EAD: An Efficient and Adaptive Decentralized File Replication Algorithm in P2P File Sharing Systems

Haiying Shen
Department of Computer Science and Computer Engineering
University of Arkansas, Fayetteville, AR 72701
hshen@uark.edu

Abstract

In peer-to-peer file sharing systems, file replication technology is widely used to reduce hot spots and improve file query efficiency. Most current file replication methods replicate files in all nodes or two endpoints on a client-server query path. However, these methods either have low effectiveness or come at a cost of high overhead. This paper presents an Efficient and Adaptive Decentralized file replication algorithm (EAD) that achieves high query efficiency and high replica utilization at a significantly low cost. EAD enhances the utilization of file replicas by selecting query traffic hubs and frequent requesters as replica nodes, and dynamically adapting to non-uniform and time-varying file popularity and node interest. Unlike current methods, EAD creates and deletes replicas in a decentralized self-adaptive manner while guarantees high replica utilization. Simulation results demonstrate the efficiency and effectiveness of EAD in comparison with other approaches in both static and dynamic environments. It dramatically reduces the overhead of file replication, and yields significant improvements on the efficiency and effectiveness of file replication in terms of query efficiency, replica hit rate and overloaded nodes reduction.

1 Introduction

Over the past years, the immense popularity of Internet has produced a significant stimulus to peer-to-peer (P2P) file sharing systems, where a file requester's query will be forwarded to a file provider in a distributed manner. They can be used in video-on-demand service and shared digital library applications where individuals dedicate files which are available to others.

P2P file sharing systems have been commonly used in today's Internet. A recent large scale characterization of HTTP traffic [20] has shown that more than 75% of Internet traffic is generated by P2P file sharing applications. The median file size of these P2P systems is 4MB which represents a thousand-fold increase over the 4KB median size of typical web objects. The study also shows that the access to these files is highly repetitive and skewed towards the most popular ones. In such circumstances, if a server receives many requests at a time, it could become overloaded and consequently cannot reply the requests quickly. Therefore, highly-popular files (i.e., hot files) or flash crowds can exhaust the bandwidth capacity of the servers, and lead to low efficiency of file sharing.

File replication is an effective method to deal with the problem of server overload by distributing load over replica nodes. It helps to achieve high query efficiency by reducing server response latency and lookup path length (i.e., the number of hops in a lookup path). A replica *hit* occurs when a file request is resolved by a replica node rather than the file owner. *Replica hit rate* denotes the percentage of the number of file queries that are resolved by replica nodes among total queries. Higher hit rate means higher effectiveness of a file replication method.

Recently, numerous file replication methods have been proposed. They can be generally classified into three categories denoted by *ServerSide*, *ClientSide* and *Path*. *ServerSide* replicates a file close to the file owner [18, 4, 23, 27], *ClientSide* replicates a file close to or at a file requester [8, 6], and *Path* replicates on the nodes along the query path from a requester to a file owner [16, 28, 3]. However, most of these methods either have low effectiveness on improving query efficiency or come at a cost of high overhead.

By replicating files on the nodes near the files' owners, *ServerSide* enhances replica hit rate and query efficiency. However, it cannot significantly reduce path length because replicas are close to the file owners. It may overload the replica nodes since a node has limited number of neighbors. On the other hand, *ClientSide* could dramatically improve query efficiency when a replica node queries for its replica file, but such case is not guaranteed to occur as node interest

varies over time. Moreover, these replicas have low chance to serve other requesters. Thus, *ClientSide* cannot ensure high hit rate and high replica utilization. *Path* avoids the problems of *ServerSide* and *ClientSide*. It provides high hit rate and greatly reduces lookup path length. However, its effectiveness is outweighed by its high cost of overhead for replicating and maintaining much more replicas. Furthermore, it may produce under-utilized replicas.

Since more replicas lead to higher query efficiency and vice versa, a challenge for a replication algorithm is how to minimize replicas while still achieving high query efficiency. To deal with this challenge, this paper presents an Efficient and Adaptive Decentralized file Replication algorithm (EAD) that achieves high query efficiency and high replica utilization at a significantly low cost. A novel feature of EAD is that it achieves an optimized tradeoff between query efficiency and overhead of file replication. Instead of creating replicas on all nodes or two ends on a client-server path, EAD chooses query traffic hubs (i.e., query traffic conjunction nodes) as replica nodes to ensure high replica hit rate. It achieves comparable query efficiency to *Path* but creates much less replicas. Unlike *ClientSide*, it guarantees high hit rate since a replica node is the query traffic hub of the file. Compared to *ServerSide*, it reduces lookup path length dramatically and avoids overloading replica nodes.

Another novel feature of EAD is that it adaptively adapts the file replica nodes to non-uniform and time-varying file popularity and node interest based on recent query traffic in a decentralized self-adaptive manner. Unlike other algorithms in which a file owner determines where to create or delete replicas in a centralized fashion, EAD lets nodes themselves decide whether to store or delete replicas based on the query traffic. This self-adaptive manner enhances EAD's scalability and meanwhile guarantees high utilization of replicas. Furthermore, EAD employs an exponential moving average technique to reasonably measure file query traffic, which is critical to the effectiveness of EAD. EAD is independent of P2P structure, and is also applicable for the caching of files and metadata (i.e., routing hints). We intend our results to be applicable to both structured P2Ps [24, 17, 29, 13, 14] and unstructured P2Ps [6, 5, 11, 9]. In this paper, we will take structured P2P system to explain the EAD algorithm.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative file replication approaches for P2P systems. Section 3 presents the EAD file replication algorithm. Section 4 shows the performance of EAD in comparison with other approaches with a variety of metrics, and analyzes the factors effecting file replication performance. Section 5 concludes this paper.

2 Related Work

As mentioned, most current file replication methods generally can be classified into three categories: *ServerSide*, *ClientSide* and *Path*. Some proposed approaches use the combination of them. In the *ServerSide* category, PAST [18] replicates each file on a set number of nodes whose IDs match most closely to the file owner's ID. It uses file caching along the lookup path to minimize query latency and balance query load. Similarly, CFS [4] replicates blocks of a file on nodes immediately after the block's owner. It also caches a file location hint along a path to improve query efficiency. Stading *et al.* [23] proposed to replicate a file in locality close nodes near the file owner. Overlook [27] places a replica of a file on a node with most incoming lookup requests for fast replica location. It needs to keep track of client-access history to decide the replica nodes.

In the *ClientSide* category, Gnutella [6] replicates files in overloaded nodes at the file requesters. In LAR [8], a node makes decisions of file replication based on its load. LAR specifies the overloaded degree of a server that a file should be replicated. In addition to replicating a file at the requester, it also has file location hint along the lookup path. Backslash [23] pushes cache to one hop closer to requester nodes as soon as nodes are overloaded.

In the *Path* category, Freenet [1] replicates files both on insertion and retrieval on the path from the requester to the target. PAST [18], CFS [4], LAR [8], CUP [16] and DUP [28] perform caching along the query path. Cox *et al.* [3] studied providing DNS service over a P2P network. They cache index entries, which are DNS mappings, along query paths.

EAD replicates a file at nodes in a path with high query load of the file, which improves replica hit rate and query efficiency and meanwhile reduces replicas. Unlike most of these methods, in which a file owner needs to keep track of replica nodes for replica creation and removal, EAD conducts the replica adjustment in a decentralized manner. EAD shares similarity with the works in [22, 12, 15] in terms of replica node selection and replica adaptation. IRM [22] aims at integrating file replication and consistency maintenance in a systematic manner in order to improve the efficiency of both file replication and consistency maintenance. EAD focuses on the efficiency improvement of only file replication. EAD addresses more detailed issues in file replication, and provides a strategy for query rate determination. In addition, this paper comprehensively studies EAD through simulations. RaDaR [12] deploys algorithms for deciding on automatic replication and migration of content. It considers server load to balance the load among the servers, and the proximity of clients to servers to place replicas in one proximity of clients from which

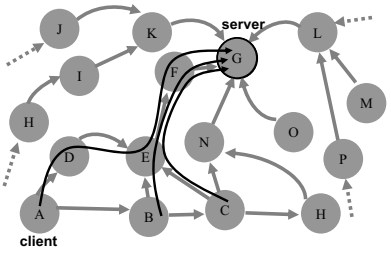


Figure 1. File queries in a file sharing system.

most of the requests originate. Moreover, to be scalable, the algorithms for making these decisions rely only on a limited amount of information about the system. Spread [15] considers network-level adaptation of replica placement. A proxy cache expresses interest in the object by subscribing to the replication service. A file is replicated by using push or a periodic-pull.

There are other studies for file replication in unstructured P2Ps [2, 25, 26, 19]. Since unstructured P2P systems use flooding or random probing based methods for file location, the number of replicas directly affects the efficiency of file query. These works study the system performance such as successful queries and bandwidth consumption when the number of replicas of a file is proportional, is uniform and is square-root proportional to the query rate. The works focused on the relationship between the number of replicas, file search time and load balance, but did not investigate the impact of replica location on file query efficiency. EAD aims to study the locations of replicate files to achieve comparably query efficiency with less replicas.

Splitting a large file into small pieces can increase the service capacity of a large file rapidly. Replicating file location hint along query path can also improve file query efficiency. EAD can employ the techniques to further improve its performance. These techniques are orthogonal to our study in this paper.

3 EAD File Replication Algorithm

In this section, we describe the EAD algorithm. We start off by describing the goals of EAD and the strategies to achieve the goals. Then, we discuss the various aspects of the algorithm in detail. Finally, we present a summary of the EAD algorithm.

3.1 Goals and Strategies

In a P2P file sharing system, overloaded conditions are common during flash crowds or when a server hosts a hot file. For example, in Figure 1, if many nodes query for a hot file in node G at a time, G will be overloaded, leading to delayed file query response. File replication is an effective method to deal with the problem of overload condition. For instance, node G replicates its file to node K, F and N, thus a file query may encounter replica nodes before it arrives at node G. By replicating a hot file to a number of other nodes, the file owner distributes load over replica nodes, leading to quick file response. A file query may encounter replica nodes before it arrives at file owner, reducing lookup path length. Thus, replication helps achieve high file query efficiency due to lookup path length reduction and quick query response.

In *ServerSide*, node G will choose its neighbors K, F, N, O and L as options for replica nodes. Though it has high hit rate, it cannot significantly reduces the lookup path length and may overload the neighbors. On the other hand, *ClientSide* replicates a file to requesters A, B and C. Replicating files close to or in the file requesters brings benefits when the requester or its nearby nodes always query for the file. However, considering non-uniform and time-varying file popularity and node interest variation, the replicas may not be fully utilized. Thus, *ClientSide* cannot guarantee high hit rate, though it can reduces lookup path length when hit occurs. *Path* replicates the file in all path nodes D, E and F. It has high hit rate and significantly reduces lookup path length, but comes at high cost of much more replicas.

The ultimate objective of EAD is to achieve an optimized tradeoff between query efficiency and file replication overhead by increasing hit rate and reducing replicas. Specifically, it EAD aims to overcome the drawback of these methods with two goals. Firstly, it aims to minimize replicas and achieve high file query efficiency. More replicas lead to higher query efficiency and vice verse. How can a replication algorithm reduces replicas without compromising query efficiency? Rather than statically replicating a file along a query path, EAD replicates a file in nodes with high query traffic of the file, thus reducing replicas while ensuring high hit rate and comparable query efficiency.

Secondly, rather than depending on a file owner to determine replica creation and deletion in a centralized manner, EAD aims to conduct the operations in a decentralized manner without compromising replica utilization. As P2P systems can be very large, decentralized replication decision making is key to scaling the system. For example, the popular KaZaA file-sharing application routinely supports on the order of two million simultaneous users, exporting more than 300 million files. To achieve this objective, EAD uses self-adaptive method in which nodes themselves decide whether to be replica nodes, and replica nodes themselves determine whether to delete replicas based on their query traffic.

EAD does not address P2P churn where nodes join and leave the system continually. These actions are handled by the underlying P2P system and should not affect the appli-

cability of the replication algorithm.

3.2 Algorithm Description

The basic idea of EAD is replicating a file in a node with high query traffic of the file, so that more queries will encounter the replica node, leading to high hit rate. To deal with time-varying file popularity and node interest, EAD adaptively adjusts the file replica nodes based on recent query traffic in a decentralized manner.

We discuss EAD from three aspects of file replication: (1) where to replicate files so that the file query can be significantly expedited and meanwhile the file replicas can be taken full advantage of? (2) how to conduct the creation of hot files and the deletion of under-utilized replicas in a decentralized manner for high replica utilization? (3) how to reasonably measure file query traffic for replica adjustment?

3.2.1 Efficient File Replication

In P2P file sharing systems, some nodes carry more query traffic load than others [7, 21]. It is mainly caused by three reasons. First, node interests are different. There will be more query traffic along the query paths from the frequent file requesters and the file owner. Second, file popularity is non-uniform and time-varying. Nodes receiving and forwarding hot file queries take on more query traffic load. Third, nodes are located in different places and may have different number of neighbors in P2P overlay network. Nodes in some overlay areas with hot files or with more neighbors will experience more query traffic. For instance, in Figure 1, because nodes A, B and C are very interested in a hot file in node G, all the queries need to pass through nodes E and F before they arrive at file server G. Thus, nodes E and F forward much more queries for the file than others.

Based on this observation, we can choose query traffic hubs E and F as replica nodes, so that the queries from different direction can encounter the replica nodes, increasing the replica hit rate. The efficiency of this strategy is determined by whether the replica nodes serve as query traffic hubs for a certain time period.

A study shows that the access to P2P files is highly repetitive and skewed towards the most popular ones [20]. We define *visit rate* of a node as the number of queries the node receives during a unit time period T.

Theorem 3.1 *In a P2P system, a node with a highly-popular file or high visit rate has high probability to be visited again by a random node.*

Proof Let v_i represent the the visit rate of node i, and p_i represent the probability that node i receives a query during unit time period T. p_i equals to the ratio of the number of queries node i receives versus the total number of queries in the system during T. That is, $p_i = v_i / \sum_{j=1}^{n} v_j$, where n is the total number of nodes in the system. Let $\tilde{p}_i$ denotes the probability that node i is visited by a random node. Assume $\tilde{p}_i$ follows Poisson distribution as in [24], then after k visits, $\tilde{p}_i = 1 - (1 - p_i)^k$. When v_i increases, p_i will increase, subsequently $\tilde{p}_i$ will increase. ∎

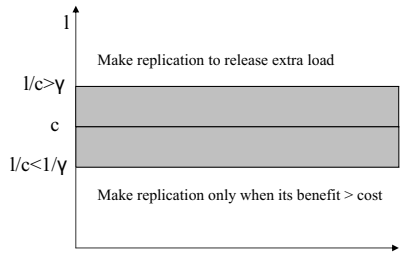


Figure 2. File replication decision by a file server.

Theorem 3.1 implies that traffic hubs have high probability to continue to be traffic hubs for a certain time period. Thus, traffic hubs usually have higher traffic than other nodes in a lookup path for a time period. In order to release file owner's load quickly and enhance the utilization of file replicas, traffic hubs in query paths and frequent file requesters should be the replica nodes.

Therefore, EAD replicates a file in nodes that have been carrying more query traffic of the file or nodes that query the file frequently. The former increases the probability that queries from different directions encounter the replica nodes, and the latter provides files to the frequent file requesters without query routing, thus increasing replica hit rate. In addition, replicating file in the middle of a query path rather than in a node near the server as in *ServerSide* speeds up the file query.

We define *query rate* of a file f, denoted by q_f, as the number of queries initiated by a requester or forwarded by a node during unit time period T. q_f should be indexed by different files such as q_{f_1} and q_{f_2}, but for brevity, we omit the indices here. A technique for reasonably determining q_f will be introduced in Section 3.2.3. EAD sets a threshold for query rate, T_q; $T_q = \alpha\bar{q}$, where α is a constant parameter, and $\bar{q}$ is the average query rate in the system.

$$\bar{q} = \sum_{j=1}^{m} q_{f_j}/m,$$

where m is the number of files in the system. If a node's $q_f > T_q$, it is regarded as a frequent requester or traffic hub for file f. A node periodically calculates its q_f. If $q_f > T_q$ and it has enough capacity such as storage space or bandwidth for a file replica, it incorporates a file replication request and its q_f into a file query when initiating or forwarding a file request. A traffic hub also needs to incorporate its

IP address and ID into the file query. Including replication requests into file queries avoids additional overhead of the file replication algorithm.

In addition to the original owner of a file, a replica node can also replicate the file to other nodes. We use *server* to denote both the original file owner and replica nodes. We use a server's visit rate to represent its query load denoted by l. We use c to denote a node's capacity represented by the number of queries it can respond during T. We use *node utilization* to denote the fraction of a node capacity that is used, represented by l/c. Each server i records its query load l_i over T periodically, and checks whether it is overloaded or lightly loaded by a factor of γ_l; *i.e.* whether

$$l_i/c_i > \gamma_l \text{ or } < 1/\gamma_l,$$

as shown in Figure 2. In the former case, it releases $(l_i - \gamma_l c_i)$ query load units by replicating files. In the latter case, though it is not overloaded, replication may enhance query efficiency. Therefore, it makes decision of file replication based on the benefits and cost brought about by the file replication.

When overloaded, a file's server selects the nodes with high query rates to be the replica nodes to release its load. Specifically, the server firstly orders the replication requesters based on their q_f in a descending order. Then, it retrieves replication requesters in the list one at a time, and replicates f at the requester until

$$\sum q_f \geq (l_i - \gamma_l c_i),$$

which makes the server lightly loaded. This scheme guarantees that nodes with higher query rates have higher priorities to be replica nodes, leading to higher replica hit rate. In the case that there is no file replication request, then a server replicates file f to its neighbors that forward the queries of f most frequently.

If the file server is not overloaded, it makes file replication only when the benefit brought about by the replication is greater than its cost. In practice, a node has various capacities in terms of bandwidth, memory storage, processing speed, and etc. We assume that different capacities can be represented by one metric. If a file is replicated in a requester with q_f and d hop distance to the file server, it can save the query forwarding resource of $q_f \times d \times \bar{l}_q$, where $\bar{l}_q$ is the resource consumption for forwarding one query. On the other hand, it costs storage resource r for a replica. If the benefit of the file replication is greater than its cost; that is,

$$q_f \times d \times l_q > r,$$

then the file server makes a replication in the requester.

3.2.2 Decentralized File Replica Adaptation

Considering that file popularity is non-uniform and time-varying and node interest varies over time, some file replicas become unnecessary when there are few queries for these files. To deal with this situation, EAD adaptively remove and create file replicas adaptively.

In previous methods, a file server maintains information of its replica nodes to manage the replicas and disseminates information about new replica sets. Rather than depending on such a centralized method, EAD makes replica adjustment in a decentralized manner. EAD lets nodes themselves determine if they should have replicas or delete replicas based on their actually experienced query traffic. If a node has higher query traffic of a file, it requests to be a replica node of the file. On the other hand, if a replica node receives less queries of a replica, it removes the replica. Such decentralized adaptation help to guarantee high hit rate and replica utilization. In addition, it reduces the extra load for replica information maintenance in file servers, making the replication algorithm more scalable.

Specifically, EAD lets each node periodically update its query rate of each file. If a node's $q_f > T_q$, it requests to have a replica as introduced in the previous Section. If a replica node's $q_f < \delta T_q$, where δ is a under-loaded factor. It means that the utilization of the replica is low, the replica node records the replica as infrequently-used replica. When the $q_f < \delta T_q$ condition occurs for a specified number of time periods, or when the node needs more space for other replicas, the node removes the replica. Therefore, the determination of keeping file replicas is based on recently experienced query traffic due to file popularity and node interest. When a file is no longer requested frequently, there will be less file replicas for it. The adaptation to query rate ensures that all file replicas are worthwhile and there is no waste of overhead for the maintenance of unnecessary replicas, thus ensuring high replica utilization.

Algorithm 1: Pseudo-code for EAD file replication algorithm.

//executed by a file requester
periodically calculate q_{f_t} by $q_{f_t} = \beta y_{t-1} + (1-\beta) q_{f_{t-1}}$
if $q_{f_t} > \alpha T_q$ **then**
  if query for file f **then**{
    include replication request into the query}

//executed by a query forwarding node
periodically calculate q_{f_t} by $q_{f_t} = \beta y_{t-1} + (1-\beta) q_{f_{t-1}}$
if $q_{f_t} > \alpha T_q$ **then**
  if receive a query for file f to forward **then**{
    include replication request into the query}

//executed by a file server i
periodically calculate l_i
if it is overloaded by a factor of γ_l {
  if there are file replication requests during T {
    order replication requesters based on their q_f

in a descending order
while $\sum q_{f_t} < (l_i - \gamma_l c_i)$ **do**{
replicate file to replication requester on the top of the list
remove the replication requester from the top of the list}}
else
replicate file to the neighbor nodes that most
frequently forward queries for file f}
else
for each requested file replication by a node with q_{f_t}
if $q_{f_t} \times d \times l_q > r$
make a replication to the replication requester

//executed by a replica node
for each replica of file f **do**
periodically calculate q_{f_t} by $q_{f_t} = \beta y_{t-1} + (1-\beta) q_{f_{t-1}}$
if $q_{f_t} \leq \delta T_q$ **do**
remove file replica

With the decentralized adaptation algorithm, when a file is becoming more and more popular, the replicas of the file will spread wider and wider to the traffic hubs and requesters in the system, and the query load of the file is distributed among the replica nodes. From the perspective of the entire system, file query can be resolved more efficiently at relatively lower cost of storing replicas. When a file is becoming less and less popular, its replicas will be removed from the system until a balanced condition is reached, where no node is overloaded by the file's queries, and the all replicas are fully utilized.

3.2.3 Query Rate Determination

File popularity and node interest vary over time. For example, a file may suddenly become hot for a very short period of time and then changes back to cold. In this case, based on the file replication algorithm introduced, a number of nodes replicate the file when they observe high q_f of the file, and then remove the replicas when q_f is low after the next periodical measurement, leading to replica fluctuation and a waste of replication overhead. To deal with this problem, rather than directly using the periodically measured results, EAD employs exponential moving average technique (EMA) [10] to reasonably determine file query rate over time period T.

EMA assigns more weight to recent observations without discarding older observations entirely. It applies weighting factors to older observed q_f, so that the weight for each older q_f decreases exponentially. The degree of decrease is expressed as a constant smoothing factor β, a number between 0 and 1.

The observation at a time period t is designated y_t, and the value of the query rate at any time period t is designated q_{f_t}. The formula for calculating q_{f_t} at time periods $t \geq 2$ is

$$q_{f_t} = \beta y_{t-1} + (1-\beta) q_{f_{t-1}}.$$

Smaller β makes the new observations relatively more important than larger β, since a higher β discounts older observations faster. Thus, node i observes the number of queries for file f periodically, and computes q_f using the EMA formula. EMA-base query rate calculation helps to reasonably measure query traffic, which is critical to EAD's effectiveness.

Table 1. Simulated environment and parameters.

Parameter	Default value
File distributon	Uniform over ID space
Number of nodes	4096
Node capacity c	Bounded Pareto: shape 2 lower bound: 500 upper bound: 50000
Number of queried files	50
Number of queries per file	1000
Number of replication operations	5-25
T_q	5
γ_l	1
β, δ	0.5
T	1 second

3.3 Summary

Unlike other methods that replicate files without considering actual query traffic, EAD guarantees high replica utilization by only replicating highly-popular files or frequently-requested files in their interested requesters or traffic hubs that are capable of a replica. Requesters that frequently query for a file can get the file from itself without query routing. Traffic hubs provide files to requesters from different directions without further request routing.

In the previous methods, servers periodically compare their load to local maximum and desired loads. High load causes a server to attempt creation of new replicas, and low load causes a server to delete replicas. The centralized file replica management is not scalable, and cannot guarantee high replica utilization. Decentralized adaptation strategy in EAD lets nodes themselves determine to create or delete file replicas based on their experienced query traffic. This decentralized manner has higher scalability, and ensures high replica utilization. Algorithm 1 shows the pseudocode of EAD file replication algorithm integrating the different strategy components.

4 Performance Evaluation

We designed and implemented a simulator for evaluating the EAD algorithm based on Chord P2P system [24]. We use *system utilization* to represent the fraction of the system's total capacity that is used, which equals to $\sum_{i=1}^{n} l_i / \sum_{i=1}^{n} c_i$.

We compared the performance of EAD with *ServerSide*, *ClientSide* and *Path* in both static and dynamic environments. Experiment results show that EAD achieves high hit rate, and balanced load distribution with less file replicas. Moreover, EAD is resilient to P2P churn, where nodes join in and leave the system continually and frequently. In addition, EAD's decentralized adaptation strategy is effective in guaranteeing high replica utilization.

To be comparable, we used the same number of replication operations when a server is overloaded in all replication algorithms. In a replication operation, the server randomly chooses one of its neighbors in *ServerSide*, chooses a frequent requester in *ClientSide*, and chooses all nodes in a lookup path in *Path* to replicate a file. Therefore, EAD, *ServerSide* and *ClientSide* replicate a file to a single node while *Path* replicates to a number of nodes in one replication operation.

We assumed bounded Pareto distribution for node capacities. This distribution reflects the real world where there are machines with capacities that vary by different orders of magnitude. The file requesters and requested files in the experiment were randomly chosen. File lookups were generated according to a Poisson process at a rate of one per second as in [24]. Table 1 lists the parameters of the simulation and their default values. The values are reasonably chosen. γ_l equals to 1 means that a node is overloaded when its query load exceeds its capacity. Though different parameter value setting affects the absolute experiment results, it will not affect the relative performance between the tested replication algorithms. Thus, it will not affect the conclusions drawn from the experiment results.

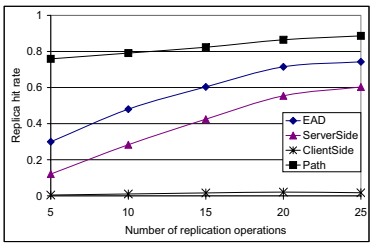
4.1 Effectiveness of Replication Algorithm

Figure 3(a) demonstrates the replica hit rate of different algorithms. We can observe that *ClientSide* generates the least hit rate, EAD has higher hit rate than *ServerSide*, and *Path* leads to higher hit rate than EAD. In *ClientSide*, files are replicated in frequent requesters, other requesters' queries have low possibility of passing through these replicas. Moreover, the replica nodes may not request the same file later due to time-varying node interest. Consequently, *ClientSide* is not able to make full use of replicas, and it has very low replica hit rate. *ServerSide* replicates a file near its owner, such that a query for the file has high probability to encounter a replica node before it arrives at the file owner. The result that EAD leads to higher hit rate than *ServerSide* is particularly intriguing given that they have the same number of replicas. Though *ServerSide* has high possibility for a query to meet a replica node near the file server, it is not guaranteed. EAD replicates a file at frequent requesters or traffic hubs, ensuring high hit rate. *Path* replicates files at nodes along the routing path, more replica nodes render higher possibility for a file request of meeting a replica node. However, its efficiency is outweighed by its prohibitively cost of overhead for keeping track of query paths and maintaining much more file replicas.

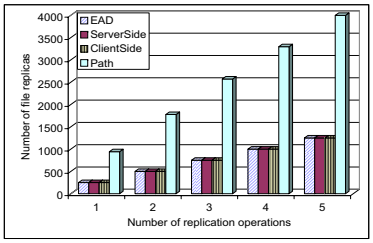
4.2 Overhead and Load Balance of Replication Algorithm

Figure 3(b) illustrates the total number of replicas in different algorithms. It shows that the number of replicas increases as the number of replication operations increases. This is due to the reason that more replication operations for a file produce more replicas. The number of replicas of *Path* is excessively higher than others, and that of others keep almost the same. It is because in each file replication operation, a file is replicated in a single node in *ServerSide*, *ClientSide* and EAD, but in multiple nodes along a routing path in *Path*. Therefore, *Path* generates much higher overhead and consumes high bandwidth for file replication and replica maintenance.

This experiment demonstrates the load balance among replica nodes in each replication method. Figure 4 plots the median, 1st and 99th percentiles of node utilizations versus system utilization. *Path* distributes load among much more replica nodes, so its load balance result is not comparable to others. Therefore, we did't include the results of *Path* into the figure. The figure demonstrates that the 99th percentile of node utilization of *ServerSide* is much higher than others. It is because *ServerSide* relies on a small set of nodes within a small range around the overloaded file owner, which makes these replica nodes overloaded. In contrast, *ClientSide* and EAD replicate files in widely distributed nodes. The figure also shows that the 99th percentile of node utilization of EAD is constrained within 1, while those of *ClientSide* and *ServerSide* are higher than 1 and increase with the system utilization. The results imply that *ClientSide* and *ServerSide* incur much more overloaded nodes due to neglect of node available capacity. In EAD, a node sends a replication request only when it has sufficient available capacity for a replica. Thus, EAD can keep all nodes lightly loaded with consideration of node available capacity.



(a) Replica hit rate



(b) Number of replicas

Figure 3. Effectiveness and overhead of file replication algorithms.

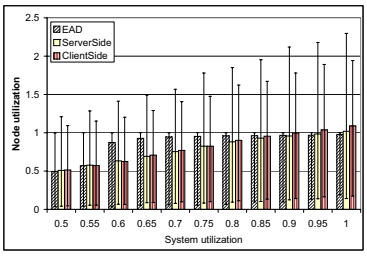
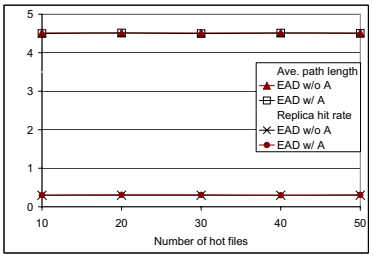
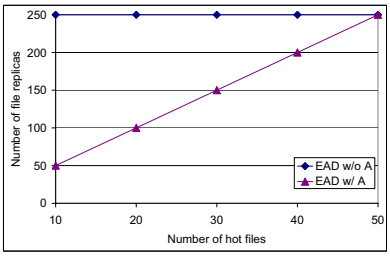


Figure 4. Node utilization.



(a) Ave. path length & replica hit rate



(b) Number of replicas

Figure 5. Effectiveness of adaptiveness in EAD file replication algorithm.

4.3 Effectiveness of Decentralized Adaptation

Figure 5 shows the effectiveness of decentralized replica adaptation strategy in EAD. We use *EADw/A* and *EADw/oA* to denote EAD with and without this strategy respectively. In this experiment, the number of hot files are ranged from 50 to 10 with 10 decrease in each step. Figure 5 (a) illustrates that *EADw/A* and *EADw/oA* can achieve almost the same average path length and replica hit rate. Figure 5 (b) shows that the number of replicas of *EADw/A* decreases as the number of hot files decreases, while that of *EADw/oA* keeps constant. *EADw/A* adjusts the number of file replicas adaptively based on the file query rate, such that less popular or requested files have less file replicas and vice versa. The results imply that *EADw/A* performs as well as *EADw/oA* with regards to lookup efficiency and replica hit rate, but it reduces unnecessary replicas and creates replicas for hot files corresponding to query rate in order to keep replicas worthwhile. Thus, *EADw/A* guarantees high replica utilization while saves overhead for maintaining replicas of cold files.

4.4 Performance in Churn

We evaluated the efficiency of the file replication algorithms in Chord P2P system with churn. Experiment results verified the resilience of the EAD algorithm in churn. We run each trial of the simulation for $20\bar{T}$ simulated seconds, where $\bar{T}$ is a parameterized time period, which was set to 60 seconds. Node joins and voluntary departures are modelled by a Poisson process as in [24] with a mean rate of R, which ranges from 0.05 to 0.40. A rate of R = 0.05 corresponds to one node joining and leaving every 20 seconds on average. In Chord, each node invokes the stabilization protocol once every 30 seconds and each node's stabilization routine is at intervals that are uniformly distributed in the 30 second interval. The number of replication operations when a server is overloaded was set to 15. We specify that before a node leaves, it also transfers its replicas to its neighbors along with its files.

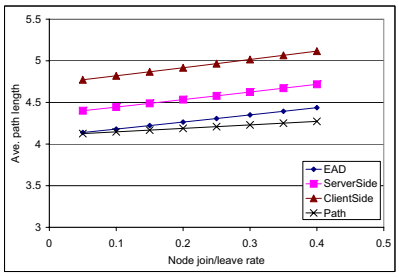


Figure 6. Ave. path length in churn.

Figure 6 plots the average lookup path length versus node join/leave rate. We can see that the results are con-

sistent with those in Figure 3 without churn due to the same reasons. We can also observe that the lookup path length increases slightly with the node join/leave rate. Before a node leaves, it transfers its replicas to its neighbors. A query for the file may pass through the neighboring replicas or other replica nodes. Otherwise, the query needs to travel to the file owner. Therefore, the path length increases marginally with the node join/leave rate. Since *Path* has much more replicas, a query has higher probability of meeting a replica node. Hence, its path length does not increase as fast as others. In summary, churn does not have significant adverse impact to file replication algorithms due to P2P self-organization mechanisms.

5 Conclusions

Traditional file replication methods for P2P file sharing systems either are not effective enough to improve file query efficiency or incur prohibitively high overhead. This paper proposes an Efficient and Adaptive Decentralized file replication algorithm (EAD) that chooses query traffic hubs and frequent requesters as replica nodes to guarantee high utilization of replicas, and high query efficiency. Unlike current methods in which file servers keep track of replicas, EAD creates and deletes file replicas by dynamically adapting to non-uniform and time-varying file popularity and node interest in a decentralized manner based on experienced query traffic. It leads to higher scalability and ensures high replica utilization. Furthermore, EAD novelly relies on exponential moving average technique to reasonably measure file query rate for replica management. Simulation results demonstrate the superiority of EAD in comparison with other file replication algorithms. It dramatically reduces the overhead of file replication and produces significant improvements in lookup efficiency. In addition, it is resilient to P2P churn. Its low overhead and high effectiveness are particularly attractive to the deployment of large-scale P2P file sharing systems.

Acknowledgments

This research was supported in part by the Acxiom Corporation.

References

[1] I. Clarke, O. Sandberg, and et al. Freenet: A Distributed Anonymous Information Storage and Retrieval System. In *Proc. of the International Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2001.

[2] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proc. of ACM SIGCOMM*, 2002.

[3] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *Proc. of IPTPS*, 2002.

[4] F. Dabek, M. F. Kaashoek, D. Karger, and et al. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.

[5] Fasttrack product description, 2001. http://www.fasttrack.nu/index_int.html.

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

[7] P. Godfrey and I. Stoica. Heterogeneity and Load Balance in Distributed Hash Tables. In *Proc. of INFOCOM*, 2005.

[8] V. Gopalakrishnan, B. Silaghi, and et al. Adaptive Replication in Peer-to-Peer Systems. In *Proc. of ICDCS*, 2004.

[9] Kazaa, 2001. Kazaa home page: www.kazaa.com.

[10] Y. lun Chou. *Statistical Analysis*. Holt International, 1975. ISBN 0030894220.

[11] Morpheus home page. http://www.musiccity.com.

[12] M. Rabinovich and A. Aggarwal. RaDaR: a scalable architecture for a global Web hosting service. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 1999.

[13] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM*, pages 329–350, 2001.

[14] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn in a DHT. In *Proc. of the USENIX Annual Technical Conference*, 2004.

[15] P. Rodriguez and S. Sibal. SPREAD: scalable platform for reliable and efficient automated distribution. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 2000.

[16] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer to Peer Networks. In *Proc. of USENIX*, 2003.

[17] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of Middleware*, pages 329–350, 2001.

[18] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-scale, Persistent Peer-to-Peer Storage Utility. In *Proc. of SOSP*, 2001.

[19] D. Rubenstein and S. Sahu. Can Unstructured P2P Protocols Survive Flash Crowds? *IEEE/ACM Trans. on Networking*, (3), 2005.

[20] S. Saroiu and et al. A Measurement Study of Peer-to-Peer File Sharing Systems. In *Pro. of MMCN*, 2002.

[21] H. Shen and C. Xu. Elastic Routing Table With Provable Performance for Congestion Control in DHT Networks. In *Proc. of ICDCS*, 2006.

[22] H. Shen and C.-Z. Xu. Hash-based Proximity Clustering for Efficient Load Balancing in Heterogeneous DHT Networks. *JPDC*, 2008.

[23] T. Stading and et al. Peer-to-peer Caching Schemes to Address Flash Crowds. In *Proc. of IPTPS*, 2002.

[24] I. Stoica, R. Morris, D. Liben-Nowell, and et al. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *TON*, 1(1):17–32, 2003.

[25] S. Tewari and L. Kleinrock. Analysis of Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of ACM SIGMETRICS*, 2005.

[26] S. Tewari and L. Kleinrock. Proportional Replication in Peer-to-Peer Network. In *Proc. of INFOCOM*, 2006.

[27] M. Theimer and M. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proc. of ICDCS*, 2002.

[28] L. Yin and G. Cao. DUP: Dynamic-tree Based Update Propagation in Peer-to-Peer Networks. In *Proc. of ICDE*, 2005.

[29] B. Y. Zhao, L. Huang, and et al. Tapestry: An Infrastructure for Fault-tolerant wide-area location and routing. *IEEE Journal on Selected Areas in Communications*, 12(1):41–53, 2004.