Swarm Intelligence based File Replication and Consistency Maintenance in Structured P2P File Sharing Systems

Haiying Shen*, *Senior Member, IEEE*, Guoxin Liu, *Student Member, IEEE*, Harrison Chandler

Abstract—In peer-to-peer file sharing systems, file replication helps to avoid overloading file owners and improve file query efficiency. There exists a tradeoff between minimizing the number of replicas (i.e., replication overhead) and maximizing the replica hit rate (which reduces file querying latency). More replicas lead to increased replication overhead and higher replica hit rates and vice versa. An ideal replication method should generate a low overhead burden to the system while providing low query latency to the users. However, previous replication methods either achieve high hit rates at the cost of many replicas or produce low hit rates. To reduce replicas while guaranteeing high hit rate, this paper presents SWARM, a file replication mechanism based on swarm intelligence. Recognizing the power of collective behaviors, SWARM identifies node swarms with common node interests and close proximity. Unlike most earlier methods, SWARM determines the placement of a file replica based on the accumulated query rates of nodes in a swarm rather than a single node. Replicas are shared by the nodes in a swarm, leading to fewer replicas and high querying efficiency. In addition, SWARM has a novel consistency maintenance algorithm that propagates an update message between proximity-close nodes in a tree fashion from the top to the bottom. Experimental results from the real-world PlanetLab testbed and the PeerSim simulator demonstrate the effectiveness of the SWARM mechanism in comparison with other file replication and consistency maintenance methods. SWARM can reduce querying latency by 40%-58%, reduce the number of replicas by 39%-76%, and achieves more than 84% higher hit rates compared to previous methods. It also can reduce the consistency maintenance overhead by 49%- 99% compared to previous consistency maintenance methods.

Index Terms—File replication, Consistency maintenance, Peer-to-peer, Distributed hash table, Swarm intelligence

1 Introduction

Over the past years, the immense popularity of the Internet has produced a significant stimulus to peer-to-peer (P2P) networks. One of the most popular applications of P2P networks is file sharing such as BitTorrent, Morpheus, eDonkey and Gnutella. P2P file sharing is a system of sharing files directly between network users, without the assistance or interference of a central server. The total traffic on the BitTorrent P2P file sharing application has increased by 12%, driven by 25% increases in per-peer hourly download volume [1] in 2010; during peak hours, BitTorrent accounted for more than a third of all upload traffic [2] in 2012. According to the Cisco's network traffic measurement and forecast [3], the P2P file sharing applications account for 83.5% of total file sharing traffic, and will still account for 60.3% of total file sharing traffic in 2018. In a P2P file sharing system, if a node receives a large volume of requests for a file at one time, it becomes a hot spot, leading to delayed responses.

File replication is an effective strategy to manage the problem of overload due to hot files. Distributing the load by replicating hot files to other nodes improves file query efficiency by reducing query latency. There exists a tradeoff between minimizing the number of replicas (i.e., replication overhead) and maximizing the replica hit rate (i.e., file querying latency). More replicas lead to higher replication overhead and higher replica hit rates and vice versa. An ideal replication method should generate a low overhead to the system while providing low query latency to users. However, previous file replication methods either achieve high hit rates at the cost of many replicas or produce low hit rates with fewer replicas. Specifically, previous replication methods can be organized into four classes denoted by *Random*, *ServerEnd*, *Path* and *ClientEnd*. *Random*, such as [4], replicates files to randomly selected nodes. *ServerEnd*, such as [5, 6] replicates a file into nodes close to its file owner on the P2P overlay. *Path*, such as [7–9] replicates a file along its query path. In *Random*, *ServerEnd* and *Path*, a file query still needs to travel until it encounters the file owner or a replica node. *Random* and *Path* may also lead to high overhead due to a large number of replicas, some of which may not be fully utilized. Recently, we proposed Efficient and Adaptive Decentralized file replication algorithm (*EAD*) [10], an improved *Path* method. The query traffic hubs of a file are the nodes where many query paths of this file intersect at. *EAD* selects the query traffic hubs and frequent requesters of a file as its replica nodes in order to increase hit rate while limit the number of replicas. *ClientEnd*, such as [11], replicates a file into the nodes of frequent requesters, so they can access the file directly without query routing. However, *ClientEnd* cannot ensure high hit rates since other requesters' queries have low probabilities of passing the frequent requesters. Furthermore, *ServerEnd*, *Path*, *ClientEnd* and *Random* cannot guarantee that a file query will encounter a replica node of the file.

To provide this guarantee while constraining the number of replicas (i.e., replication overhead), this paper presents SWARM, a file replication mechanism based on swarm intelligence. Swarm intelligence is the property

• * *Corresponding Author. Email: shenh@clemson.edu; Phone: (864) 656 5931; Fax: (864) 656 5910.*

• *Haiying Shen and Guoxin Liu are with the Department of Electrical and Computer Engineering, Clemson University, Clemson, SC, 29634. E-mail: {shenh, guoxinl}@clemson.edu*
Harrison Chandler is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: hchandl@umich.edu

of a system whereby the collective behaviors of agents cause coherent functional global patterns [12]. Recognizing the power of collective behaviors, SWARM identifies node swarms with common interests and close proximity. Common-interest swarms further form into a *colony*. Unlike previous methods that replicate a file according to the query rate of a single requester, SWARM replicates a file according to the accumulated query rates of nodes in a swarm and enables the replica be shared among the swarm nodes. Without spreading replicas all over the network and enabling replicas to be shared among common-interest nodes in close proximity, file replicas are significantly reduced and fully utilized while improving query efficiency. More importantly, most previous methods replicate files in the nodes that increase the likelihood but cannot guarantee that a query encounters a replica node. In SWARM, nodes can easily determine the locations of replica nodes to actively query files.

In addition, SWARM has a novel algorithm for consistency maintenance. Most consistency maintenance methods update files by relying on structures [7, 8, 13–15] or message spreading [16, 17]. A structure-based method builds all replica nodes into a structure and spreads the updates through it. Structure-based methods do not waste resources since non-replica nodes do not receive update messages, but structure maintenance generates high overhead. A message spreading method propagates updates based on either broadcast or multicast schemes. In message spreading methods, there is no structure maintenance overhead, but more overhead is needed for propagation, and some non-replica nodes may receive update messages. In addition, the methods cannot guarantee that all replica nodes receive updates. The propagation tree proposed in [14] for consistency maintenance is a traditional d-nary tree without taking proximity into account. SWARM consistency maintenance is novel in that it dynamically builds a locality-aware balanced d-nary tree with all replica nodes that does not need to be maintained and enables message propagation between nodes with close proximity in a tree fashion from the top to the bottom, thus enhancing the efficiency of consistency maintenance.

We summarize the contributions of this paper below.

- A structure construction method that efficiently builds node swarms, and a structure maintenance mechanism. It is proved that the number of messages for the construction of swarms is bounded.
- A file replication algorithm that conducts replications based on the constructed swarm structure.
- A file query algorithm that takes advantage of replicas to improve file query efficiency. It is proved that file query latency remains bounded.
- A file consistency maintenance algorithm based on the constructed swarm structure. It allows messages to be propagated between nodes with close proximity in a tree fashion without the need for tree construction and maintenance.
- Comprehensive PlanetLab and simulation experiments demonstrate the superior performance of SWARM in comparison with other file replication and consistency maintenance methods.

Our experimental results show that SWARM can reduce query latency by 40%-58%, reduce the number of replicas by 39%-76% and achieve more than 84% higher hit rates compared to previous methods. It also can reduce consistency maintenance overhead by 49%-99% compared to previous consistency maintenance methods. In this paper, we explain the SWARM mechanism within the environment of structured P2P (i.e. Distributed Hash Table (DHT)) systems, though it can also be applied to unstructured P2P systems. The conference version of this work [18] introduces the basic idea of SWARM file replication. This version provides the structure maintenance mechanism and consistency maintenance mechanism of SWARM, and also presents comprehensive experimental results from simulations and PlanetLab.

The rest of this paper is structured as follows. Section 2 presents a concise review of representative approaches for file replication, consistency maintenance, and node clustering in P2P networks. Section 3 presents an overview of the SWARM mechanism. Section 4 describes the construction and maintenance of the SWARM structure. Section 5 details the swarm-based file replication and querying algorithms. Section 6 presents the swarm-based consistency maintenance algorithm. Sections 7 and 8 show the performance of SWARM in comparison to other methods with a variety of metrics in PlanetLab and the PeerSim simulator, respectively. Section 9 concludes this paper with remarks on possible future work.

2 Related Work

Previous file replication methods can generally be organized into four categories: *Random*, *ServerEnd*, *ClientEnd* and *Path*. In the *Random* category, Chen *et al.* [4] proposed BloomCast, which replicates file items uniformly across a P2P network by randomly selecting nodes in which to replicate files. It guarantees a search success rate under a bounded query communication cost. In the *ServerEnd* category, RelaxDHT [5] replicates each file into a set number of nodes whose IDs match most closely to the file owner's ID. Beehive [6] replicates an object into nodes i hops prior to the server in the lookup path and determines a file's replication degree based on its popularity. In the *ClientEnd* category, LAR [11] replicates hot files into the file requester nodes. In the *Path* category, CUP [7], DUP [8] and Freenet [9] perform caching along the query path. *EAD* [10] enhances the utilization of file replicas by selecting query traffic hubs and frequent requesters as replica nodes. Cooperative multi-factOr considered file Replication Protocol (CORP) [19] takes into account multiple factors including file popularity and update rate to minimize the replication cost. The works in [20–22] studied the relationship between the number of replicas and the system performance such as successful queries and bandwidth consumption. As a ClientEnd method, SWARM considers the total requests of different groups of clients to determine the locations of replicating files to achieve high query efficiency with fewer replicas.

Along with file replication, numerous file consistency maintenance methods have been proposed. They generally can be classified into two categories: structure based [7, 8, 13–15, 23–26] and message spreading based [16]. CUP [7] and DUP [8] propagate updates along a routing path. SCOPE [13] constructs a tree for update propagation. Li *et al.* [14] proposed building a two-tiered update message propagation tree (UMPT)

dynamically for propagating update messages, in which a node in the lower layer attaches to a node in the upper layer with close proximity. Hu *et al.* [15] presented a tree-like framework for balanced consistency maintenance (BCoM) in structured P2P systems. Shen *et al.* [23] proposed an adaptive polling method based on a dynamic tree-like structure with locality-awareness. In these methods, if any node in the structure fails, some replica nodes are no longer reachable. Moreover, they have high overhead costs for structure maintenance, especially in churn in which node join and leave the system constantly and frequently. Some structure based works build a simple unicast client-server structure. The work in [24] relies on polling for consistency maintenance. Xiong *et al.* [25] proposed a consistent metadata processing protocol to achieve metadata consistency of cross-metadata server operations in supercomputers. Tang and Zhou [26] investigated update scheduling algorithms to improve consistency in distributed virtual environments. These methods easily overload the server due to the resource limitations of a single server. In the message spreading methods, the hybrid push/poll algorithm [16] uses rumor spreading, in which a node spreads an update to a subset of nodes without broadcasting to avoid excessive duplicate messages in flooding. In these methods, an update is not guaranteed to reach every replica, and redundant messages generate high propagation overhead. SWARM distinguishes itself by enabling message propagation among nearby nodes in a tree fashion without building a tree structure.

Supernode based, proximity based and interest based clustering [27, 28] have been extensively studied previously. Leveraging these clustering techniques, SWARM distinguishes itself by a number of novel features. First, structured P2P systems have strictly defined topologies, which has posed a challenge for clustering. SWARM deals with this challenge by taking advantage of the functions of structured P2P systems to collect the information nodes with a common interest and close proximity. Second, rather than considering either proximity or interest in swarm construction, SWARM groups nodes with joint treatment of proximity and interest. Third, to the best of our knowledge, SWARM is the first work that applies a clustering technique based on proximity and interest to file replication in order to achieve high efficiency and effectiveness.

3 Overview of SWARM

We use Chord [29] as an example to explain SWARM in structured P2P file sharing networks. SWARM can also be applied to other structured P2P systems such as Pastry, Tapestry, and CAN (Content-Addressable Network). Figure 1 shows the basic idea of the SWARM mechanism based on swarm intelligence. The nodes $a - h$, p and s have the same interest, "book". Node s is the owner of a "book" file, and other nodes frequently request the file. *ClientEnd* will make replicas in all nodes, and *ServerEnd* will copy replicas at nodes near node s on the P2P overlay that is logically close to s. *Path* replicates a file into every node along the lookup path, and *EAD* replicates a file into the forwarding nodes that receive many queries for the file. Unlike these methods, as indicated by the arrows in the figure, SWARM forms nodes with a common interest and close proximity into a node swarm, and creates one replica for each swarm. The

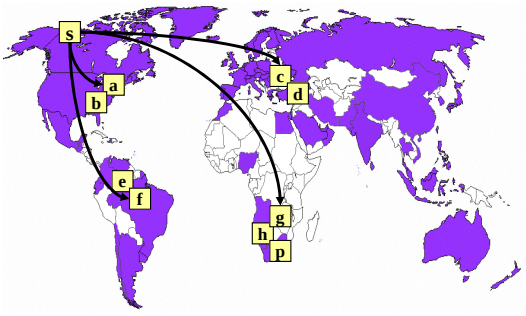


Fig. 1: File replications in SWARM.

swarms with the same interest further constitute a node colony. A replica is shared by all nodes in a swarm and can be queried using the SWARM server communication algorithm in Section 6 by all swarms in a colony. Thus, SWARM reduces replicas and replication overhead, and enhances replica utilization; it additionally improves query efficiency by allowing nodes to get a file from a nearby node. Each swarm has a supernode, which has high capacity, as a swarm server. A file update message is propagated among nearby servers in a tree fashion from the top to the bottom. Specifically, SWARM addresses the following problems:

(1) How to build a node swarm consisting of common-interest and proximity-close nodes and a node colony consisting of common-interest swarms? (Section 4)
(2) How to replicate files using the swarm-based structure and query files based on the replication algorithm? (Section 5)
(3) How to propagate a file update message between nearby servers in a tree fashion without maintaining a structure? (Section 6)

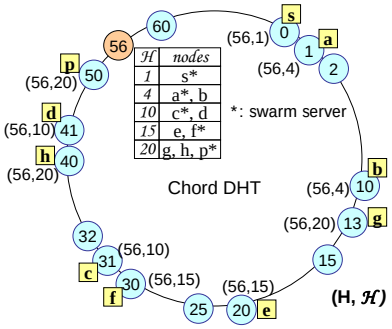
Note that the swarm-based structure for file replication and querying cannot be used for update message propagation which only involves replica nodes. Splitting a large file into small pieces can increase the service capacity of a large file rapidly. Replicating file location hints along the query path can also improve file query efficiency. SWARM can employ these techniques to further improve its performance. These techniques are orthogonal to our study in this paper.

The works in [30] and [31] measured the inter-ISP traffic in BitTorrent and indicated the importance of locality-aware traffic in reducing the traffic over long-distance connections. Since SWARM clusters nodes with a common interest and close proximity into a swarm, BitTorrent can easily adopt SWARM's techniques to facilitate their file searching with locality awareness. It can build SWARM's overly into their infrastructure, and use BitTorrent techniques to share files within a swarm. In addition to P2P systems, SWARM can also be used in the cloud datacenters where file replication and consistency maintenance are needed [32–34].

4 SWARM Structure Construction and Maintenance

4.1 SWARM Structure Construction

SWARM builds node swarms by leveraging our previous work [35] that clusters nodes with close proximity for load balance. SWARM builds the swarm structure using a landmarking method [35, 36] that represents



$\mathcal{H}$	nodes
1	s*
4	a*, b
10	c*, d
15	e, f*
20	g, h, p*

Fig. 2: Information marshaling for swarm construction. *By* `Insert(`$\mathcal{I}$`,Info)`*, the information of nodes with a common interest is marshalled in node 56, which further clusters the information of the same $\mathcal{H}$ into a group.*

node closeness on the network by indices. Landmark clustering is based on the intuition that nodes close to each other are likely to have similar distances to a few selected landmark nodes. Sophisticated strategies [37] can be used for landmark node selection. We assume m landmark nodes are scattered in the Internet. Each node measures its proximity to the m landmarks, and uses the vector of distances $< d_1, d_2, ..., d_m >$ as its landmark vector. By "proximity", we mean the average ping latency between two nodes. Two nodes in close proximity have similar landmark vectors. A Hilbert curve [36, 38] is then used to map m-dimensional landmark vectors to real numbers, such that the closeness relationship among the nodes is preserved. This mapping can be regarded as filling a curve within the m-dimensional space until it completely fills the space. The m-dimensional landmark space is then partitioned into 2^{mx} grids of equal size (where m refers to the number of landmarks and x controls the number of grids used to partition the landmark space), and each node gets a number according to the grid into which it falls. The smaller the x is, the larger the likelihood that two nodes will have same Hilbert numbers, and the coarser grain the proximity information. The Hilbert mapping may generate inaccuracy. For the details of this problem and the solution to reduce inaccuracy, please refer to [39, 40]. The *Hilbert number* of a node, denoted by $\mathcal{H}$, indicates the proximity closeness of nodes on the Internet. Two proximity-close nodes have close $\mathcal{H}$ values. SWARM clusters the nodes with similar Hilbert numbers into a swarm.

Each node's interests are described by a set of attributes described by globally known strings such as "image", "music" and "book". Each interest corresponds to a category of files. If a node does not know its interests, it can derive the attributes according to its frequently requested files [28].

Consistent hash functions [41] such as SHA-1 are widely used in DHT networks to generate node or file IDs due to their collision-resistant nature. Using this hash function, it is computationally infeasible to find two different messages that produce the same message digest. Therefore, the hash function is effective to group interest attributes because the same interest attributes will have the same consistent hash value, while different interest attributes will have different hash values.

SWARM uses the Hilbert number and a consistent hash function to build node swarms based on node interest and proximity. To facilitate such structure construction, the information of nodes with close proximity and common interests should be marshaled in one node in the DHT network, which enables these nodes to locate each other and form a swarm. Although logically close nodes may not have common interests or be in close proximity to each other, SWARM enables common-interest nodes to report their information to the same node, which further clusters the gathered information of nodes in close proximity into a group.

Algorithm 1 Pseudo-code for node join.

n.**join** ( ){
1: *//When joining in the system*
2: **for** each interest **do**
3: *//reports information to the repository node of the interest*
4: calculates the consistent hash value of the interest $\mathcal{I}$
5: sends `Insert(`$\mathcal{I}$`,Info)`
6: **end for**
7: }

Algorithm 2 Pseudo-code for having new interests.

n.**newInterest** ( ){
1: *//reports information to the repository node of the new interest*
2: **for** each new interest **do**
3: calculates the consistent hash value of the interest $\mathcal{I}$
4: sends `Insert(`$\mathcal{I}$`,Info)`
5: **end for**

Algorithm 3 Pseudo-code for losing interests.

n.**loseInterest** ( ){
1: *//reports information to the repository node of the old interest*
2: **for** each lost interest **do**
3: calculates the consistent hash value of the interest $\mathcal{I}$
4: requests to remove its information by `Lookup(`$\mathcal{I}$`)`
5: **end for**
6: }

In a DHT overlay, an object with a DHT key is allocated to a node via the interface of `Insert(key,object)`, and the object can be located via `Lookup(key)`. If two objects have the same key, they are stored in the same node. We use $\mathcal{I}$ to denote the consistent hash value of a node's interest, and $Info$ to denote the information of a node:

$$Info =< \mathcal{H}, IP, ID >,$$

where IP and ID are the IP address and ID of the node. $\mathcal{I}$ distinguishes node interests. If nodes report their information to the DHT with their $\mathcal{I}$ as the key by `Insert(`$\mathcal{I}$`,Info)`, the information of common-interest nodes with the same $\mathcal{I}$ will reach the same node, which we call a *repository node*. The highest-capacity node with interest $\mathcal{I}$ is elected to be a repository node in the DHT. The repository node further clusters the information with the same $\mathcal{H}$ into a group. As a result, a group of information in a repository node is the information of nodes with close proximity with a common interest. Figure 2 shows an example of information marshaling in Chord with the nodes in Figure 1. These nodes have interest "book", and $\mathcal{I}(book) = 56$. The 2-tuple notation such as (56,4) in the figure represents $(\mathcal{I}, \mathcal{H})$ of a node. The nodes send their information with their $\mathcal{I} = 56$ as the key, using `Insert(56,Info)`. All of the nodes' information will arrive at node 56, which further groups the information that has the same $\mathcal{H}$ as shown in the table. Consequently, the information of nodes with a common interest and close proximity is clustered in one

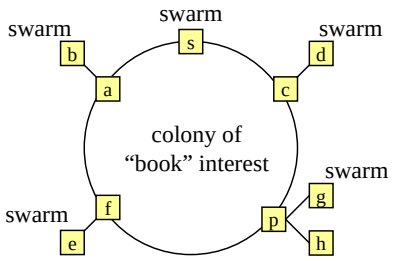


Fig. 3: The SWARM structure. *Common-interest and proximity-close nodes constitute a swarm, and all common-interest swarms constitute a colony.*

group in the repository node. Algorithm 1 shows the pseudocode of node join for information reporting. To handle repository node dynamism, SWARM relies on Chord's node join and leave algorithms [29] to transfer the responsibilities of repository nodes to their predecessors and successors as new repository nodes, and relies on its stabilization process to update the routing tables in repository nodes.

If a node has a number of interests, it reports its information based on each interest. When a node has a new interest, it reports its information to a repository node based on this interest as shown in Algorithm 2. When the node loses an interest, it asks the repository node of the interest to remove its information as shown in Algorithm 3. In this way, SWARM tunes to handle time-varying node interests dynamically, and a node colony always consists of frequent requesters. On node departure, it depends on Algorithm 3 to remove itself from all repository nodes accordingly. The node interest is important for accurately clustering nodes into swarms. Inappropriate clustering leads to excessive inter-swarm queries, since nodes may be interested in the files in other swarms. Determining node interest correctly is orthogonal to our work. In the future, we will study how to accurately derive users' interests from their file searching and sharing logs. Proposition 4.1 shows the efficiency of information marshaling in SWARM.

Proposition 4.1: In SWARM, with high probability[1], the average number of hops for reporting information for a node with m interests is $m(\log n)$, where n is the number of nodes in the network.

Proof Relying on the Chord routing algorithm [29], the path length is $\log n$ in the average case for one message. Therefore, w.h.p., the average hops for forwarding m messages is $m(\log n)$.

A node can find other nearby nodes with the same interest in its repository node with `Lookup(`$\mathcal{I}$`)`. For instance, by visiting node 56, node a can discover that node b is nearby and shares the interest in "book". After these nodes discover each other, they constitute a swarm. All swarms of an interest constitute a colony. Figure 3 shows the swarm structure corresponding to Figure 2. Specifically, in each swarm, the highest-capacity node is elected as the *server* of the other nodes (i.e. *clients*) in the swarm. Swarm servers are responsible for file query among swarms in a colony. The servers are marked with * in the repository node in Figure 2. Note there are many repository nodes rather than a single repository node in the system based on the node interest. Each swarm server can know all other servers through its repository node. In a swarm, each client has a link connecting to its server, and the server connects to a group of clients. A server maintains an index of all files and file replicas in its clients. Every time when a client accepts or deletes a file replica, it reports to its server.

For inter-swarm file searching, servers can send file queries using broadcasting, the client/server model or the DHT model. If there are many servers, broadcasting is not efficient. In the client/server model, the highest-capacity server among all swarm servers can be elected as the *colony-server* and each server connects to the colony-server. The colony-server maintains an index of files and replicas in all swarms in the colony. A server contacts the colony-server for replica queries. Assigning a single colony-server for a colony may overload the colony-server and leads to inefficiency. In the DHT model, each server needs to maintain a routing table, leading to high overhead. SWARM utilizes a novel server communication algorithm based on a locality-aware balanced d-nary tree. This algorithm is also used for consistency maintenance. The details of this algorithm will be presented in Section 6. In addition to the communication between swarm servers on the top level, this communication algorithm can also be used by a server for the communication with all of its clients in a large-scale swarm.

4.2 SWARM Structure Maintenance

A P2P system is characterized by dynamism, with nodes joining, leaving and failing constantly and frequently. When a node joins in the system or has a new interest, it reports its information with `Insert(`$\mathcal{I}$`, Info)`. At the same time, it knows its swarm's server and connects to the server directly. The server adds the newly-joined node into its index. If the newly-joined node has higher capacity than the current server, it replaces the server and the previous server becomes its client. When a client leaves the P2P system or loses an interest, it requests that its repository node(s) remove its information and notifies its server to remove the client from its client list and index. It also transfers its replicas to a neighbor client, and notifies its server about the replica transfer. When a server leaves the P2P system or loses the corresponding interest, it selects the highest-capacity client from its client list to be the new server before leaving the swarm. SWARM uses lazy-update to handle node departures without warning or failures. If a server has not received a response from a client for a file query within a predefined time period T, the server assumes the client has failed. It removes the client from its client list and index and requests the repository nodes(s) to remove the client's information. If a client has not received a reply from its server for its file query within T, the client assumes the server has failed. It communicates with other clients in its swarm for election of a new server. Since the swarm server fails, SWARM adopts a distributed election process in [42] to facilitate the swarm nodes' election. The nodes are connected to other nodes, which respond their file queries recently, and the election vote goes through the links between them. Every time when a new server is generated, it fetches the information of other servers in the same colony from its repository node and sends a notification to them.

1. An event happens with high probability when it occurs with probability $1 - O(n^{-1})$.

5 SWARM-BASED FILE REPLICATION AND QUERYING

5.1 Swarm-based File Replication

Rather than replicating a file into individual requesting nodes, SWARM considers the request frequency of a node swarm and makes replicas for the swarm. Thus, SWARM reduces the number of replicas, leading to low replication cost, and increases replica utilization, significantly improving file query efficiency. Since nodes with a shared interest and close proximity form a swarm, a node can find a frequently requested file within its own swarm without routing a query through the entire system.

We define a requester's *query rate* of file f, q_f, as the number of queries for file f that the requester initiates during a time period. The time period is determined by each application, as the file visit frequency can vary greatly across different applications. A rate calculation method based on an exponential moving average [16] can be used to reasonably determine the query rate. We define the *swarm query rate* of file f, s_f, as the number of queries for f initiated by the nodes in a swarm during one second. Nodes in close proximity have the same $\mathcal{H}$. Therefore, a requester includes its $\mathcal{H}$ in its file request in order to facilitate computing s_f.

In addition to file owners, SWARM enables replica nodes to replicate their replica files. A replica node of file f periodically calculates a file requester's q_f for a file and s_f based on $\mathcal{H}$. That is:

$$(s_f = \sum q_{f_i} | \mathcal{H}_i = v),$$

where v is a specific value of $\mathcal{H}$. When overloaded, the node replicates the file in the most frequent requester in the swarm with the highest s_f. Such a replication strategy increases the replica utilization by making it shared by more frequent requesters. Specifically, the node chooses the swarm with the highest s_f, then orders the swarm nodes in descending order of q_f, and selects a non-replica node in a top-down fashion. The replica node will report to its server of its new replica.

Initially, a file owner replicates files when overloaded. Later, when a replica node is overloaded, it replicates its file to another node. Thus popular files will have increasing numbers of replica nodes until no node is overloaded due to file queries.

Unlike *ClientEnd* and *EAD* that replicate a file in all frequent requesters, SWARM avoids under-utilized replicas by replicating a file for a group of frequent-requesters in close proximity. It helps guarantee that file replicas are fully utilized. Requesters that frequently query for file f can get file f from itself or its swarm without query routing. Thus, SWARM improves file query efficiency and saves file replication overhead.

Considering that file popularity varies over time, some file replicas become under-utilized when there are few queries for them. To cope with this situation, SWARM sets up a threshold T_f for swarm query rate that is determined based on the average swarm query rate. If $s_f \leq \alpha T_f$, where α ($\alpha < 1$) is a percentage factor, a replica in the swarm will be removed. A smaller T_f value leads to more replicas with larger query rates while a larger T_f value leads to fewer replicas with smaller query rates. Thus, a smaller T_f increases replica hit rates at the cost of storage and consistency maintenance overhead, while a larger T_f reduces costs but may decrease the file hit rate. A proper value of T_f is determined based on the hit rate requirement of the system and the overhead it can afford. Hence, the decision to keep file replicas is based on recently experienced query traffic due to file popularity and query rate. If a file is no longer requested frequently, there will be fewer file replicas for it. The adaptation to swarm query rate ensures that all file replicas are worthwhile and no overhead is wasted on unnecessary file replica maintenance. Algorithm 4 shows the file replication algorithm pseudocode. SWARM's replication algorithm works well if the swarm query rate of a file f does not fluctuate frequently and largely over time; otherwise, SWARM may waste replication costs due to frequent deletion of replicas with a small swarm query rate and frequent creation when the swarm query rate becomes large. However, in reality, a file is usually requested frequently soon after its creation and rarely has queries after a certain time, such as the shared data in [43]. Therefore, the replication cost waste is usually not high. To overcome the frequent replication cost in SWARM, a replica node deactivates a replica instead of deleting it when there is enough storage capacity. Then, when this replica needs to be created again, the file owner only needs to send out all updates after its deactivation to save on replication costs.

Algorithm 4 Pseudo-code for file replication in a replica node.

```
n.manageReplica ( ){
 1: //periodically executed by a node for each file
 2: updates q_f of each requester
 3: updates s_f based on the H of the requesters
 4: while n is overloaded do
 5:   //replicates the file
 6:   chooses the swarm with the highest s_f
 7:   orders nodes in the swarm in descending order of q_f
 8:   for each node from the top of the list do
 9:     if the node is a non-replica node then
10:       replicates the file in the node
11:       update n's load by excluding s_f
12:       break
13:     end if
14:   end for
15: end while
16: for any swarm's_f ≤ αT_f do
17:   //removes under-utilized replicas
18:   orders nodes in the swarm in ascending order of q_f
19:   for each node from the top of the list do
20:     if the node is a replica node then
21:       notifies the replica node to remove the replica
22:       break
23:     end if
24:   end for
25: end for
26: }
```

5.2 File Querying Algorithm

When node i requests a file, if the file is not in the requester's interests, the node uses a DHT `Lookup(key)` function to query the file. Otherwise, node i first queries the file in its swarm among nearby nodes interested in the file, and then in its colony among nodes interested in the file. Specifically, node i first sends a request to its swarm server for that file's interest category. The server searches the index for the requested file in its swarm. If the search fails, the server uses the SWARM server communication algorithm (Section 6) to query for the file replica in a nearby swarm. This communication

algorithm enables servers to propagate messages in a d-nary tree fashion when the number of servers S is greater than a threshold T_s. If this search also fails, node i uses DHT routing to search for the file.

Algorithm 5 Pseudo-code for the file query in a requester.

```
n.lookup_requester (key){
 1: gets file f's key;
 2: if key ∈ interests then
 3:    sends request to its server of swarm H
 4:    if receives negative response from the server then
 5:       Lookup(key);
 6:    end if
 7: else
 8:    Lookup(key);
 9: end if
10: }
```

Algorithm 6 Pseudo-code for the file query in a swarm sever.

```
n.lookup_swarmserver (key){
 1: if receives query from its client then
 2:    if there is a replica of requested file in its swarm then
 3:       returns the location of the replica
 4:    else
 5:       uses SWARM server communication algorithm to query for a
          replica in the colony
 6:       if receives the location of a replica then
 7:          responds to node n of the location
 8:       else
 9:          returns a negative response
10:       end if
11:    end if
12: end if
13: }
```

For example, in Figure 3, when a node g queries for a file in "book", it sends a request to node p, which returns the location of a replica in its swarm. If there is no replica in its swarm, node p relies on the swarm server communication algorithm to query for a replica location in the colony. If there is no replica in the colony, node g uses `Lookup(key)` to request the file from its file owner. Algorithms 5 and 6 show the pseudocode for the file query algorithms in a requester and a swarm server, respectively. We then define S as the number of swarms in a colony, and n as the number of nodes in the network. Proposition 5.1 demonstrates the efficiency of file query in SWARM.

Proposition 5.1: In SWARM, w.h.p., the average lookup latency for a file is $3\gamma + \log_d S\beta + (1-\gamma-\beta)\log n$ hops, where $S > T_s$ and γ and β represent the probabilities that a replica of the requested file exists in the requester's swarm and colony respectively.

Proof In SWARM, if a replica of a requested file exists in a requester's swarm, it takes the requester three hops' latency to query its server for a requested file. Otherwise, if a replica exists in the requester's colony and $S > T_s$, it takes the requester $\log_d S$ hops' latency to inquire for the file in the tree-based communication. If there is no replica in the colony, using the Chord routing algorithm, the path length is $\log n$ on average.

6 SWARM-BASED CONSISTENCY MAINTENANCE

SWARM adopts a single-master replication protocol [43], in which the file owner holds the master replica and all other replicas are slaves. According to the protocol, a slave replica node forwards an update to the file owner, which then pushes the update to all slave replicas. In consistency maintenance, the client that updates a file sends an update message to its swarm server which is responsible for propagating the message to other swarm servers. After a server receives the update message, it further forwards the message to its clients that have the replicas of the file. As mentioned, structure-based and message spreading propagation methods have their own advantages and disadvantages. To combine the advantages of both methods, the SWARM consistency maintenance algorithm sets a threshold for the number of swarms T_s, which is a value that will not generate overhead burden on nodes for broadcasting messages. When $S < T_s$, a server uses broadcasting for update. Otherwise, a server propagates messages based on a locality-aware balanced d-nary tree (LBDT).

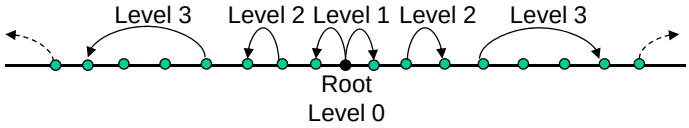


Fig. 5: The process of child assignment in LBDT.

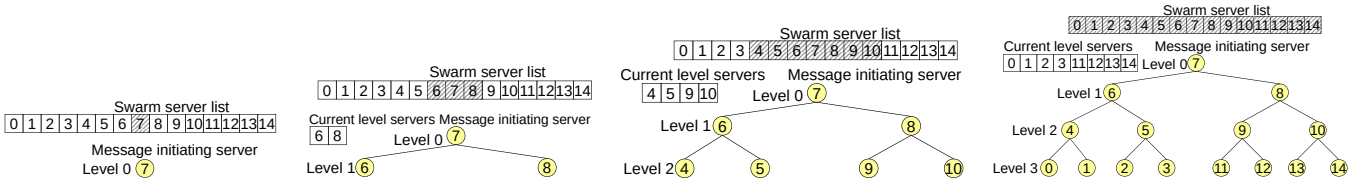
General d-nary tree construction methods [14] build a traditional d-nary tree based on node IDs but do not consider node proximity. The d-nary tree allows all nodes to receive updates quickly, but cannot enable updates to be propagated between nearby nodes. The proposed LBDT-based propagation method distinguishes itself in three aspects. First, it takes proximity into account to enable messages to be propagated among nodes in close proximity, reducing communication costs. Second, it enables updates to be propagated in the fashion of a balanced d-nary tree, improving communication efficiency. Third, it does not require the construction and maintenance of a tree structure, reducing overhead. After a swarm server receives a message, it discovers its children in the LBDT and sends messages to them.

"No tree structure construction" means that the entire tree does not need to actually be constructed. Instead, the tree is dynamically and virtually built in a file owner with all replica nodes and itself as nodes in the tree whenever there is an update. SWARM propagates the updates based on the source routing until they reach the leaf nodes in the tree; that is, the update is sent to a child along with the sub-tree, in order to help the child further propagate the update. "No tree structure maintenance" means that the connected nodes in the tree do not need to probe each other periodically, since the tree is dynamically constructed before update propagation. As in previous consistency maintenance methods that require the file owner to maintain the replica nodes of its file, SWARM requires each swarm server to maintain other swarm servers for replica nodes.

Below, we first introduce LBDT and then introduce how a swarm server identifies its children in LBDT in a distributed manner to forward an update.

6.1 Locality-aware Balanced d-nary Tree (LBDT)

To enable the communication conducted between physically closer swarm servers, SWARM takes advantage of the Hilbert number to allow the messages to be propagated along an LBDT. In an LBDT, each non-leaf node has d children which have close Hilbert numbers to its Hilbert number. Figure 4(d) shows a simple example of LBDT with $d = 2$. Assume a tree has L levels in



(a) Step 1 (b) Step 2 (c) Step 3 (d) Step 4

Fig. 4: An example of of LBDT-based message propagation.

total. Level l ($0 \leq l < L$) has d^l nodes, denoted by N[i] ($0 \leq i < d^l$), from the left to the right. The index i is called *the index of the node in level l*. We use N^l_{lt} and N^l_{rt} to denote all nodes in the left and right half of nodes in level l, respectively. In a LBDT, as shown in Figure 4(d), the Hilbert numbers of N^l_{lt} are less than those of N^{l-1}_{lt}, and the Hilbert numbers of N^l_{rt} are higher than those of N^{l-1}_{rt}. That is,

$$\mathcal{H}_{N^l_{lt}} < \mathcal{H}_{N^{l-1}_{lt}} \quad and \quad \mathcal{H}_{N^l_{rt}} > \mathcal{H}_{N^{l-1}_{rt}}.$$

In the horizontal direction, the Hilbert numbers of nodes at the same level are in ascending order. Thus, nodes with close Hilbert numbers are connected. That is, proximity closer nodes are connected in the LBDT.

We first introduce how to construct an LBDT given a list of swarm servers (SS) sorted by their $\mathcal{H}$s as shown in Figure 4(a). The tree root is the middle server in the list. In a high-level view of the construction process, the children are assigned to the tree one level by one level in the top-down manner. We first use Figure 5 to demonstrate the process of child assignment. The *root* is in level 0 of the tree. Its predecessor and successor in the list become its children in level 1, child[0] and child[1]. Then, the next two nodes on the left become the children of child[0], the next two nodes on the right become the children of child[1], and they are in level 2. After that, the next four nodes on the left and right respectively become the children in level 3, and so on.

Therefore, the d^l nodes at level l actually are the d^l nodes in the center of SS that have not been assigned to the tree. Thus, to construct an LBDT, starting from the root, d^1 nodes in the center of *SS* are fetched and assigned as the children of the root at level 1 (Figure 4(b)). The shaded elements mean that the servers are fetched and assigned. Then, d^2 nodes in the center of *SS* that have not been assigned are fetched and assigned to the nodes in level 1 as their children in level 2. The children assignment to parents is from the left to the right, and every parent has d children (Figure 4(c)). After that, d^3 nodes in the center of *SS* that have not been assigned are fetched and assigned to the nodes in level 2 as their children in level 3 (Figure 4(d)). This process is repeated until all servers in the list are fetched and assigned to the tree. Generally, we assume the file server sorts all replica nodes by their Hilbert numbers and forms a vector of all replica nodes as $V =< r_0, r_1, ..., r_m >$. We define $lstart$ and $lend$ as the index of first and last nodes at level l in this vector, and we assume d is even for simplicity. Then, the next nodes in the next level should be $V^{l+1} =< r_{l'start}, .., r_{lstart-1}, r_{lend+1}, .., r_{l'end} >$, where $l'start = lstart - d^(l+1)/2$ and $l'end = lend + d^(l+1)/2$. For the k^{th} node of level l, the $(k*d)^{th}$ to $((k+1)*d-1)^{th}$ nodes in V^{l+1} are its children. This regular children assignment can save file owner's workload of calculating and transferring the whole d-nary tree structure; by simply transferring the information as the procedure input shown in Algorithm 7, the child can find their children recursively. The child assignment achieves the smallest total Hilbert number $\mathcal{H}$ difference between the parents and children in two adjacent levels in LBDT. This is because the difference is calculated as $\sum \mathcal{H}_k - \sum \mathcal{H}_j - \sum_{i=l'start}^{lstart-1} \mathcal{H}_i + \sum_{i=lend+1}^{l'end} \mathcal{H}_j$, where $\mathcal{H}_k$ stands for the Hilbert number of the parent for a child between $l'start$ to $lstart$, and$\mathcal{H}_j$ stands for the Hilbert number of the parent for a child between $lend$ to $l'end$. Since the remaining part in the equation is constant, our assignment achieves the smallest $\sum \mathcal{H}_k - \sum \mathcal{H}_j$, which equals $d * \sum_{i=lstart}^{lstart+d^l/2-1} \mathcal{H}_i - d * \sum_{i=lend-d^l/2+1}^{lend} \mathcal{H}_i$. That is, our method achieves the smallest difference between the parents and children in two adjacent levels in LBDT.

Algorithm 7 Pseudo-code of LBDT-based message propagation between swarm servers.

```
n.MsgChildren (l, lstart, lend, k, SS){
1: //l: current level
2: //[lstart,lend]: the index range in SS of current level
3: //k: this node's index at current level
4: if lstart > 0 || lend < SS.Length - 1 then
5:    // this node has children, and it calculates the index range in SS of the
      children in the next level
6:    cnodes = d^(l+1) // # of nodes at the next child level
7:    // calculates the start index of the next child level
8:    cstart = lstart - Floor(cnodes / 2)
9:    if IsOdd(l) and IsOdd(d) then
10:      cstart=cstart-1
11:   end if
12:   if cstart < 0 then
13:      cstart = 0
14:   end if
15:   // calculates the end index of the next child level
16:   cend = lend - lstart + cstart + cnodes
17:   if cend ≥ SS.length then
18:      cend = SS.length - 1
19:   end if
20:   // discovers its own children in SS[cstart,cend]
21:   cindex = 0 // child level node index
22:   for i = cstart to cend do
23:      if i < lstart || i > lend then
24:         // SS[i] is a node at the next level
25:         if k == Floor(cindex++ / d) then
26:            // SS[i] is the node's child
27:            sends the message to SS[i] with l = l + 1, lstart = cstart,
               lend = cend, k = cindex
28:         end if
29:      end if
30:   end for
31: end if
32: }
```

6.2 Distributed Update Propagation in LBDT

We now explain how update messages are propagated based on LBDT in SWARM in a distributed manner in order to save consistency maintenance costs with locality awareness. In file consistency maintenance, the swarm server initiating the update message becomes the root of

LBDT. It locates its children in the LBDT, and forwards the message to them. The children further discover their own children and send the messages to them, and so on. Thus, the messages are propagated one level by one level in a top-down manner. Specifically, each node in level l executes Algorithm 7 to discover its children in level $l+1$, and messages them. This process is repeated until all swarm servers receive update messages.

We use Figure 4 to explain the algorithm. The shaded elements represent the servers that have already received update messages. We assume the root's $\mathcal{H} = 7$. At first, as shown in Figure 4(a), the root sorts the swarm servers (including itself) into a ring based on their Hilbert numbers, and breaks the ring to generate a *SS* with itself in the middle. The root is in the middle of the sorted list in the example. If the root's $\mathcal{H} = 4$, then the *SS* is [12,13,14,0,1...4,5...11]. The root is in level 0 of the tree, and its index in level 0 is 0. The index range of level 0 in SS is [7,7]. The root executes *MsgChildren(0,7,7,0,SS)* to discover its children and send update messages to them. To discover its children, a node calculates the index range in SS of all children in the next level (line 7-19), and selects its own children among the children (line 22-30).

First, the root calculates the number of nodes in the next child level: *cnodes*=$2^1 = 2$ (line 6). It then calculates the start index and the end index of the next level: *cstart*=7-1=6 (line 10) and *cend*=7-7+6+2=8 (line 16-19). Thus, the index range in SS of children in level 1 is [6,8]. When d is an odd number, in the odd level, *cstart* is decremented by 1 (line 9-14) for the balance of the propagation tree. Then, the root selects its children satisfying *k == Floor(cindex++/d)* where *cindex* is the index of a child in its level (line 25). All children in the next level are the root's children, and the root sends the message to its children. This step is shown in Figure 4(b). After servers 6 and 8 receive the messages, they execute *MsgChildren(1,6,8,0,SS)* and *MsgChildren(1,6,8,1,SS)*, respectively. Server 6 finds the index range of all children in level 2 is [4,10]. It selects its own children 4 and 5 from the children and sends messages to them. Similarly, server 8 locates its children 9 and 10 and sends messages to them. This step is shown in Figure 4(c). After the children receive the messages, they execute the algorithm to locate their own children and send messages to them, which is demonstrated in Figure 4(d).

This message propagation algorithm can also be used for the communication between swarm servers for replica query. After a server receives a query, it checks its index. If its client has the requested file, it responds to the root without further forwarding the message to its children. In the LBDT-based communication, children and their parent are nodes in close proximity, which helps improve communication efficiency.

7 Performance Evaluation on PlanetLab

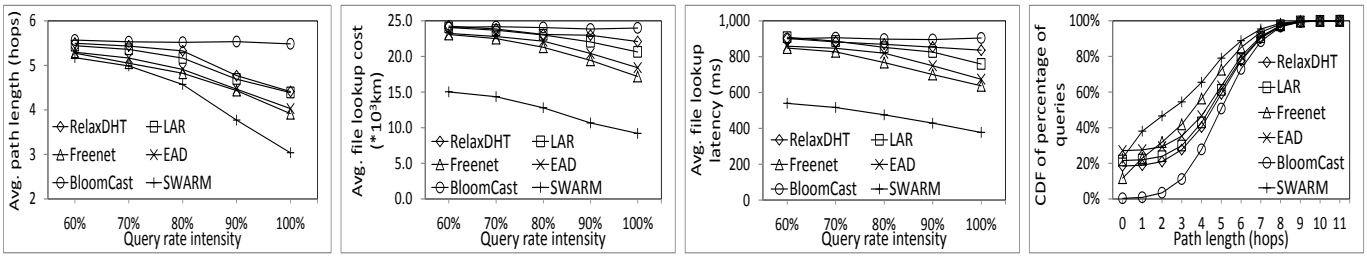
This section first presents the performance evaluation of SWARM in comparison with *LAR* [11], *RelaxDHT* [5], *Freenet* [9], *EAD* [10] and *BloomCast* [4] in file replication. We use *LAR* to represent methods in the *ClientEnd* category. In *LAR*, replicas are created in frequent requesters. We use *Freenet* to represent methods in the *Path* category, in which replicas are created along the lookup paths of a file. *EAD* selects querying traffic hubs and frequent requesters as replica nodes. We use *RelaxDHT* to represent methods in the *ServerEnd* category, in which replicas are created in the neighbors of the original file owner. We use *BloomCast* to represent methods in the *Random* category, which replicates files uniformly across the P2P network by randomly selecting nodes to replicate files. A file owner stops replicating files when it releases its excess load by replication. We use *replica hit rate* to denote the percentage of queries that are resolved by replica nodes among total queries.

We also compared SWARM with three push-based consistency maintenance methods, SCOPE [13], UMPT [14] and BCoM [44], with regards to proximity-aware message propagation. In SCOPE, all nodes form a tree without considering locality to propagate updates. In UMPT, high-capacity nodes dynamically form an update tree; then, low-capacity nodes attach to nodes in the tree with close proximity to receive updates. In BCoM, replica nodes of each file are organized into a d-ary ($d = 2$ in our experiments) dissemination tree without proximity-awareness to propagate updates. In SWARM, since not all nodes can use ping to measure the network distance to the selected landmarks, we use geographical distance to measure the distance vector for all experiments.

We conducted experiments on the real-world PlanetLab testbed [45]. We randomly selected 256 PlanetLab nodes all over the world, and used each node to represent 8 virtual peers. We constructed the 2048 virtual peers into a Chord P2P network with the dimension equals 11. We used the statistics derived from 4 million MSN video users' viewing behavior in the trace collected by Microsoft [46] to decide the files and node capacity in our experiments. Specifically, we randomly selected 500 files from the trace; each file has its query rate q and file size. We also randomly selected 2,048 nodes from the trace to map their capacities to our P2P nodes. In the experiments, we set each file's query rate to ϕq, where ϕ (called *query rate intensity*) was varied from 60% to 100% with 10% increase in each step. Based on a BitTorrent trace [47], we set the total number of interests to 200. Each node had 5 randomly selected interests. Each file was randomly assigned an interest, and was assigned to a randomly selected node with the interest. Each node randomly selected 10 files in its interests to query. For a given file with query rate ϕq, the query rates of its requesters follow a Power Law distribution [48], in which 20% of requesters account for 80% of the queries for the file.

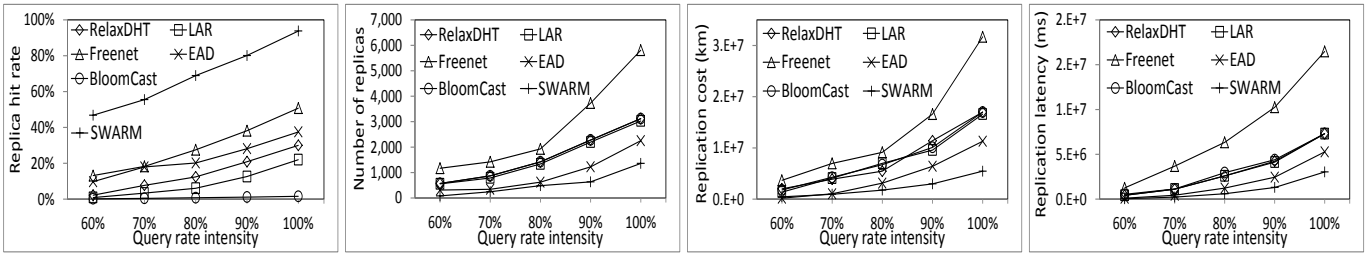
7.1 Performance of File Replication

Figure 6(a) plots the average path length for different query rate intensities. We can see that *BloomCast* generates the longest path length, *RelaxDHT* and *LAR* generate longer path lengths than *Freenet* and *EAD*, and SWARM leads to the shortest path length. All methods except SWARM are unable to guarantee that every query can encounter a replica. In contrast, SWARM enables a replica to be shared within a group of frequent requesters, which dramatically increases the utilization of replicas and reduces path lengths. SWARM reduces the path length of other methods by 22%- 44% when the query rate intensity equals 100%. The figure also shows that the average path lengths of all methods except



(a) Avg. path length (b) Avg. file lookup cost (c) Avg. file lookup latency (d) CDF of lookup lengths

Fig. 6: Effectiveness of the file replication methods on PlanetLab.



(a) Replica hit rate (b) Total number of replicas (c) Total replication cost (d) Total replication latency

Fig. 7: Effectiveness and efficiency of the file replication methods on PlanetLab.

BloomCast decrease as the query rate intensity increases. A larger query rate intensity leads to more overloaded servers hence more replicas. The decreasing rates of the methods follow the same order as the path lengths, i.e., *BloomCast*>*RelaxDHT*/*LAR*>*Freenet*/*EAD*>SWARM, due to the same reasons. Figure 6(a) indicates that SWARM offers the most efficient file querying by reducing the expected path lengths due to its most effective file replication.

Figure 6(b) shows the average file lookup cost measured by the total geographical distance of a query path. It shows similar trends and order of performance for the methods as in Figure 6(a) due to the same reasons. Figure 6(c) shows the average query latency versus the query rate intensity. All methods show the same tendency as in Figure 6(b) due to the same reasons. SWARM reduces the query latency of other methods by 40%- 58% when the query rate intensity equals 100%. These results confirm that SWARM achieves the shortest query cost and latency in all methods due to its effective file replication.

Figure 6(d) shows the cumulative distribution function (CDF) of the query path lengths when the query rate intensity equals 100%. We see that in SWARM, around 50% of total files are offered by nodes within 2 hops, while in other methods, 50% of files are offered by nodes within at least 4 hops. In SWARM, almost all files are provided by nodes within 8 hops, but other methods require 10 hops. The results show that most files can be offered by nodes in shorter distances in SWARM than other file replication methods due to the same reasons as in Figure 6(a).

Figure 7(a) shows the replica hit rates of all methods versus the query rate intensity. We observe that the replica hit rate follows *BloomCast*<*LAR*<*RelaxDHT*< EAD<*Freenet*<SWARM. SWARM replicates a file for a group of common-interest nodes and enables a node to actively retrieve the locations of replica nodes, which significantly improves the probability that a file query is resolved by a replica node, leading to the highest hit rate. As a result, SWARM achieves more than 84% higher hit rates compared to other methods. This result confirms that SWARM achieves the highest replica hit rate, indicating that the replicas in SWARM have the highest utilization and relieve greater load from the file owners.

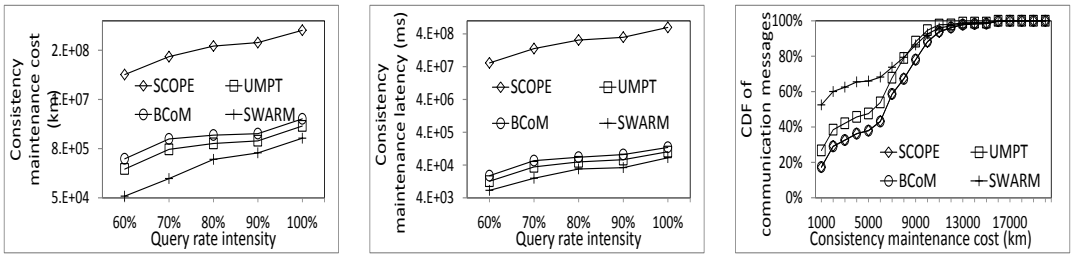
Figure 7(b) illustrates the total number of replicas of all methods versus query rate intensity. It shows that the number of replicas increases as the query rate intensity increases because of more overloaded file owners. In SWARM, a replica is created for a swarm rather than a single node. Hence, a replica is fully utilized through being shared by a group of nodes, generating high replica hit rate and reducing the probability that a file owner is overloaded. Thus, SWARM produces fewer replicas and lower overhead for replica consistency maintenance. Specifically, SWARM reduces the number of replicas compared other methods by 39%-76% when the query rate intensity equals 100%, which also means 39%-76% less overhead for replica consistency maintenance.

Figures 7(c) and 7(d) show the total replication cost measured by the geographical distance and latency of all file replications, respectively. They show similar trends and performance as in Figure 7(b) due to the same reasons, which confirm that SWARM generates the lowest cost and highest efficiency for replication.

7.2 Performance of Consistency Maintenance

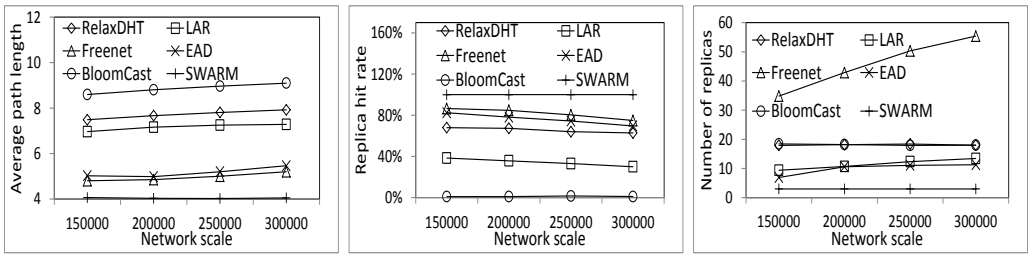
In this experiment, we test the proximity-aware performance of SWARM's LBDT-based consistency maintenance method. The update rate of each file was randomly chosen from [0.05,0.15] updates per second. We report the average total consistency maintenance cost and latency every 10s for 200s.

We measured the consistency maintenance cost by the total transmitting geographical distance of all update messages. Figure 8(a) and Figure 8(b) show the consistency maintenance cost and latency, respectively, versus the query rate intensity. The figures show that the cost and latency follow SWARM<UMPT<BCoM<SCOPE. All methods except SWARM either propagate update messages to all nodes or propagate updates without considering proximity to partial replica nodes. Compared



(a) Total consistency maintenance cost (b) Total consistency maintenance latency (c) CDF of update message propagation distances

Fig. 8: Consistency maintenance cost and latency on PlanetLab.



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

Fig. 9: Scalability of the file replication methods on PeerSim.

to them, SWARM always propagates update messages along geographically close replica nodes, thus leading to the lowest communication cost and latency. SWARM reduces the consistency maintenance cost by 66%-87% compared to BCoM due to its locality awareness. It reduces the consistency maintenance cost by 99% compared to SCOPE due to its saved number of update messages. Overall, SWARM reduces consistency maintenance costs by 49%-99% compared to other consistency maintenance methods. We see that the cost and latency of all methods increase as the query rate intensity increases. As query rate intensity increases, the number of replicas increases as shown in the Figure 7(b), which introduces higher update costs and longer latencies. The experimental results imply that SWARM is effective in enabling messages being propagated between geographically close nodes, while still achieving fast propagation due to tree structure.

Figure 8(c) depicts the CDF of communication messages versus the consistency maintenance cost when the query rate intensity equals 100%. We see that 52% of total messages in SWARM are transmitted between a pair of nodes within 1000km of each other, while only 26% of total messages in other methods are. Similarly, SWARM transmits 66% of total messages between nodes within 5000km of each other, while other methods transmit 45%. The results show that most messages are transmitted short geographical distances in SWARM, while most messages are transmitted in long distances in the other methods due to the same reason as in Figure 8(a). It confirms the high efficiency of the locality-aware LBDT-based consistency maintenance method in SWARM.

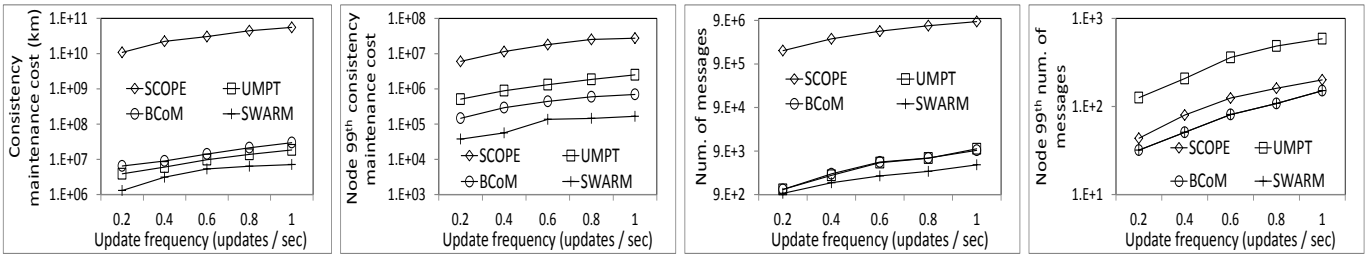
8 Performance Evaluation on PeerSim

While PlanetLab experiments can provide real-world experimental results in terms of geographical distance and latency, it cannot test large-scale networks and the performance in different degrees of churn. To comprehensively test SWARM, we then implemented the methods on the event-driven P2P simulator PeerSim [49, 50] for performance evaluation. In this section, we aim to test the performance of SWARM in large-scale P2P networks with different degrees of churn, different update rates and different number of replicas. The network has 150,000 nodes unless otherwise noted, and the maximum number of nodes is 300,000, around 1/10 of all nodes in [47]; a node's five most frequent download/upload file categories in the trace are set to the node's interests; the geographic locations of the nodes are randomly selected from node geographic locations in the BitTorrent trace [47]. Similarly, we set the number of interests to 200, and randomly selected 10000 files from the trace [47]. The node capacity distribution was determined based on the MSN trace data set [46]. Each experiment lasted 10,000s, and 100 queries were generated per second. These 100 requesters were randomly chosen from all peers; each peer queried a randomly selected file within its interests.

8.1 Scalability of File Replication

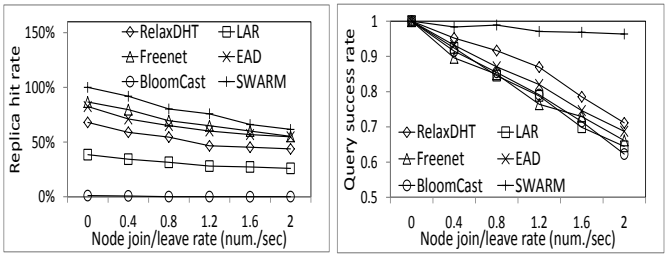
Figure 9(a) shows the average path lengths of different file replication methods versus the network scale. It shows that *BloomCast* result in the longest average path lengths, and *LAR* and *RelaxDHT* generate shorter path length followed by *EAD* and *Freenet*; SWARM produces the shortest path length due to the same reasons as Figure 6(a). This figure shows that as the network scale increases, the average path length of SWARM increases slightly while that of other methods increases relatively faster. This result implies that SWARM is more scalable than other methods.

Figure 9(b) shows the replica hit rate of different file replication methods versus the network scale. As in Figure 7(a), SWARM has the highest hit rate, followed by *Freenet*, *EAD*, *RelaxDHT*, *LAR* and *BloomCast*. We also see that *Freenet*, *EAD* and *LAR* decrease gradually as network scale increases, while SWARM and *RelaxDHT* remain nearly constant. As the number of nodes increases, the replicas in *Freenet*, *EAD* and *LAR* have lower



(a) Total communication cost (b) The 99th percentile of a node's communication cost (c) The total number of messages (d) The 99th percentile of a node's number of messages

Fig. 11: Consistency maintenance cost versus update frequency on PeerSim.



(a) Replica hit rate (b) Querying success rate

Fig. 10: Performance of the file replication methods with churn in PeerSim.

probability to be encountered by many requests, while the replicas in SWARM and *RelaxDHT* always have a constant probability to be encountered by each request. This figure confirms that SWARM can maintain a stable replica hit rate at different network scales.

Figure 9(c) shows the number of replicas as network scale increases. *Freenet* has the greatest number of replicas, followed by *RelaxDHT*/*BloomCast*, *LAR*, *EAD*, and finally SWARM due to the same reason as in Figure 7(b). Unlike other methods, SWARM replicates files into interest- and locality-based swarms; thus, its number of replicas is not greatly affected by network size.

The experimental results in Figures 9(a), 9(b) and 9(c) confirm that SWARM has high scalability, high replica utilization and low overhead for file replication.

8.2 Churn-resilience of File Replication

Figure 10(a) shows the replica hit rates for different replication methods versus the node join/leave rate. The performance of all methods except *BloomCast* degrades similarly as churn increases; this is expected, since some departing nodes might hold replicas. Additionally, all five methods rely on P2P routing to a certain degree, and node churn can adversely affect routing success. In *BloomCast*, since its hit rate is already very small as shown in Figure 9(b), it is not greatly affected by the churn. Figure 10(b) shows the success rate of queries for different replication methods versus the node join/leave rate. The success rates of all methods except SWARM have similar decreasing trends as the rate increases; since each method uses DHT routing to locate a file, the success rate decreases as churn increases. Since the SWARM replica querying method takes much fewer hops, the success rate for SWARM is not as greatly affected by churn as other methods. The experimental results show that SWARM has higher churn-resilience than other methods.

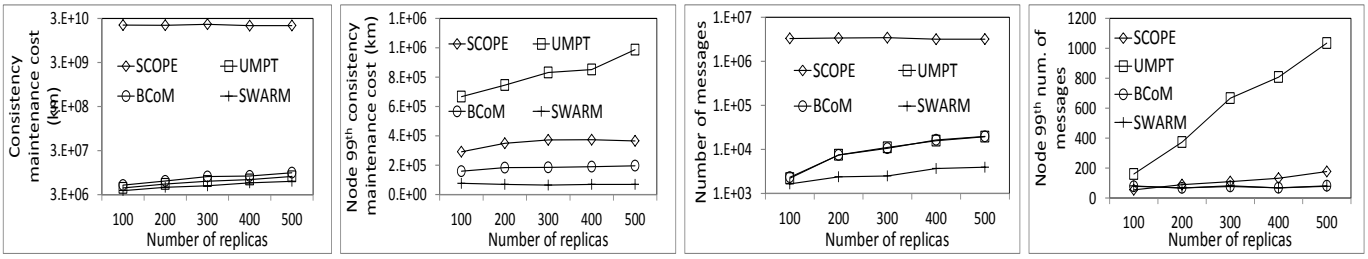
8.3 Performance of Consistency Maintenance

In this experiment, we test SWARM's consistency maintenance scheme against SCOPE, UMPT and BCoM. Since none of these experiments consider the update/visit rates for consistency maintenance and they build the structures based on the proximity, the update rate will not affect the performance difference among systems. Thus, we set a high update rate in order to distinguish the performance differences between different systems in a short time. SWARM replicas are shared by entire swarms, which are clustered by both interests and locations. There are only 169 different locations in the trace [47], which leads to at most 169 swarms in a colony. Therefore, unless otherwise noted, we randomly placed 100 replicas for a single file and set the update rate to 0.1 updates/second. Figure 11(a) shows the consistency maintenance cost measured by total distances versus the file update frequency. SCOPE produces the greatest communication cost, followed by BCoM, UMPT, and then SWARM, due to the same reason as in Figure 8(a). It confirms that SWARM produces the lowest communication cost since it propagates update messages along geographically close nodes.

Figure 11(b) shows the 99th percentile of a node's total communication cost in consistency maintenance. UMPT has the highest communication cost, SCOPE has lower communication cost, followed by BCoM and SWARM. UMPT produces the highest communication cost because many low-capacity nodes attach to a single high-capacity node to receive updates, resulting in a highly uneven distribution of communication cost. SCOPE, BCoM and SWARM propagate updates in a tree-like fashion, so message sending is more evenly distributed among nodes. As SCOPE involves all nodes in the system while SWARM and BCoM only involve the replica nodes, SCOPE generates a higher result. Since SWARM builds a locality-aware tree while BCoM builds a locality-oblivious tree, SWARM generates lower communication cost than BCoM. This result confirms that SWARM generates more even distribution of update overhead among nodes, which indicates its better scalability.

Figure 11(c) shows the total number of messages sent in each consistency maintenance method versus the update frequency. SCOPE propagates updates throughout the entire network, so its total number of messages is significantly higher than BCoM, UMPT and SWARM. These three methods build only replica nodes into trees for propagating updates, requiring much fewer messages for consistency maintenance.

Figure 11(d) shows the 99th percentile of a node's number of messages. UMPT has the highest number of messages, SCOPE has the middle number of messages, and SWARM and BCoM have the lowest number of messages. UMPT has the highest result because many



(a) Total communication cost (b) The 99th percentile of a node's communication cost (c) The total number of messages (d) The 99th percentile of a node's number of messages

Fig. 12: Consistency maintenance cost versus the number of replicas of a file on PeerSim.

low-capacity nodes attach to a single high-capacity node to receive updates, resulting in a highly uneven distribution of sending messages. SCOPE, BCoM and SWARM propagate updates in a tree-like fashion, so message sending is more evenly distributed among nodes. As SCOPE involves all nodes in the system while SWARM and BCoM only involves the replica nodes, SWARM and BCoM have a lower result than SCOPE.

Figure 12(a) shows the consistency maintenance cost measured by the total distances versus the number of replicas of a single file. SCOPE generates significantly higher communication cost than BCoM, UMPT and SWARM. Also, as the number of replicas increases, SCOPE remains nearly constant while BCoM, UMPT and SWARM increase slightly. Figure 12(b) shows the 99th percentile of a node's communication cost. UMPT has the highest result, SCOPE is in the middle followed by BCoM, and SWARM has the lowest result. The results in these two figures are caused by the same reasons as in Figure 11(a) and Figure 11(b), respectively.

Figure 12(c) shows the total number of messages for consistency maintenance versus the number of replicas of a single file. As in Figure 11(c), SWARM has the fewest messages, followed by BCoM, UMPT and SCOPE, for the same reasons. Figure 12(d) shows the 99th percentile of a node's number of messages. UMPT generates the highest result, followed by SCOPE, then BCoM and SWARM due to the same reasons as in Figure 11(d).

These experimental results confirm the effectiveness of the LBDT-based consistency maintenance method in reducing both the highest and the average overhead for updates of a node, which indicates that SWARM has higher scalability and efficiency than other methods in consistency maintenance.

9 Conclusions

Current file replication methods for P2P file sharing systems are not sufficiently effective in improving file query and replica utilization. This paper proposes a swarm intelligence based file replication and consistency maintenance mechanism called SWARM. SWARM includes algorithms for swarm structure construction and maintenance, file replication, querying, and file consistency maintenance. It builds common-interest nodes in close proximity into a swarm and relies on swarm servers to connect swarms with the same interest into a colony. It replicates a file in a swarm with the highest accumulated file query rates of the swarm nodes and enables the replica to be shared among the nodes in a swarm and colony. Furthermore, it dynamically adapts to time-varying file popularity and node interest. The SWARM consistency maintenance algorithm allows updates to be propagated among nearby swarm servers in a tree fashion, enhancing the efficiency of traditional tree-based propagation. SWARM reduces file replicas and improves query efficiency and replica utilization. Experimental results demonstrate the superiority of SWARM in comparison with other approaches. It dramatically reduces the overhead of file replication and consistency maintenance, and produces significant improvements in lookup efficiency and replica hit rate. Its low overhead and high effectiveness are particularly attractive to the deployment of large-scale P2P file sharing systems.

In our future work, we plan to explore adaptive methods to fully exploit file popularity and update rate for efficient consistency maintenance. We will use a real trace of file replication and updates for experiments. We will also work on the extenuation of SWARM for cloud computing. Cloud computing (such as Amazon EC2, Google Cloud Platform) is a scalable service platform that consists of globally distributed data centers. Applications of file sharing or file storage, such as Dropbox, operate on the cloud platform. We can organize the virtual machines involved to a P2P overlay, and apply SWARM's file replication and consistency maintenance methods to save the storage and traffic load of cloud services as well as reduce the service latency to tenants.

Acknowledgements

This research was supported in part by U.S. NSF grants NSF-1404981, IIS-1354123, CNS-1254006, CNS-1249603, CNS-1049947, CNS-0917056 and CNS-1025652, Microsoft Research Faculty Fellowship 8300751.

References

[1] J. S. Otto, M. A. Sanchez, D. R. Choffnes, F. E. Bustamante, and G. Siganos. On Blind Mice and the Elephant-Understanding the Network Impact of a Large Distributed System. In *Proc. of SIGCOMM*, 2011.

[2] Bittorrent traffic increases 40% in half a year. http://torrentfreak.com/bittorrent-traffic-increases-40-in-half-a-year-121107/ [Accessd in Oct. 2014].

[3] Cisco visual networking index: Forecast and methodology, 2013c2018. Technical report, Cisco, 2014. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/ip-ngn-ip-next-generation-network/white_paper_c11-481360.pdf, [Accessd in Nov. 2014].

[4] H. Chen, H. Jin, X. Luo, Y. Liu, T. Gu, K. Chen, and L. M. Ni. BloomCast: Efficient and Effective Full-Text Retrieval in Unstructured P2P Networks. *TPDS*, 2012.

[5] S. Legtchenko, S. Monnet, P. Sens, and G. Muller. RelaxDHT: A Churn-Resilient Replication Strategy for Peer-to-Peer Distributed Hash-Tables. *ACM TAAS*, 2012.

[6] V. Ramasubramanian and E. Sirer. Beehive: The Design and Implementation of a Next Generation Name Service for the Internet. In *Proc. of ACM SIGCOMM*, 2004.

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

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

[9] V. Martin, P. Valduriez, and E. Pacitti. Survey of Data Replication in P2P systems. Technical Report inria-00122282, 2007.

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

[11] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *Proc. of ICDCS*, 2004.

[12] Swarm Intelligence. http://www.sce.carleton.ca/netmanage/tony/swarm.html [Accessd in Oct. 2014].

[13] X. Chen, S. Ren, H. Wang, and X. Zhang. SCOPE: scalable consistency maintenance in structured P2P systems. In *Proc. of INFOCOM*, 2005.

[14] Z. Li, G. Xie, and Z. Li. Efficient and scalable consistency maintenance for heterogeneous peer-to-peer systems. *TPDS*, 2008.

[15] Y. Hu, M. Feng, and L. N. Bhuyan. A Balanced Consistency Maintenance Protocol for Structured P2P Systems. In *Proc. of INFOCOM*, 2010.

[16] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proc. of ICDCS*, 2003.

[17] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. 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.

[18] H. Shen. Efficient and Effective File Replication in Structured P2P File Sharing Systems. In *Proc. of P2P*, 2009.

[19] H. Shen and G. Liu. A lightweight and cooperative multi-factor considered file replication method in structured P2P systems. *TC*, 2012.

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

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

[22] D. Rubenstein and S. Sahu. Can Unstructured P2P Protocols Survive Flash Crowds? *TON*, (3), 2005.

[23] H. Shen and G. Liu. A geographically-aware poll-based distributed file consistency maintenance method for P2P systems. *TPDS*, 2012.

[24] H. Shen. IRM: Integrated File Replication and Consistency Maintenance in P2P Systems. *TPDS*, 2010.

[25] J. Xiong, Y. Hu, G. Li, R. Tang, and Z. Fan. Metadata Distribution and Consistency Techniques for Large-Scale Cluster File Systems. *TPDS*, 2011.

[26] X. Tang and S. Zhou. Update Scheduling for Improving Consistency in Distributed Virtual Environments. *TPDS*, 2010.

[27] C. Hang and K. C. Sia. Peer Clustering and Firework Query Model. In *Proc. of WWW*, 2003.

[28] W. Nejdl, M. Wolpers, W. Siberski, A. Löser, I. Bruckhorst, M. Schlosser, and C. Schmitz. Super-peer-based routing and clustering strategies for rdf-based peer-to-peer networks. In *Proc. of WWW*, 2003.

[29] 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.

[30] R. Cuevas, N. Laoutais, X. Yang, G. Siganos, and P. Rodriguez. BitTorrent Locality and Transit Traffic Reduction: When, Why and at What Cost? *TPDS*, 2013.

[31] C. Decker, R. Eidenbenz, and R. Wattenhofer. Exploring and Improving BitTorrent Topologies. In *Proc. of P2P*, 2013.

[32] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: pay only when it matters. In *Proc. of VLDB*, 2009.

[33] J. Du, S. Elnikety, and A. W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *Proc. of SoCC*, 2013.

[34] H. Abu-Libdeh, R. V. Renesse, and Y. Vigfusson. Leveraging Sharding in the Design of Scalable Replication Protocols. In *Proc. of SoCC*, 2013.

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

[36] Z. Xu, M. Mahalingam, and M. Karlsson. Turning Heterogeneity into an Advantage in Overlay Routing. In *Proc. of INFOCOM*, 2003.

[37] T. S. Eugene Ng and H. Zhang. Towards global network positioning. In *Proc. of the 1st ACM SIGCOMM Workshop on Internet Measurement*, 2001.

[38] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier. Space Filling Curves and Their Use in Geometric Data Structure. *Theoretical Computer Science*, 181(1):3–15, 1997.

[39] Z. Xu, C. Tang, and Z. Zhang. Building topology-aware overlays using global soft-state. In *Proc. of ICDCS*, 2003.

[40] T. Li, Y. Lin, and H. Shen. A locality-aware similar information searching scheme. *Springer International Journal on Digital Libraries*, pages 1–15, 2014.

[41] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proc. of STOC*, pages 654–663, 1997.

[42] G. Liu, H. Shen, and L. Ward. An efficient and trustworthy p2p and social network integrated file sharing system. *Computers, IEEE Transactions on*, 64(1):54–70, 2015.

[43] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebooks Distributed Data Store for the Social Graph. In *Proc. of ATC*, 2013.

[44] Y. Hu, L. N. Bhuyan, and M. Feng. Maintaining Data Consistency in Structured P2P Systems. *TPDS*, 2012.

[45] PlanetLab. http://www.planet-lab.org [Accessd in Oct. 2014].

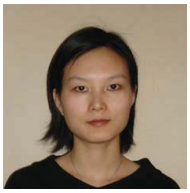
[46] C. Huang, J. Li, and K. W. Ross. Can internet video-on-demand be profitable. In *In Proc. of SIGCOMM*, 2007.

[47] BitTorrent User Activity Traces. http://www.cs.cmu.edu/∼pavlo/datasets/torrent/ [Accessd in Oct. 2014].

[48] C. Huang, J. Li, and K. W. Ross. Can internet video-on-demand be profitable? In *Proc. of SIGCOMM*, pages 133–144. ACM, 2007.

[49] A. Montresor and M. Jelasity. Peersim: A scalable P2P simulator. In *Proc. of P2P*, 2009.

[50] M. Jelasity, A. Montresor, G. P. Jesi, and S. Voulgaris. Peersim: A peer-to-peer simulator. http://peersim.sourceforge.net/ [Accessd in Oct. 2014].



Haiying Shen Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University, China in 2000, and the MS and Ph.D. degrees in Computer Engineering from Wayne State University in 2004 and 2006, respectively. She is currently an Assistant Professor in the Department of Electrical and Computer Engineering at Clemson University. Her research interests include distributed computer systems and computer networks, with an emphasis on P2P and content delivery networks, mobile computing, wireless sensor networks, and cloud computing. She is a Microsoft Faculty Fellow of 2010, a senior member of the IEEE and a member of the ACM.

Guoxin Liu Guoxin Liu received the BS degree in BeiHang University 2006, and the MS degree in Institute of Software, Chinese Academy of Sciences 2009. He is currently a Ph.D. student in the Department of Electrical and Computer Engineering of Clemson University. His research interests include Peer-to-Peer, CDN and online social networks.

Harrison Chandler Harrison Chandler received his BS degree in Computer Engineering from Clemson University in 2012. He is currently a Ph.D. student in the Department of Electrical Engineering and Computer Science at University of Michigan. His research interests include embedded and distributed systems.