compare asymptotically the performance of the two models.

Theorem 1. For any time $t = \Omega(N)$, with high probability $p > 1 - 1/N$, the response times can be bounded above and below as follows: a) in the vector model, RESPONSE_TIME (VM) = Θ(1) + Θ(QNS), and b) in the merged tree model, RESPONSE_TIME (MTM) = Θ(FL) + Θ(QS).

Proof.

The number of nodes connected to the supernode has a Poisson distribution. Using standard techniques for bounding the tail of the distribution [**19**] it can be shown that with probability $p > 1 - 1/N$, the number of nodes connected to the supernode at any time $t = \Omega(N)$ is $\Theta(N)$, and furthermore, the number of nodes added and deleted during a unit of time is O (1). Finally, with high probability, we have that the number of queries within a time unit is $\Theta(Q)$, and furthermore, the size of each query string is $\Theta(S)$.

In the vector model, the addition and the removal times of each node are both constants because pointers to the GSTs simply need to be added to or removed from the vector, and each such operation is constant time. Thus the Phase 1 preprocessing time for VM is $\Theta(1)$. The search time for each query is linear in terms of the product of the size of each query and the number of GSTs to be searched. Thus the Phase 2 search time for VM would be $\Theta(QNS)$.

In the merged tree model, the Phase 1 processing time, depends linearly on the size of the data sets involved, and thus would require Θ(FL) time. The Phase 2 search time for MTM would be proportional to the product of the length of each query and the number of queries, and thus equal to Θ(QS). The theorem thus follows. ∴

Theorem 1 states that there are three significant parameters that determine the performance of the vector model, and that there are four significant parameters that determine the performance of the merged tree model. However, the analysis is asymptotic, ignores constant factors, and is probabilistic. In the following section we report on a series of simulation experiments which provide verification of the theoretical analysis, and permits a determination of the magnitude of the constant factors involved in the performance analysis.

## V. EMPIRICAL ANALYSIS

A series of simulations were performed to verify the correctness of the expressions for the RESPONSE_TIMEs given by Theorem 1, and to help determine the hidden constant factors. We simulated the operations of the supernode to compare the performances of the two schemes using a set of software tools created in Java. Our simulation code is available from the authors upon request.

We model the P2P networks application by simulating the dynamic execution of the supernode, where client nodes connect and disconnect in an uncoordinated and unpredictable fashion. As noted above, we model this setting by a stochastic, memoryless, continuous-time setting, see [14]. The arrival/connection of new nodes is modeled by a Poisson distribution with rate $\lambda=1$, and the duration of time a node stays connected to the supernode has an exponential distribution with parameter $\mu = 1/N$. The number of nodes connected to the supernode rapidly converges to $\lambda / \mu = N$. So in our model, a new node connects to the supernode every unit time, and the probability that a node is still connected at time $t$ is given by $e^{-(t-\tau)/N}$ where $\tau$ is the time when the node connected to the supernode and $N$ is the average number of nodes connected to the supernode.

Data sets of filenames for the client nodes were generated randomly, with set sizes given by a normal distributed. When a node connects to the supernode, it transfers to the supernode its index of filenames given by a GST. If the supernode uses the merged tree model, the GST of the new node must be merged or superimposed over the primary GST of the supernode. If the supernode uses the vector model, the GST of the new node is simply appended to the vector of GSTs stored in the supernode. Similarly when a node disconnects, its filenames must be removed from the primary GST in the merged tree model and from the vector of GSTs in the vector model.

Our simulation runs in iterations. Each iteration is divided in two phases, as discussed in Section IV. The response time is computed as the sum of the running times of these two phases. In the first phase, a new node connects to the supernode, and zero, one, or more nodes disconnect from the supernode (as determined by the distribution discussed above). The data structures are then modified to reflect these connection changes. In the second phase of the iteration., a random set of string-matching queries are generated, normally distributed in size, and the supernode executes the lookup on each query and responds accordingly. We record the time taken for a series of such two phase iterations, and plot the resulting response times. We create a series of plots by considering different independent variables. We consider three diffenent series by choosing the independent variable as 1) the mean query rate, 2) the mean file size, and 3) the mean number of simultaneous connections. Since we chose for each series one independent variable, we need to fix the other four parameters. By studying data from activity on P2P networks we selected mean values on the fixed parameters. Finally, we have divided each of the three series into three separate plots by varying one of the fixed parameters over a set of three fixed values. In what follows we have three series of experiments comparing the performance of the vector and the merged tree model,. Each series is divided into three separate plots with the same independent variable. Note also that the same randomly generated input sets was used for simulating both models.

### *A. Plotting Response Time as function of Query Rate*

The first three charts (Figure 3) illustrate the results obtained for the response times plotted against the increasing number of queries per time unit. Plots are generated for three different values of the number of files per node. It can be seen that the effect of the increase in the number of queries per time unit does not affect the response times of the merged tree model much, but has a large effect

on the response times of the vector model. For case when F=200, the performance of the vector model is better as long as the number of queries per time unit is less than 130. When the number of queries per time unit exceeds 130, the merged tree model performs better.

### *B. Plotting Response Time as a function of File Size*

The next three charts (Figure 4) illustrate the results obtained for the response times plotted against an increasing number of files per node. Plots are shown for three different values of the number of connected nodes. It can be seen that the effect of the increase in the number of files per node does not affect the response times of the vector model much, but has a significant effect on the response times of the merged tree model. For the case when N=80, the performance of the merged tree model is better as long as the number of files per node is less than 150. When the number of files per node exceeds 150, the vector model performs better.

### *C. Plotting Response Time as function of Number of Connections*

The final three charts (Figure 5) illustrate the results obtained when the response times were plotted against an increasing mean number of connected nodes. Plots are shown for three different values of the number of queries per time unit. It can be seen that the effect of the increase in the number of connected nodes does not affect the response times of the merged tree model much, but has a large effect on the response times of the vector model. For the case Q= 100, the performance of the vector model is better as long as the mean number of connected nodes is less than 100. When the number of connected nodes exceeds 100, the merged tree model performs better.

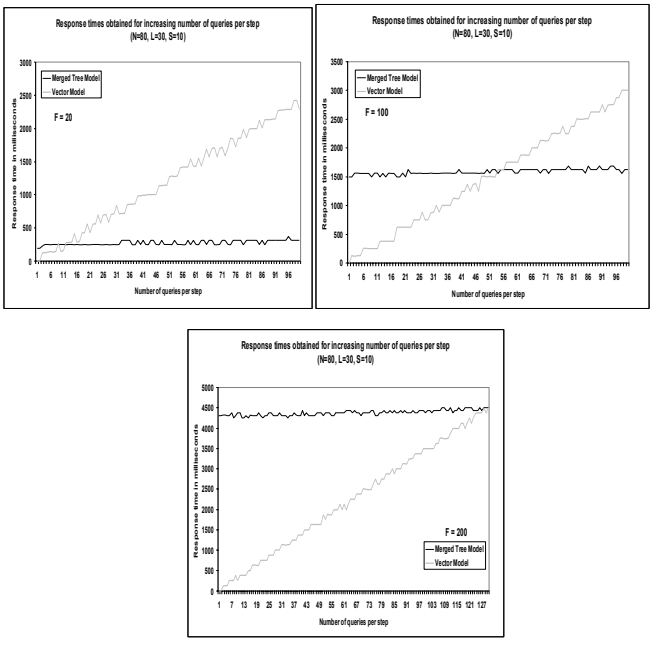


Figure 3. Response times plotted as a function of increasing number of queries per time unit. One plot for each of 3 different *F* values (*F=20, F=100, F=200*). This test series used fixed average values *N=80, L=30, S=10*.

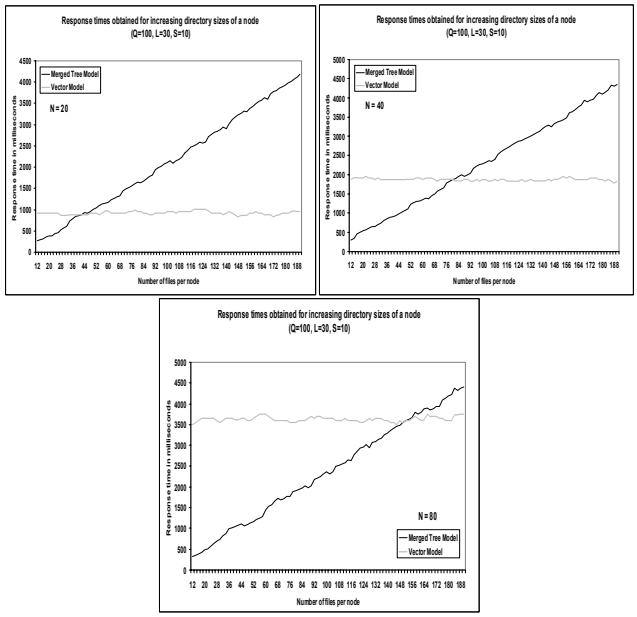


Figure 4. Response times plotted as a function of increasing average filename sizes per time unit. One plot for each of 3 different *N* values (*N=20, N=40, N=80*). This test series used fixed average values *Q=100, L=30, S=10*.

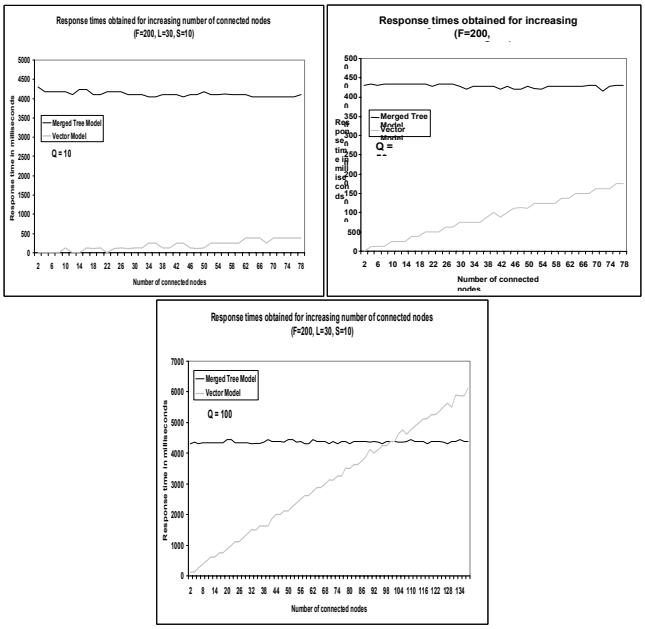


Figure 5. Response times plotted as a function of increasing average number of connections to the supernode. One plot for each of 3 different *Q* values (*Q=10, Q=50, Q=100)*. This test series used fixed average values *F=200, L*=30, *S*=10.

### *D. Estimates on Constant Factors*

All the results discussed above were obtained from simulations run on a system with a 1 GHz processor and a 1 GB RAM. Due to the large memory capacity, the whole index could be loaded into the memory and processing could be done at a very high speed. When the simulations were run in a system with a 512 MHz processor and 128 MB RAM,

the whole index could not be loaded into the memory. In such conditions, it was noticed that the performance of merged tree model was always much worse than the performance of the vector model, due to memory access overheads. For our reported analysisWe only considered cases where the supernode has a high speed processor and a large capacity RAM. A simple "eyeball" analysis of our results shows the following constant factors:

When N, L and S are fixed at 80, 30 and 10 respectively, the expressions for the response times are

Response_Time (MTM) = 20 F + 1.15 Q
Response_Time(VM) = 34 Q

From the two expressions, we notice that when 1.5 Q > F, MTM performs better, otherwise the VM performs better.

When Q, L and S are fixed at 100, 30 and 10 respectively, the expressions for the response times are

Response_Time (MTM) = 20 F + 150
Response_Time (VM) = 43 N

From these two expressions we notice that when 2 N > F, MTM performs better, otherwise the VM performs better.

When F, L and S are fixed at 200, 30 and 10 respectively, the expressions for the response times are

Response_Time (MTM) = 4250 + 1.5 Q
Response_Time(VM) = 0.44 QN

From these two expressions, we find that when QN > 8000, MTM performs better, otherwise the VM performs better.

Hence, overall our general conclusions can be summerized as follows: when the query rate exceeds the rate of data updates, the new merged model is preferable to the vector model. However, the fine details of our analysis allow us to consider combinations of several parameters, and thereby enable the design of near optimal indexing schemes via the incorporation of measurements of the parameters of particular applications.

## VI. CONCLUSIONS AND FUTURE WORK

We studied the problem of determining an optimal indexing scheme for filesharing in a scalable P2P network system. We analyzed two different methods of indexing using a supernode as a proxy, both methods based on using generalized suffix trees for the indexing. The two different algorithmic models, the vector model and the merged tree model, were analyzed theoretically using probabilistic and asymptotic analysis. These results were complemented by an empirical study using a simulation of the execution of the supernode in a dynamic environment. The empirical results provided both a validation of the theoretical results and details about the relevant constant factors involved in performance.

From our results, we posit that an optimal P2P supernode indexing technique would be a hybrid of the two schemes which could respond to changing parameter values. The supernode could maintain both a primary GST and a vector of GSTs. Whenever a node connects to the supernode, its GST could be added to the vector of GSTs. The decisions as to which GSTs should be merged or inserted into the primary GST, could be based on several different factors determined by a priori information or usage patterns. The decision criteria could be a complex one, based on the expected connection duration, the directory sizes of the nodes, the current load determined by the number of connected nodes, and on the current query rate. Future work is required on more realistic probabilistic models of connection patterns, effective strategies for implementing hybrid indexing schemes, and finally, studying the impact of hash-type indexing and filtering techniques (such a Bloom filters) on improving overall performance.

## BIBLIOGRAPHY


[1] Annexstein, F.S., Berman, K.A. and Jovanovic, M. Latency Effects on Reachability in Large-scale Peer-to-Peer Networks. Proc. 13th ACM Symposium on Parallel Algorithms and Architectures, 2001.

[2] Amir, A., Farach, M., Galil, Z., Giancarlo, R., and Park, K. Dynamic dictionary matching. Journal of Computer and System Sciences, 49(2):208-222, 1994.

[3] Clip2. The Gnutella Protocol Specification v0.4 (Document Revision 1.2). Internet document (www.clip2.com/GnutellaProtocol04.pdf), 2000.

[4] Ferragina, P., and Grossi, R. Fast incremental text indexing. Proc. 6th ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 531-540, 1995.

[5] Ferragina, P. Dynamic text indexing under string updates. Journal of Algorithms, 22(2):296-328, 1997.

[6] Ferragina, P., and Grossi, R. Improved dynamic text indexing. Journal of Algorithms, 31(2):291-319, 1998.

[7] Grossi, R., and Italiano, G.F. Suffix trees and their applications in string algorithms. Proc. 1st South American Workshop on String Processing, pages 57-76, 1993.

[8] Gu, M., Farach, M., and Biegel, R. An efficient algorithm for dynamic text indexing. Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 697-704, 1994.

[9] Karp, R.M., Miller, R.E., and Rosenberg, A.L. Rapid identification of repeated patterns in strings, trees and arrays. ACM STOC '72, pages 125-136, 1972.

[10] Knuth, D.E., Morris, J.H., and Pratt, V.R. Fast pattern matching in strings. SIAM Journal on Computing, 6:323-350, 1977.

[11] Manber, U., and Myers, E.W. Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing, pages 935-948, 1993.

[12] McCreight, E.M. A space-Economical Suffix Tree Construction Algorithm. Journal of the ACM, 23(2):262-272, 1976.

[13] Oram, A., Editor. Peer-to-Peer: Harnessing the Power of Disruptive Technology, O'Reilly, 2001.

[14] Pandurangan, G., Raghavan, P., and Upfal, E. Building Low-Diameter P2P Networks. Proc. 42nd IEEE Symposium on Foundations of Computer Science, 2001.

[15] Shirky, C. What is P2P… and what isn't. Internet document (http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html), 2000.

[16] Shoaff, W. Pattern Matching in Text. Internet document (http://www.cs.fit.edu/~wds/classes/algorithms/Text/text.ps), 1999.

[17] Ukkonen, E. On-line construction of suffix trees. Algorithmica, 14(3):249-260, 1995.

[18] P. Weiner. Linear Pattern Matching Algorithms. Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, pp1-11, 1973

[19] Ross, S.M.. Applied Probability Models with Optimization Applications, Holden-Day, San Francisco, 1970

[20] Gnutella Projects, www.gnutella.com and www.limewire.com