This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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

Haiying Shen
Department of Electrical and Computer Engineering
Clemson University, Clemson SC 29634
shenh@clemson.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. File replication in server side enhances replica hit rate hence lookup efficiency but produces overloaded nodes and cannot significantly reduce query path length. File replication in client side could greatly reduce query path length, but cannot guarantee high replica hit rate to fully utilize replicas. Though replication along query path solves these problems, it comes at a high cost of overhead due to more replicas and produces under-utilized replicas. 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. Theoretical analysis shows the high performance of EAD. 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.

Keywords: Peer-to-peer system, Distributed hash table, File sharing system, File replication.

1 INTRODUCTION

The immense popularity of Internet and P2P networks has produced a significant stimulus to P2P file sharing systems, where a file requester's query is forwarded to a file provider in a distributed manner. The systems can be used in video-on-demand service and shared digital library applications where individuals dedicate files which are available to others. A recent large scale characterization of HTTP traffic [1] 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 respond to the requests quickly. Therefore, highly-popular files (i.e., hot files) could exhaust the bandwidth capacity of the servers, leading to low efficiency in 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 higher effective file replication method produces higher replica hit rate. 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.

Recently, numerous file replication methods have been proposed. The methods can be generally classified into three categories denoted by *ServerSide*, *ClientSide* and *Path*. *ServerSide* replicates a file close to the file owner [2, 3, 4, 5], *ClientSide* replicates a file close to or at a file requester [6, 7], and *Path* replicates on the nodes along the query path from a requester to a file owner [8, 9, 10]. 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 file 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 files, 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 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

Digital Object Indentifier 10.1109/TPDS.2009.127 1045-9219/09$26.00 © 2009 IEEE
Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 10:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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 but more maintenance overhead, 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). One novel feature of EAD is that it achieves high query efficiency and high replica utilization at a significantly low cost. 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. It also produces higher hit rate than *ClientSide*, and dramatically reduces lookup path length and avoids overloading replica nodes in *ServerSide*. Moreover, EAD takes full advantage of file replicas by dynamically choosing replica nodes based on file query rate.

Another novel feature of EAD is that it adaptively adjusts the file replicas to non-uniform and time-varying file popularity and node interest in a decentralized manner. Unlike other algorithms in which a file owner determines where to create or delete replicas in a centralized fashion, EAD enables nodes themselves to decide whether to store or delete replicas based on their actual query traffic. This self-adaptive manner enhances EAD's scalability and meanwhile guarantees high utilization of replicas. It also facilitates EAD to deal with churn. In addition, EAD is highly capable of tackling skewed lookups. Furthermore, EAD employs an exponential moving average technique to reasonably measure file query traffic. The contribution of this paper also includes theoretical analysis and comprehensive simulations for the performance of EAD.

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

2 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 a detail.

2.1 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. By replicating a hot file to a number of other nodes, the file owner distributes load over replica nodes, leading to quick file response. Moreover, a file query may encounter replica nodes before it arrives at the file owner, reducing lookup path length. Thus, file replication helps achieve high file query efficiency due to lookup path length reduction and quick query response.

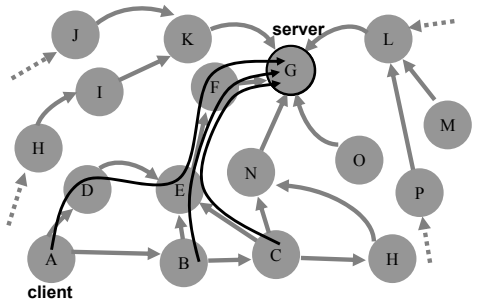


Fig. 1. File querying in a file sharing system.

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 reduce the lookup path length and may overload the neighbors. On the other hand, *ClientSide* replicates a file to requesters A, B and C. It 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, the replicas may not be fully utilized. *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 high query efficiency and low file replication overhead. Specifically, EAD aims to overcome the drawback of the previous 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 versa. How can a replication algorithm reduce 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. Since 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 replica creation and deletion.

Splitting a large file into small pieces can increase the service capacity of a large file rapidly. Replicating file location hint along a 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.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

2.2 Algorithm Description

The basic idea of EAD is replicating a file in nodes with high query traffic of the file, so that more queries will encounter the replica nodes, 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 present EAD from the following 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? (Section 2.2.1)
(2) how to conduct the replication of hot files and the deletion of under-utilized replicas in a decentralized manner for high replica utilization? (Section 2.2.2)
(3) how to reasonably measure file query traffic for replica adjustment? (Section 2.2.3)
(4) how is the performance of EAD from the perspective of theoretical analysis? (Section 2.2.4)
(5) how to deal with P2P churn for highly efficient and effective file replication? (Section 2.2.5)
(6) how is the performance of EAD in handling skewed file lookups? (Section 2.2.6)

2.2.1 Efficient File Replication

In a structured P2P system, the query load is distributed in an imbalanced manner. The existence of query imbalance is confirmed by recent studies of P2P file sharing systems [11, 1], which demonstrate that node query patterns are heavily skewed in the systems. The query load imbalance in a structured P2P system is mainly caused by three reasons. First, file requests are routed according to a strictly defined routing algorithm, and nodes are located in different places and have different number of neighbors in a P2P overlay network. Second, node interests are different and time-varying. There will be more query traffic along the query paths from the frequent file requesters and the file owner. Third, file popularity is non-uniform and time-varying. Nodes receiving and forwarding hot file queries experience more query traffic load. Nodes in some overlay areas with hot files or with more neighbors will experience more query traffic. It is easy to understand the last two reasons for query imbalance. The explanation for the first reason is presented below.

Most of the structured P2P systems such as Chord, Tapestry and Pastry use a variant of the routing algorithm developed by Plaxton *et al.* [12]. The routing algorithm works by correcting a single digit at a time in the left-to-right order. If a node with nodeID 12345 receives a lookup query with key 12456, which matches the first two digits, then the routing algorithm forwards the query to a node which matches the first three digits (e.g., node 12467). Therefore, each routing step reduces the query's distance to the destination. To facilitate the routing, the neighbors in a node's routing table are the nodes that match each prefix of its own identifier.

To handle P2P churn where nodes join and leave the system continuously and rapidly and meanwhile improve the efficiency of routing, a node with nodeID x maintains another neighbor list of nodes whose nodeIDs succeed x. Facilitated by a neighbor list, in a routing, the file query for a key is forwarded to a node whose nodeID shares a prefix with the key as long as the current node, but is numerically closer to the key than the present node's nodeID. Assume the source node ID is 12345 and the destination's nodeID is 13579, Figure 2 shows a routing path from the source node to the destination node. In the first three steps of the routing, each node forwards the query to one of its neighbors in its routing table. In the last two steps when the query is close to the destination, the routing node forwards the query to a node in its neighbor list.

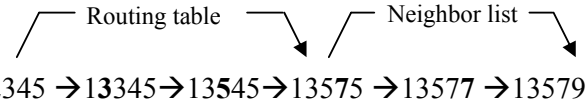


Fig. 2. An example of a query routing in structured P2P system.

We can see that the routing algorithm is characterized by convergence, in which a query travels towards its destination. Thus, queries for the same file from different directions converge when they are approaching their destination. The traffic hubs have more query load than other nodes. A node which is a neighbor of many nodes carries much more query load than others, since the nodes forward the query to the neighbor. Therefore, in structured P2P file sharing systems, some nodes carry more query traffic load than others [13, 14, 15]. The degree of node A is the number of nodes that take node A as their neighbor. Since a node owning larger nodeID space has higher probability to be other nodes' neighbor and has higher degree, the node will also carry more query load. This is confirmed by Godfrey and Stoica [13].

For example, 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 always serve as query traffic hubs. The answer for this question is given by a study which shows that the access to P2P files is highly repetitive and skewed towards the most popular ones [1].

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 a file in the middle of a query path rather than near its server as in *ServerSide* speeds up file querying.

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 a unit time period T, say one second. q_f should be indexed by different files such as q_{f_1} and q_{f_2}, but we omit the indices here for brevity. A technique for reasonably determining q_f will be introduced in Section 2.2.3. EAD sets a threshold for query rate denoted by T_q; $T_q = \alpha\bar{q}$, where

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

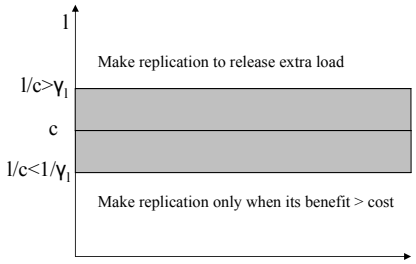


Fig. 3. File replication decision by a file server.

$\alpha(\alpha \geq 2)$ is a constant parameter, and $\bar{q}$ is the average query rate in the system.

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

where $\acute{m}$ 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 and bandwidth for a file replica, it piggybacks a file replication request and its q_f into a file query when initiating or forwarding a file request for this file. A traffic hub also needs to incorporate its IP address and nodeID into the file query. Piggybacking replication requests in 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 define *visit rate* of a node as the number of queries the node receives during T. We use a server's visit rate of all its files 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 periodically measures its query load l_i over T, and checks whether it is overloaded or lightly loaded by a factor of γ_l; *i.e.* whether

$$l_i/c_i > \gamma_l \text{ or } < 1/\gamma_l \qquad (2),$$

as shown in Figure 3. 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, the server makes decision of file replication based on the benefits and cost brought about by the file replication.

When overloaded, a file's server releases its load by selecting the nodes with high query rates to be the replica nodes. Specifically, if the server receives queries with replication requests, it 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 benefits 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 [13]. If a file is replicated in a requester with q_f and d hop distance to the file server, it saves 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 in one step. On the other hand, it costs extra storage resource b for a replica. If the benefit of the file replication is greater than its cost; that is,

$$q_f \times d \times l_q > b, \qquad (3)$$

then the file server makes a replication in the requester.

We assume homogeneous benefits for each replica of a file. If a replica is visited during T, the system earns a benefit. An unvisited replica generates system cost b. When a file server receives R requests, how many replicas it should create in order to earn the maximum average benefit during T? We use x to denote the number of created replicas that leads to the maximum average benefit. x is a value $\in [1, R]$. We use Y to denote the benefit earned by the system during T due to the file replication. Y is a function of V, which is the server visit rate of the file during T.

$$Y = f(V) = \begin{cases} ax & when\ V \geq x, \\ aV - b(x - V) & when\ V < x. \end{cases} \qquad (4)$$

Hence, the average benefit is:

$$\begin{aligned} E(Y) = E[f(V)] &= \sum_{k=1}^{R} f(k) \cdot \frac{1}{R} \\ &= \frac{1}{R}\{\sum_{k=1}^{x-1}[ak - b(x-k)] + \sum_{k=x}^{R} ax\} \\ &= \frac{1}{R}[\int_{1}^{x-1} ak - b(x-k) + \int_{x}^{R} ax] \\ &= \frac{1}{R}(-\frac{a+b}{2})x^2 + [(R-1)a + b]x. \end{aligned}$$

Thus, $$\frac{d(E(Y))}{d(x)} = (-\frac{a+b}{2})x + [(R-1)a + b].$$

Because $$\frac{d^2(E(Y))}{d(x)} = (-\frac{a+b}{2}) < 0,$$

$E(Y)$ achieves the maximum value. Therefore, when

$$x = \frac{2[(R-1)a + b]}{a+b}, \qquad (5)$$

$E(Y)$ achieves the maximum value. In other words, when the file server creates $\frac{2[(R-1)a+b]}{a+b}$ number of replicas, the system earns the maximum benefits with high replica utilization.

2.2.2 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 removes and creates file replicas.

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

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

in a decentralized manner. EAD enables nodes themselves to determine whether they should create replicas or delete replicas based on their actual experienced query traffic. If a node has too high query traffic of a file, it requests to be a replica node of the file. On the other hand, if a replica node receives too few queries of a replica, it removes the replica. Such decentralized adaptation helps 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 arranges each node to 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 $q_f < \delta T_q$ $(\delta < 1)$, where δ is a under-loaded factor, it marks the replica as infrequently-used replica. When the $q_f < \delta T_q$ condition continually occurs for a specified number of time periods, or when the node needs more space for other replicas, the node removes the replica. If this condition does not happen for the specified number of time periods which means the replica is still useful, then the replica node removes the mark. 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.

We use N_Q to denote the number of generated queries of a certain file during T, and use $\hat{q}_i$ to denote the probability that a query of the file passes node i during T. The probability that node i creates a new replica of the file is:

$$P_{creation} = \sum_{k=T_q}^{N_Q} \binom{N_Q}{k} \hat{q}_i^k (1-\hat{q}_i)^{N_Q-k}. \qquad (6)$$

The probability that node i deletes a replica is:

$$P_{delection} = 1 - \sum_{k=\delta T_q}^{N_Q} \binom{N_Q}{k} \hat{q}_i^k (1-\hat{q}_i)^{N_Q-k}. \qquad (7)$$

Formula (6) and Formula (7) show that $P_{creation}$ increases and $P_{delection}$ decreases as N_Q increases, and $P_{creation}$ decreases and $P_{delection}$ increases as N_Q decreases. It means that 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.

EAD arranges each node to keep track of its query rate of a file for both replica creation and deletion. In *ClientSide* and *ServerSide*, each file owner keeps track of every node's query rate for replica creation. In addition, each replica node also needs to keep track of its query rate of a file for replica deletion. Therefore, EAD produces less cost than *ClientSide* and *ServerSide* for replica creation and deletion.

2.2.3 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 be cold. In this case, based on the file replication algorithm introduced earlier, 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) [16] to reasonably determine file query rate over time period T.

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

```
//Executed by a file requester
Periodically calculate q_{f_t} by q_{f_t} = βq_{f_{t-1}} + (1 − β)q_{f_t}
if q_{f_t} > αT_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} = βq_{f_{t-1}} + (1 − β)q_{f_t}
if q_{f_t} > 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 Σ q_{f_t} < (l_i − γ_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} × d × 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} = βq_{f_{t-1}} + (1 − β)q_{f_t}
  if q_{f_t} ≤ δT_q do
    Remove the file replica}
```

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 $\beta \in [0, 1]$, which serves as the fading mechanism.

The value of q_f at time period $t-1$ is designated by $q_{f_{t-1}}$, and the value of the query rate at any time period T is designated by q_{f_t}. The formula for calculating q_{f_t} at time periods $T \geq 2$ is

$$q_{f_t} = \beta q_{f_{t-1}} + (1-\beta) q_{f_t} \qquad (8).$$

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

Smaller β makes the new observations relatively more important than larger β, since a higher β discounts older observations faster. An appropriate value of β can be determined according to the actual situation of a file sharing system. If the query rate fluctuates greatly, it is better to use a large β. Otherwise, small β can be used. Based on the query rate determination algorithm, node i observes the number of queries for file f periodically, and computes q_f using the EMA Formula (8). EMA-base query rate calculation helps to reasonably measure query traffic, which is critical to EAD's effectiveness. Algorithm 1 shows the pseudocode of EAD file replication algorithm integrating its different strategy components.

2.2.4 Performance Analysis

We first analyze the effect of EAD by replicating files in traffic hubs, which sheds additional insight into the performance of EAD. Let p_j denote the request probability of the jth request for file f from a random requester, and $J = \sum q_{f_j}$ denote the number of all requests for file f in the system in time period T. We use P_{hit} to denote the hit probability, u_i to denote the probability that routing node i has a replica, and o_i to denote the probability that node i is online. We use x_i to indicate whether node i contains a replica of file f. If yes, $x_i = 1$; otherwise $x_i = 0$. We use d_j to denote the path length of request j.

The hit probability of the J requests is:

$$P_{hit} = 1 - \sum_{j=1}^{J} p_j \prod_{i=1}^{d_j} [(1-o_i)(1-u_i)]^{x_i} \qquad (10).$$

We consider a special case of the hit probability of one request. In this case, each node is online with the same probability, each node in the lookup path has the same probability of having a replica, and each request has a lookup path length with the same length. That is, $o_i = o$, $u_i = u$ and $d_j = d$. We use ϕ_j to represent a group of routing nodes for the jth request. We use $L\{(\phi_j \cap (\phi_1, \phi_{j-1}, \phi_{j+1} \dots \phi_J)) \geq T_q\}$ to denote the number of routing nodes in group j that appear in the rest $J-1$ groups for at least $T_q - 1$ times. Recall that when a node's query rate exceeds query rate threshold, i.e., $q_f > T_q$, the node requests to become a replica node. Therefore, $L\{(\phi_j \cap (\phi_1, \phi_{j-1}, \phi_{j+1} \dots \phi_J)) \geq T_q\}$ means the number of replica nodes in the lookup path of the jth request if all requests are approved.

In the case of homogeneous node online probabilities and routing node having replica probabilities, and the same lookup path length, the problem of how files should be optimally replicated is to choose T_q such that the following P_{hit} is maximized.

$$P_{hit} = 1 - [(1-o)(1-u)]^d \Rightarrow 1 - P_{hit} = [(1-o)(1-u)]^d \quad (11).$$

subject to

$$L\{(\phi_j \cap (\phi_1, \phi_{j-1}, \phi_{j+1} \dots \phi_J)) \geq T_q\} = ud \qquad (12).$$

For instance, assume $o = 90\%$, $d = 10$. In order to achieve $P_{hit} = 90\%$, based on (11), we get:

$$0.1 = [0.1 + (1-u)]^{10} \Rightarrow u = 20\%.$$

It means 20% nodes in a routing path should be replica nodes. Therefore, based on (12), T_q should set to a value that makes $L\{(\phi_j \cap (\phi_1, \phi_{j-1}, \phi_{j+1} \dots \phi_J)) \geq T_q\} = 2$.

We assume that the number of generated queries N_Q of a file follows Poisson distribution [17]; $N_Q \backsim \mathbb{P}(np)$, where p is probability that a node generates a query for the file. We assume that a replica node only responds to one request. The problem of guaranteeing the probability of C that each query generated can be resolved by a replica node is to calculate x in the following inequation.

$$\begin{cases} P(N_Q \leq x) \geq C \\ P(N_Q \leq x-1) < C \end{cases}$$

Suppose that $n = 10000$, $p = 0.0004$ and $C = 99\%$, thus $N_Q \backsim \mathbb{P}(4)$. Replacing C in the inequation with 0.99, we derive $x = 9$. This means that 9 replicas are needed to ensure that a request is resolved by a replica node with 99% probability.

We now analyze the performance of EAD compared to *ServerSide* and *ClientSide*. We assume all replication methods have $m (m < n)$ replicas for a specific file f. The hit probability for the J requests in *ClientSide* is:

$$P_{hit} = \sum_{j=1}^{J} p_j \times \frac{m}{n} \qquad (13).$$

Without the loss of generality, we assume m replicas cover all the neighbors of the owner of file f. Thus, each request can encounter one of m replicas. Therefore, P_{hit} in *ServerSide* is:

$$P_{hit} = \sum_{j=1}^{J} p_j \times 1 = \sum_{j=1}^{J} p_j \qquad (14).$$

In EAD, a replica is created when $q_f > \alpha\bar{q}$. It means that on average, α requests can encounter the replica during T. Hence, P_{hit} in EAD is:

$$P_{hit} = \sum_{j=1}^{J} p_j \times \frac{m}{n/\alpha} = \sum_{j=1}^{J} p_j \times \frac{\alpha m}{n} \qquad (15).$$

The results show that when $\alpha = \frac{n}{m}$, EAD achieves the same hit rate as *ServerSide*. Since the results of (14) and (15) are higher than the result of (13) respectively, we can arrive at Proposition 2.1.

Proposition 2.1: Given the same number of replicas for a file, EAD and *ServerSide* lead to higher hit rate for the file than *ClientSide*.

In *ClientSide*, when a request hits a replica, the path length of the request is 0 because the replica is in the requester. Otherwise, the path length of the request is $\log n$ on the average case. Therefore, the total path lengths for J requests, denoted by L_J, is:

$$L_J = \sum_{j=1}^{J} p_j (1 - \frac{m}{n}) \times \log n \qquad (16).$$

Since replica nodes are the neighbors of file owner in *ServerSide*, the path length of a lookup is approximately $\log n$. Then,

$$L_J = \sum_{j=1}^{J} p_j \times \log n \qquad (17).$$

In EAD, query traffic hubs are in the middle of a lookup

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

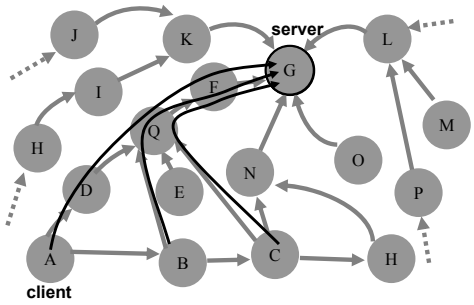


Fig. 4. Node join in a file sharing system.

path. Suppose replica nodes are located $\frac{\log n}{\omega}$ hops from the requesters on the average case. Therefore,

$$L_J = \sum_{j=1}^{J} p_j \times (1 - \frac{\alpha m}{n}) \times \log n + \sum_{j=1}^{J} p_j \times \frac{\alpha m}{n} \times \frac{\log n}{\omega}$$
$$= \sum_{j=1}^{J} p_j (1 - \frac{\alpha m}{\omega n} \log n) \quad (18).$$

We can find that the result of (17) is larger than (16) and (18). Comparing results (16) and (18), we observe that when $\alpha > \omega$, EAD produces shorter L_J than *ClientSide*. Based on the structured P2P routing algorithm, the query flows usually converge in the second half of a path. This implies that $\omega < 2$. Since $\alpha \geq 2$ and $\alpha > \omega$ in most cases, (18)<(16). Thus, we can get the following Proposition:

Proposition 2.2: On the average case, EAD leads to shorter path length than *ClientSide*, and *ClientSide* produces shorter path length than *ServerSide*.

2.2.5 Discussion of EAD in Churn

P2P systems are characterized by churn in which participating nodes continuously join and leave the network or even fail unexpectedly. EAD should be able to efficiently handle node joins, departures and failures.

Node joins. A node's successor is the node with the smallest nodeID among the nodes whose nodeIDs are larger than the node's ID. A node's predecessor is the node with the largest nodeID among the nodes whose nodeIDs are less than the node's nodeID. As shown in Figure 4, when a new node Q joins a structured P2P system shown in Figure 1, it becomes responsible for a portion of P2P global ID space based on its nodeID. The original node responsible for the files in this ID space portion, say node E, moves the files to node Q. In addition, the routing tables of nodes that take Q as a neighbor should be updated. There are mainly two policies for updating the routing tables of affected nodes. Systems like Pastry and Tapestry update affected routing tables once a new node joins in. Rather than using instant updating, systems like Chord rely on stabilization, in which each node updates its neighbors periodically. This is a basic component of P2P maintenance to guarantee the connectivity of nodes for successful routings in churn.

For EAD, the issue at hand is deciding what to do with file replicas. As indicated in Section 2.2.1, node degree influences query imbalance. The newly-joined node Q may affect the degree and hence the query load of node E. Some nodes that take node E as a neighbor and frequently forward E the query of a replica may replace node E with Q in their routing tables. Thus, if node Q takes over most of the query load of a replica from node E, node E should transfer the replica to node Q. Therefore, to determine whether to move a replica depends on how many nodes will replace node E with Q in their routing tables. For example, in Figure 4, when node Q joins in the system, nodes B, C and D replace their previous neighbor E with Q. As a result, queries for a replica from nodes B, C and D will flow to node Q. In this case, node E should transfer the replica to node Q. Thus, each node such as node E keeps track of the traffic flow coming from each of its neighbors. When a node, say node C, replaces E with Q in its routing table, it notifies node E. When node E finds that most of its neighbors that frequently forward the query for its replica replace itself with Q, it transfers the replica to node Q. However, this method has a number of drawbacks. First, keeping the record for traffic flow from each backward link is resource consuming. Second, it is difficult to decide how much of a replica's traffic is moved from node E to node Q before the replica transfer is triggered. Third, in P2P systems such as Chord in which the routing tables are not updated soon after a node join operation, E won't transfer its replica to node Q even if Q becomes a traffic hub soon after joining.

Algorithm 2: Pseudo-code for a node join in EAD algorithm.

```
//Executed by a joining node
Get its nodeID
Build its routing table and neighbor list
Notify its predecessor and successor
Receive the files in nodeID space [pre.ID, ID] and [ID, suc.ID]
Receive the indices of replicas in its predecessor and successor
Periodically compute the query rate of each of the replicas of f
if q_f > T_q then
   Request a replica from the owner of the replica

//Executed by a joining node's predecessor and successor
Get the notification from the newly-joined node
Update its predecessor or successor
if it is the predecessor then
   Transfer the files in nodeID [pre.ID, ID] to the newly-joined node
else
   Transfer the files in nodeID [ID, suc.ID] to the newly-joined node
Transfer the indices of its replicas to the newly-joined node
Periodically calculate q_f of each of its replica of f
if q_f < δT_q then
  if the newly-joined node has the replica then
     Remove the replica
  else
     Transfer the replica to the newly-joined node
```

To avoid these drawbacks, EAD depends on a less complex and light-weight method. When node E transfers its files to node Q, it also stores the indices of its replicas to node Q indicating their location. If node Q receives a query for the file of the replica, it directly forwards the query to node E based on the indices. Later on, when node Q's $q_f > T_q$, it requests to have a replica from node E. Since node Q and node E are neighbors, the communication cost generated between them is less than the cost of requesting a replica from the file owner. When the $q_f < \delta T_q$ condition continually occurs in node E for a specified number of time periods, node E deletes the

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

replica if node Q already has the replica. Otherwise, it moves the replica to node Q in case Q needs the replica. If the replica is under-utilized in node Q, Q will remove the replica. This strategy ensures that there is no unnecessary replica transfer in a node join and meanwhile the replica is fully utilized. Algorithm 2 shows the pseudo-code for a node join in the EAD file replication algorithm in a system where a file is stored in a node whose nodeID is the closest to the file's fileID and stabilization is used for updating routing tables.

Algorithm 3: Pseudo-code for node departure in EAD.

```
//Executed by a leaving node
Notify its predecessor and successor
Transfer files in [pre.ID,ID] to its predecessor
Transfer files in [ID,suc.ID] to its successor
if |suc.ID − ID| < |ID − pre.ID| then
  N=successor
else
  N=predecessor
Transfer all its replicas to N
Transfer the indices of its replicas to the other neighbor

//Executed by a leaving node's predecessor and successor
Get the leaving notification
Update its predecessor or successor
Receive the files from the leaving node
if receive the replicas then
  Periodically calculate q_f of each of its replica of f
  if q_f < δT_q then
    if its predecessor or successor has the replica then
      Remove the replica
    else
      Transfer the replica to its successor or predecessor
else
  Periodically calculate q_f of each file whose index is stored in itself
  if q_f > T_q then
    Request a replica from the indexed replica node
```

Node departures. Before a node leaves the system, it transfers its files to its successor and predecessor based on the structured P2P system's file allocation algorithm. Like the node join operation, there are two policies for routing table update for node departure. Systems like Chord depend on stabilization. In others systems, the leaving node notifies affected nodes that need to update their routing tables. Most structured systems resort to stabilization to deal with node departures without warning and node failures.

The problem that EAD needs to resolve is that the leaving node should transfer its replicas to its successor or predecessor. If there are more affected nodes originally forwarding the file query take the successor as the replacement for the leaving node, the file's replica should be transferred to the successor. Otherwise, the replica should be transferred to the predecessor.

According to the P2P neighbor selection policy, among the successor and predecessor, the one closer to the leaving node has higher probability to replace the leaving node in affected nodes' routing tables. For example, leaving node 12345's successor is 12349 and its predecessor is 12339, then its successor has higher probability to be its replacement in affected nodes' routing table. Based on this, EAD arranges the leaving node to move its replica to its successor or its predecessor that is closer to itself, and store the indices of the replicas in the other node.

Suppose leaving node Q transfers its replicas to its successor E and stores the indices of the replicas in its predecessor C. Later on, when node C receives a query for the file of one of the replicas, it directly forwards the query to its successor node E. Node C keeps track of the query rate of the replicas whose indices are stored in node C. When a file's $q_f > T_q$, node C requests a replica from node E. Since node E and node C are neighbors, the communication between these two nodes will not generate much overhead. At the same time, node E periodically checks q_f of its replicas. When $q_f < \delta T_q$ condition continually occurs for a specified number of time periods, it deletes the replica if node C already has the replica. Otherwise, it moves the replica to node C in case C needs it later. If the replica is under-utilized in node C, C removes the replica.

Node departures without warning and node failures lead to replica loss. EAD's decentralized file replica adaption algorithm helps to cope with the negative results due to replica loss. In this case, the failed node's otherwise carrying traffic are moved to some other nodes. When these nodes observe the high traffic volume of a file, they will create replicas. This algorithm plays an important role in dealing with churn. It ensures that traffic hubs have replicas for high utilization of replicas, and there is few under-utilized replicas in churn. Algorithm 3 demonstrates the pseudo-code for node departure in the EAD file replication algorithm.

2.2.6 Discussion of EAD in Skewed Lookups

In skewed lookups, many nodes visit a file repeatedly and continuously. For example, during the period of Olympic games, Olympic news becomes very popular. Queries flowing from all over the world to one file owner of Olympic news will overload the owner, leading to high query latency. In this case, much more replicas than usual are needed to release the load of the owner, generating very high overhead. *ServerSide* replicates the file at the neighbors near the owner. Thus, queries still need to travel long way before encountering a replica node. In addition, these replica nodes around the file owner can easily be overloaded under the tremendous volume of query flow. In *ClientSide*, much more replicas will be stored in many widely distributed clients all over the world. However, the replicas may not be shared by other requesters, resulting in low replica utilization and very high overhead for the maintenance of widely scattered replicas. In contrast, skewed lookup is an asset in EAD, which enables EAD to exploit its fullest capacity.

In skewed lookups, a significant amount of queries flow towards the same destination. Thus, much more queries will meet with each other than the regular lookups, generating many traffic hubs. EAD replicates the file in traffic hubs. Therefore, a high volume of query flow is resolved in the middle of routes, reducing the query latency and meanwhile enhancing the utilization of replicas. Therefore, skewed lookups increase the number of traffic hubs, facilitating EAD to find the traffic hubs for replications. In addition, it increases the probability that a file query is resolved by a traffic hub, raising

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

TABLE 1
Simulated environment and parameters.

Parameter	Default value
File distributon	Uniform over nodeID 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

the replica utilization. Unlike *ServerSide* and *ClientSide* that cannot handle skewed lookups effectively and efficiently, EAD exhibits its fullest advantages in skewed lookups.

To make *ServerSide*, *ClientSide* and EAD comparable, we suppose *ServerSide* replicates a file in the file's neighbor, and *ClientSide* replicates a file in a requester if the requester's $q_f \geq \alpha\bar{q}$. We assume in skewed lookups, $J(J \to \infty)$ requests for a file are generated by random nodes in time period T. We now analyze the probability that random node i becomes a replica node in the skewed lookups.

Recall that $\hat{q}_i$ denotes the probability that a request passes the random node i. Thus, the probability that k requests pass node i is:

$$P(X=k) = \lim_{J\to\infty} \binom{J}{k} \hat{q}_i^k (1-\hat{q}_i)^{J-k} = e^{-\lambda} \cdot \frac{\lambda^k}{k!},$$

where $\lambda = J\hat{q}_i$. That is, $X \sim \mathbb{P}(\lambda)$. Hence,

$$E(X) = \lambda = J\hat{q}_i \qquad (19).$$

Thus, the probability that a random node becomes a replica node is:

$$P_r = 1 - P(X \leq T_q) = 1 - \sum_{k=1}^{\alpha} e^{-\lambda} \cdot \frac{\lambda^k}{k!} \qquad (20).$$

We use ^{C}, ^{S} and ^{E} to represent *ClientSide*, *ServerSide* and EAD respectively. Considering the locations of replica nodes in three methods, we get $\hat{q}_i^C < \hat{q}_i^E < \hat{q}_i^S$. Based on Equation (19), we can get $E(X)^C < E(X)^E < E(X)^S$. Therefore, we can arrive at the following proposition.

Proposition 2.3: In skewed lookups, on the average case, file owner neighbors in *ServerSide* have higher probability to become replica nodes than the routing nodes in EAD, which have higher probability to become replica nodes than requesters in *ClientSide*.

We use N_{hit} to denote the number of replica hits for another group J requests for the file in skewed lookups. Then, $N_{hit} = JP_r$. On the average case, $N_{hit}^C < N_{hit}^E < N_{hit}^S$. Hence, we can get the following proposition:

Proposition 2.4: In skewed lookups, the utilization of replicas in *ClientSide* is not as high as in EAD, and that in EAD is not as high as in *ServerSide*.

3 Performance Evaluation

We designed and implemented a simulator for evaluating the EAD algorithm based on Chord P2P system [17]. We use

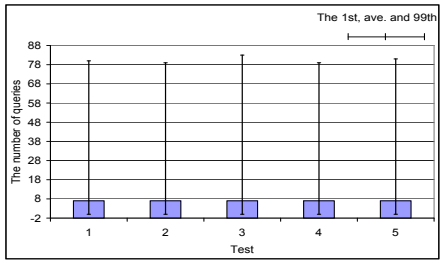


Fig. 5. Query load imbalance.

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 file query efficiency, high hit rate, and balanced load distribution with less file replicas. Moreover, EAD is resilient to P2P churn and skewed lookups. 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*, a frequent requester in *ClientSide*, and 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 [17]. Table 1 lists the parameters of the simulation and their default values.

3.1 Query Load Imbalance

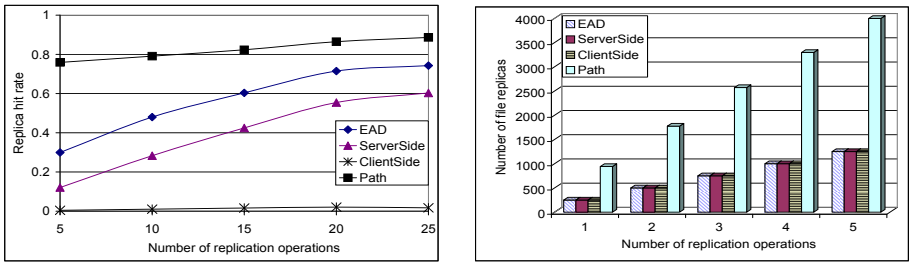
This experiment is conducted in order to verify the fundamental basis that there exists traffic hubs based on which EAD is developed. In the experiment, 5000 randomly selected nodes query for the same file that is randomly chosen. We conducted five tests, and recorded the average, the 1st and 99th percentiles of the number of requests a node received. Figure 5 plots the results of the five tests. The figure shows that the 99th percentile keeps around 80, the 1st percentile keeps at 0, and the average is around 7. The results imply that the query load varies among nodes, and some nodes have much heavier query load than others. These highly-loaded nodes are traffic hubs, where requests for a file meet together. The experiment results confirm the existence of traffic hubs, which is the cornerstone of the EAD algorithm. EAD takes advantage of these traffic hubs to achieve efficient and effective file replication with low overhead.

3.2 Effectiveness of Replication Algorithms

Figure 6(a) demonstrates the replica hit rate of different algorithms versus the number of replication operations when a

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.



(a) Replica hit rate (b) Number of replicas

Fig. 6. Effectiveness and overhead of file replication algorithms.

server is overloaded. 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. *ClientSide* replicates a file in the file requesters, which may not request the same file later. Also, other file requests have very low possibility of passing through these replica nodes. Consequently, *ClientSide* 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. This implies the effectiveness of EAD to replicate files in nodes with high query rate, which enhances the utilization of replicas and hence reduces the lookup path length. *Path* replicates files at nodes along a routing path. More replica nodes render higher possibility for a file request to meet a replica node. Therefore, *Path* increases replica hit rate and produces shorter path length. However, its efficiency is outweighed by its prohibitive cost of overhead for keeping track of query paths and maintaining much more file replicas.

3.3 Overhead of Replication Algorithms

Figure 6(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. 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* needs much higher overhead for file replication and replica maintenance.

In conclusion, *Path* has high hit rate and short lookup path length, but this benefit comes at the cost of prohibitively higher overhead. *ServerSide* and *ClientSide* incur less overhead for file replicas, but are relatively less efficient in lookups. EAD can achieve approximately the same lookup efficiency at a significantly lower cost.

3.4 Load Balance of Replication Algorithms

This experiment demonstrates the load balance among replica nodes in each replication method. Recall that *ClientSide* and *ServerSide* don't take into account node available capacity, while EAD proactively takes into account node available capacity during file replication. It avoids exacerbating overloaded node problem by choosing nodes with enough available capacity as replica nodes. Thus, it outperforms *ClientSide* and *ServerSide* by controlling the overloaded nodes and hence extra overhead for load balancing or further file replication. We measured the maximum node utilizations of all nodes and took the 1st, 99th percentiles and median of those results as experiment results. Figure 7 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 system utilization. The results imply that *ClientSide* and *ServerSide* incur much more overloaded nodes due to the 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 the consideration of node available capacity.

3.5 Effectiveness of Decentralized Adaptation

Figure 8 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 is ranged from 50 to 10 with 10 decrease in each step. Figure 8 (a) illustrates that *EADw/A* and *EADw/oA* can achieve almost the same average path length and replica hit rate. Figure 8 (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*

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

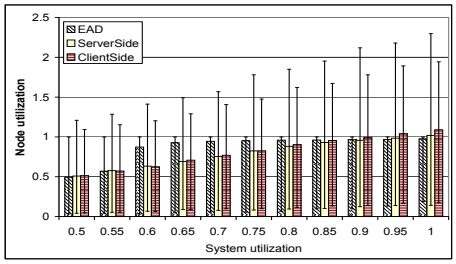
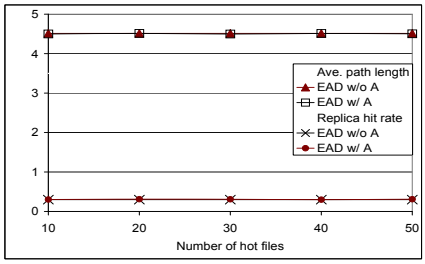
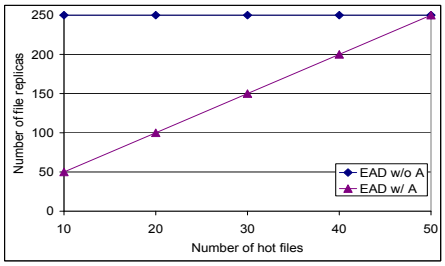


Fig. 7. Node utilization.



(a) Ave. path length & replica hit rate



(b) Number of replicas

Fig. 8. Effectiveness of adaptiveness in EAD file replication algorithm.

guarantees high replica utilization while saves overhead for maintaining replicas of cold files.

3.6 Performance in Skewed Lookups

We tested the file replication algorithms in skewed lookups. In the experiment, only the files whose fileID is between [0,99] are queried. The fileIDs of the queried files are randomly generated. Figure 9 shows the performance of different replication methods in skewed lookups. Figure 9(a) plots the average path length of different algorithms. It demonstrates that *Path* generates the shortest lookup path length, and EAD leads to marginally longer lookup path length than *Path*. *ServerSide*'s path length is longer than EAD, and *ClientSide*'s path length is longer than *ServerSide*. Since the file destinations are gathered together in a small nodeID space interval, the file requests from different directions are very likely to take similar routes and the probability that two requests meet together is very high. Therefore, by replicating a file in the nodes along the route, *Path* significantly reduces the path length. EAD replicates a file in nodes with high query load. It also reduces path length since a request meets the replica node with very high probability in skewed lookups. Because it has less replicas than *Path*, its path length is slightly longer than *Path*. *ServerSide* replicates a file around its server. Thus, a file request still needs to travel until it is close to the server before it encounters a replica node, which results in longer path length. Replicating a file in a client doesn't enable other requesters to share the replica. The requests from other requesters still need to be forwarded hop by hop to the file server, leading to longer path length.

Figure 9(b) shows the replica hit rate of each replication algorithm. *Path* generates the highest hit rate followed by EAD. *ServerSide* produces lower hit rate than EAD followed by *ClientSide*. The results are consistent with the lookup path results in Figure 9(a). Higher rate leads to shorter lookup path length and vice versa. The reasons for the results are due to the same reasons observed in Figure 9(a). Figure 9(c) demonstrates the number of file replicas in each replication algorithm. We can observe that *Path* produces much more replicas than others. It is because *Path* replicates a file in all node in a route and the EAD, *ServerSide* and *ClientSide* algorithms only replicate a file in one node. The results imply that *Path* achieves shorter path length and high hit rate at the cost of dramatically more replicas. EAD's path length is slightly longer than *Path* and much shorter than *ServerSide* and *ClientSide*, but its cost is almost the same as them.

3.7 Performance in Churn

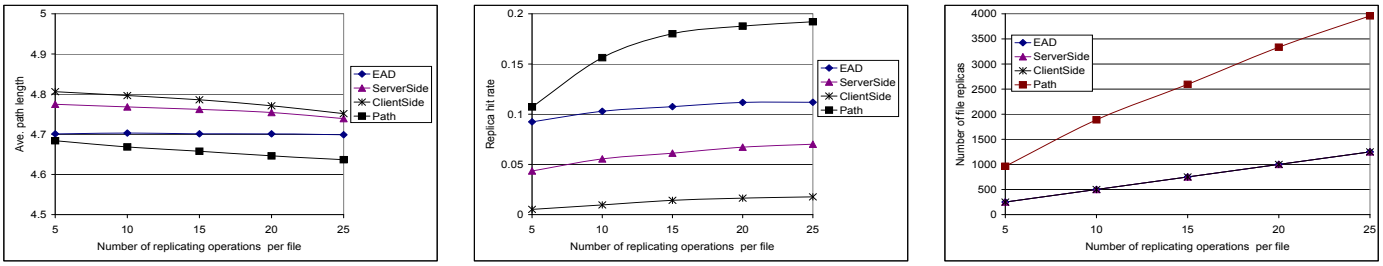
We evaluated the efficiency of the file replication algorithms in Chord P2P system with churn. We run each trial of the simulation for $20\bar{T}$ simulated seconds, where $\bar{T}$ is a parameterized time period, which was set to 60 seconds. Node joins and voluntary departures are modelled by a Poisson process as in [17] with a mean rate, which ranges from 0.05 to 0.40. A rate of 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 10(a) plots the average lookup path length versus node join/leave rate. We can see that the results are consistent with those in Figure 6 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. In addition, churn may lead to detour routing with more node hops along the routing path. 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 the file replication algorithms due to the P2P self-organization mechanisms.

The next experiment is to test the lookup latency of each replication algorithm. We assume that the request forwarding latency is 0.2 second in a lightly loaded node, and 1 second in a heavily loaded node. Figure 10(b) displays the average lookup latency versus the node join/leave rate. It can be observed that the latency increases as node join/leave rate grows due to the same reasons observed in Figure 10(a). It is intriguing to see that the latency of *Path* is longer than EAD, and that of *ServerSide* is longer than *ClientSide* since the path length of the algorithms are in the opposite case. In addition to the path length, node utilization is also an important factor affecting the lookup latency. A heavily loaded node needs more time to process and forward a request than a lightly loaded node. EAD avoids overloading nodes by replicating a file to nodes with sufficient capacity. However, *Path* just replicates a file in

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.



(a) Ave. path length (b) Replica hit rate (c) Number of replicas

Fig. 9. Performance of file replication algorithms in skewed lookups.

the nodes along a lookup path without considering the node capacity and load status, which tend to generate heavily loaded nodes. Thus, although *Path* has average less hops in a lookup, it leads to longer lookup latency due to its neglect of node load status. *ServerSide* replicates a file around a server, overloading the nodes close to the server. In contrast, *ClientSide* distributes replicas among requesters in the entire system, making it less possible to generate heavily loaded nodes. Consequently, although *ServerSide* has shorter path length than *ClientSide*, its average lookup latency is longer than *ClientSide*. The results demonstrate that an effective file replication algorithm should reduce lookup path length and meanwhile avoid generating heavily loaded nodes.

4 Related Work

Numerous methods have been proposed for file replication in P2P systems. As mentioned, most current file replication methods generally can be classified into three categories: *ServerSide*, *ClientSide* and *Path*. Some proposed approaches use a combination of the methods.

ServerSide category includes PAST [2], CFS [3], Backslash [4] and Overlook [5]. PAST is an Internet based global P2P storage utility with a storage management and caching system. It replicates each file on a set number of nodes whose nodeIDs match most closely to the file owner's nodeID. The number of replicas is chosen to meet the availability needs of a file, relative to the expected failure rates of individual nodes. The work also proposed file diversion method in which a file is diverted to a different part of ID space by choosing a different fileID when a file insert operation fails. PAST uses file caching along the lookup path to minimize query latency and balance query load. Cooperative File System (CFS) [3] is a P2P read-only storage system for file storage and retrieval. CFS is built on Chord [17] and replicates blocks of a file on nodes immediately after the block's owner on the Chord ring. CFS also caches a file location hint along a path to improve query efficiency and avoid overloading servers that hold popular data. Stading *et al.* [4] proposed Backslash, which is built on a structured P2P overlay and caches aggressively a file that experiences a high request load. Each Backslash node has available storage splitting into two categories: replica space and temporary cache space. Replicas are placed in the overlay by insertion operations of the distributed hash table (DHT). A temporary cache is a cached copy of a document that is placed opportunistically at a node. Overlook [5] is built on Pastry [18]. It places a replica of a file on a node with most incoming lookup requests for fast replica location. Overlook does this by selecting the incoming forwarding node with the highest recorded request rate and sending that node a create-replica request message.

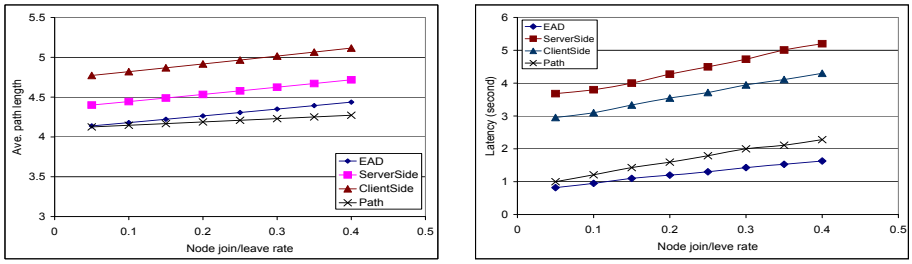
In the *ClientSide* category, Gnutella [7] replicates files in overloaded nodes at the file requesters. Nodes store and serve only what they requested, and a replica can be found only by probing a node with a replica. FarSite [19, 20, 21] is a traditional file system with high reliability and availability. It replicates the same number (e.g., 3 or 4) of a file on the client side to enhance file availability. LAR [6] is a lightweight, adaptive and system-neutral replication protocol. It specifies the overloaded degree of a server that a file should be replicated, and replicates a file to a client. In addition to replicating a file at the requester, LAR also replicates file location hints along the lookup path. Once a replica is created, LAR installs cache state on the path from the new replica to the node that created the replica.

In the *Path* category, Freenet [22] replicates files on the path from the requester to the target. Freenet's request mechanism will cause popular data to be transparently replicated by the system and mirrored closer to requestors. In addition to PAST [2], CFS [3] and LAR [6], CUP [8], DUP [9] and the work in [10] also perform caching along the query path. CUP [8] is a protocol for performing Controlled Update Propagation to maintain caches of metadata in P2P networks. The propagation is conducted by building a CUP tree similar to an application-level multicast tree. A node proactively receives updates for metadata items from a neighbor only if the node has registered interest with the neighbor. However, intermediate nodes along the path receive the updated index even if they do not need it. Dynamic-tree based Update Propagation (DUP) scheme [9] is proposed to improve CUP. DUP builds a dynamic update propagation tree on top of the existing index searching structure. Because the update propagation tree only involves nodes that are essential for update propagation, the overhead of DUP is very small and the query latency is significantly reduced. Cox *et al.* [10] studied providing DNS service over a P2P network. In order to increase the robustness as servers come and go, a Chord-based distributed hash table automatically moves data so that it is always stored on a fixed number of replicas (typically six). Specifically, the method caches index entries, which are DNS mappings, along query paths.

Ghodsi *et al.* [23] proposed a symmetric replication scheme in which a number of fileIDs are associated with each other,

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.



(a) Ave. path length (b) Lookup latency

Fig. 10. Performance of file replication algorithms in churn.

and any file with a fileID can be replicated in nodes responsible for the fileIDs in this group. HotRoD [24] is a structured P2P based architecture with a replication scheme. An arc of peers (i.e. successive peers on the ring) is "hot" when at least one of these peers is hot. In the scheme, "hot" arcs of peers are replicated and rotated over the identifier space. LessLog [25] determines the replicated nodes by constructing a lookup tree based on nodeIDs to determine the location of the replicated node. Ni *et al.* [26] introduced expected costs computed from user request rates, file storage/transfer/miss costs, and node up/down statistics. The authors then proposed file replication schemes that determine the sets of nodes to store replicas in order to minimize the expected costs. DHT-based self-adapting replication protocol [27] determines the locations of replicas by the DHT data allocation algorithm. It enables a newly-joined peer first to assume an initial value for the number of replicas, and then adjusts autonomously the number of replicas to deliver a configured data availability guarantee. The works in [28, 29, 30, 31, 32, 33] employed erasure coding technique for file replication. The technique allows nodes to generate encoded blocks of information for replication, which could enhance availability and reliability in storage and communication systems.

There are other studies for file replication in unstructured P2P systems [34, 35, 36, 37, 38, 34, 39, 40]. 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, uniform and 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.

5 Conclusions

Traditional file replication methods for P2P file sharing systems replicate files close to file owners, file requesters or query path to release the owners' load and meanwhile improve the file query efficiency. However, replicating files close to the file owner may overload the nodes in the close proximity of the owner, and cannot significantly improve query efficiency since replica nodes are close to the owners. Replicating files close to or in the file requesters only brings benefits when the requester or its nearby nodes always query for the file. In addition, due to non-uniform and time-varying file popularity and node interest variation, the replicas cannot be fully utilized and the query efficiency cannot be improved significantly. Replicating files along the query path improves the efficiency of file query, but it incurs significant 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 relies on exponential moving average technique to reasonably measure file query rate for replica management.

Theoretical study sheds insight into the efficiency and effectiveness of EAD. 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 and skewed lookups.

In the future work, we will further study the effect of churn on the efficiency and effectiveness of EAD. In a public P2P system, a node may not be willing to have replicas for others. We will study the effect of real system constraints and access right constraints for replications. We also plan to further explore adaptive methods to fully exploit file popularity and update rate for efficient replica consistency maintenance.

Acknowledgment

The authors are grateful to the anonymous reviewers for their valuable comments and suggestions. This research was supported in part by U.S. NSF grants CNS-0834592, CNS-0832109 and CNS 0917056. An early version of this work [41] was presented in the Proceedings of P2P08.

References

[1] S. Saroiu, P. Gummadi, and S. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN*, 2002.

[2] 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.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

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

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

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

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

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

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

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

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

[11] P. Gummadi, R. Dunn, S. Saroiu, S. Gribble, H. Levy, and J. Zahorjan. Measurement, Modeling, and Analysis of a Peer-to-Peer File-Sharing Workload. In *Proc. of SOSP*, 2003.

[12] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM SPAA*, 1997.

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

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

[15] Q. Lv, S. Ratnasamy, and S. Shenker. Can Heterogeneity Make Gnutella Scalable? In *Proc. of IPTPS*, 2002.

[16] Y.-L. Chou. *Statistical Analysis*. Holt International, 1975. ISBN 0030894220.

[17] 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.

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

[19] A. Adya et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of OSDI*, 2002.

[20] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of International Conference on Distributed Computing Systems*, 2002.

[21] J. R. Douceur and R. P. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *Proc. of SRDS*, pages 4–13, 2001.

[22] 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.

[23] A. Ghodsi, L. Alima, and S. Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In *Proc. of International Workshop on Databases, Information Systems and Peer-to-Peer Computing*, page 12, 2005.

[24] T. Pitoura, N. Ntarmos, and P. Triantafillou. Replication, Load Balancing and Efficient Range Query Processing in DHTs. In *Proc. of EDBT*, 2006.

[25] K. Huang and et al. LessLog: A Logless File Replication Algorithm for Peer-to-Peer Distributed Systems. In *Proc. of IPDPS*, 2004.

[26] J. Ni, J. Lin, S. J. Harrington, and N. Sharma. Designing File Replication Schemes for Peer-to-Peer File Sharing Systems. In *Proc. of IEEE ICC*, page 5609.

[27] P. Knezevič, A. Wombacher, and T. Risse. DHT-based self-adapting replication protocol for achieving high data availability. In *Proc. of SITIS*, 2008.

[28] W. K. Lin, D. M. Chiu, and Y. B. Lee. Erasure code replication revisited. In *Proc. of International Conference on Peer-to-Peer Computing*, 2004.

[29] R. Rodrigues and B. Liskov. High availability in DHTs: Erasure coding vs. replication. In *Proc. of International Workshop on Peer-to-Peer Systems*, 2005.

[30] C. Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *Proc. of INFOCOM*, 2005.

[31] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *Proc. of Symposium on Networked Systems Design and Implementation*, 2005.

[32] J. Kangasharju, K. W. Ross, and D. A. Turner. Optimizing File Availability in Peer-to-Peer Content Distribution. In *Proc. IEEE INFOCOM*, 2007.

[33] J. Kangasharju, K. W. Ross, and D. A. Turner. Adaptive content management in structured P2P communities. In *Proc. of Infoscale*, 2006.

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

[35] L. Massoulie and M. Vojnovic. Coupon Replication Systems. In *Proc. of ACM SIGMETRICS*, 2005.

[36] S. Tewari and L. Kleinrock. On Fairness, Optimal Download Performance and Proportional Replication in Peer-to-Peer Networks. In *Proc. of IFIP Networking*, 2005.

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

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

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

[40] Q. Lv and K. Li. Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of ICS*, 2002.

[41] H. Shen. EAD: An Efficient and Adaptive Decentralized File Replication Algorithm in P2P File Sharing Systems. In *Proc. of P2P*, 2008.

Authorized licensed use limited to: CLEMSON UNIVERSITY. Downloaded on January 6, 2010 at 11:22 from IEEE Xplore. Restrictions apply.