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 Geographically-aware Poll-based Distributed File Consistency Maintenance Method for P2P Systems

Haiying Shen*, *Member, IEEE*, Guoxin Liu

Abstract—File consistency maintenance in P2P systems is a technique for maintaining consistency between files and their replicas. Most previous consistency maintenance methods depend on either message spreading or structure-based pushing. Message spreading generates high overhead due to a large amount of messages; structure-based pushing methods reduce this overhead. However, both approaches cannot guarantee that every replica node receives an update in churn, because replica nodes passively wait for updates. As opposed to push-based methods that are not effective in high-churn and low-resource P2P systems, polling is churn-resilient and generates low overhead. However, it is faced with a number of challenges: (1) ensuring a limited inconsistency; (2) realizing polling in a distributed manner; (3) considering physical proximity in polling; and (4) leveraging polling to further reduce polling overhead. To handle these challenges, this paper introduces a poll-based distributed file consistency maintenance method called Geographically-aware Wave (GeWave). GeWave further reduces update overhead, enhances the fidelity of file consistency, and takes proximity into account. Using adaptive polling in a dynamic structure, GeWave avoids redundant file updates and ensures that every node receives an update in a limited time period even in churn. Furthermore, it propagates updates between geographically close nodes in a distributed manner. Extensive experimental results from the PlanetLab real-world testbed demonstrate the efficiency and effectiveness of GeWave in comparison with other representative consistency maintenance schemes. It dramatically reduces the overhead and yields significant improvements on effectiveness, scalability and churn-resilience of previous file consistency maintenance methods.

Keywords: Consistency maintenance, File replication, Peer-to-peer systems, Proximity, Churn.

1 Introduction

Over the past several years, the immense popularity of the Internet has produced a significant stimulus to peer-to-peer (P2P) computer networks. A P2P computer network uses the cumulative resources of network participants, rather than conventional centralized resources, in which a relatively low number of servers provide services to clients. P2P systems enable peers to find data without relying on a centralized index server. File sharing is one of the most popular applications of P2P systems. Users who engage in P2P file sharing on the Internet both provide and receive files in a decentralized manner. It was indicated that P2P content responds well to caching because it has high reuse patterns and most P2P content is requested multiple times. Thus, file replication and caching have been studied [1–21] to improve P2P file sharing system performance. Shared files are replicated on a number of nodes to improve system reliability and availability. Moreover, query results are always cached along the routing path to reduce the querying latency of subsequent queries.

Although files in some P2P file sharing systems (i.e. KaZaA [22] and Morpheus [23]) are always consistent, other P2P systems (i.e., OceanStore [24] and Publius [25]) permit users to modify their files, leading to inconsistency between a modified file and its replicas [26]. In addition, with the tremendous development of P2P applications, newly-developed P2P applications require frequent content updates, such as trust management [27], remote collaboration, bulletin-board systems, and e-commerce catalogues. In these applications, files are frequently and regularly changed. Therefore, maintaining consistency between a file and its replicas is needed for correct operation of a P2P system so that the service quality of existing P2P applications is improved and the basic requirements of newly-developed P2P applications are met. Without effective consistency maintenance, a P2P system is limited to providing only static or infrequently-updated file sharing [26].

The number of nodes in a P2P system usually varies from thousands to millions, and the nodes are scattered across geographically distributed areas. Also, P2P systems are characterized by churn, in which many nodes join, leave and fail continuously and rapidly. Thus, a scalable and churn-resilient consistency maintenance method is demanded. In addition, the method is required to achieve the following objectives:

(1) High fidelity of consistency of queried results. It is important that every replica node can receive the update message in a limited time, so that a file requester can receive up-to-date files with high probability.
(2) Low overhead. Efficient consistency maintenance methods do not generate high overhead, which would otherwise become an obstacle for achieving high scalability.

Centralized schemes, as employed in [28, 10] and current Facebook datacenters, are a straightforward way to maintain replica consistency. In these methods, a file owner keeps track of the replica nodes and notifies the nodes every time the file is changed. However, these lack scalability and suffer from single-points of failure. Some decentralized consistency maintenance methods rely on message spreading [29, 30], in which

- * 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

Digital Object Indentifier 10.1109/TPDS.2012.317 1045-9219/12/$31.00 © 2012 IEEE

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 update message is flooded in the network. This method generates very high overhead because of the overwhelming number of messages. To reduce overhead, other decentralized methods depend on structures [26, 31–33] to propagate updates by pushing so that each node receives only one message. However, node failures may break the structure, leading to unsuccessful update propagation. For example, in a tree structure [31], if a parent fails, all of its descendents are not able to receive the update message. In addition to the update failures, these methods generate high overhead for structure maintenance. Nodes need to update their neighbors in the structure to deal with replica node creation and deletion, especially in churn. In both message spreading and structure-based decentralized pushing approaches, because replica nodes passively wait for updates, they do not know when they fail to receive updates due to churn. Therefore, these approaches cannot guarantee that every replica node receives an update in churn.

Push-based methods are advantageous for real-time updates, i.e., replica nodes receive an update soon after the file is updated. In a P2P system with high churn that cannot afford high overhead and does not require real-time updates, polling is more suitable for file consistency maintenance as it is churn-resilient and generates low overhead. By "churn-resilient", we mean that nodes can be aware that they have not received update in time due to churn and then actively obtain the update by polling. For example, a principle for scaling eBay's distributed system is "embracing inconsistency" [34]. However, a poll-based method faces a number of challenges: (1) ensuring a limited inconsistency (i.e., high fidelity of consistency); (2) realizing polling in a distributed manner; (3) considering physical proximity in polling; and (4) leveraging polling to further reduce overhead.

To handle these challenges, this paper introduces a poll-based distributed file consistency maintenance method called Geographically-aware Wave (GeWave). GeWave further reduces update overhead and enhances the fidelity of file consistency, while considering proximity. In GeWave, each replica node has a polling frequency for its replica that ensures replica consistency with high probability when requested. GeWave builds a tree structure, in which a child's parent has a higher polling frequency and is geographically close to the child. In an update process, children poll their parents from the top to the bottom, level by level. Due to its wavy pattern among geographically close nodes, we name this method as geographically-aware wave. GeWave also has a lightweight structure maintenance algorithm to resiliently deal with P2P churn and replica node creation and deletion.

Most previous methods aim to inform all replica nodes soon after a file is updated. The ultimate goal of consistency maintenance is to guarantee that query results are not out-of-date. Therefore, rather than trying to achieve strong consistency of replicas all the time, GeWave aims to achieve the consistency of query results with high probability by avoiding unnecessary updates. A significant feature of GeWave is that it achieves an optimized tradeoff between overhead and fidelity of consistency. Specifically, GeWave possesses the following distinguishing features.

(1) *High fidelity of consistency.* It uses adaptive polling to ensure that all replica nodes receive updates, and to achieve high consistency fidelity of queried results with high probability.
(2) *Low overhead.* It conducts update propagation between geographically close nodes. Also, it avoids redundant file updates by dynamically adapting to time-varying file update and query rates, and it generates much lower overhead for structure maintenance.
(3) *High scalability.* It conducts consistency maintenance in a decentralized manner rather than completely relying on the file owner, and a node receives updates directly without relay nodes.
(4) *Churn-resilience.* It ensures that a replica node can always receive an update, even in churn.

GeWave is proposed for a high-churn and low-resource system that can tolerate inconsistency for a limited time period (i.e., minimum update interval), and in which the query rate of replicas varies in a certain range. Though GeWave can adjust its file polling frequency in response to the file update rate and query rate changes, it is not suitable when a replica file's update or query rate experiences significant fluctuations very frequently and abruptly. In such a scenario, GeWave may not able to adapt to the changes quickly.

2 Related Work

Message spreading. One class of consistency maintenance methods is based on message spreading. Lan *et al.* [30] proposed the use of flooding-based push for static files and polling for dynamic files. In the hybrid push/pull algorithm [29], flooding is replaced with rumor spreading to reduce communication overhead. A replica node randomly selects a set of replica nodes and forwards the message to them with a probability. In order to reduce redundancy and help replica nodes discover replicas unknown to them, a partial list of replica nodes to which the same message has been sent is enclosed with the update message. The update message will not be forwarded to the replica nodes in this partial list. When a new node joins or a node reconnects, it contacts online replica nodes to poll updated content. However, the hybrid push/pull scheme only offers probabilistic guarantee of replica consistency.

While suitable for unreliable P2P systems with churn, message spreading methods generate high propagation overhead due to a large number of update messages. They also cannot guarantee that an update message reaches every replica node. In addition, the scheme does not consider proximity in message spreading. In contrast, GeWave generates low overhead by relying on a GeWave structure comprised of only replica nodes. It can guarantee that each replica node receives the update by polling. Further, it considers proximity in update message transmission.

Structure-based pushing. Another class of methods relies on push-based structures. Li *et al.* [26] presented a scheme that forms the replica nodes into a proximity-aware hierarchical structure (UMPT): the upper layer is a distributed hash table (DHT), and a node in the lower layer attaches to a physically close node in the upper layer. An update tree is built dynamically when the upper layer propagates update messages. Though it takes proximity into account, clusters of

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.

physically close nodes are fragile in churn, which may lead to update propagation failures. Moreover, periodical message exchange for cluster maintenance leads to high overhead. Further, it may not be effective in a network where nodes are widely scattered without a clustering pattern. Also, it is not completely decentralized since it relies on the high-capacity nodes in the upper-layer for update propagation. Further, the update propagation along the tree is not locality-aware. SCOPE [31] builds a replica-partition-tree for each key based on its original P2P system. It keeps track of the locations of replicas and then propagates updates. SCOPE introduces three operations to maintain consistency: subscribe, unsubscribe and update. Hu *et al.* [35] presented a framework for balanced consistency maintenance (BCoM) in structured P2P systems. Replica nodes of each object are organized into a tree for disseminating updates, and a sliding window update protocol is developed to bound the consistency. The window enables BCOM to balance availability, performance and consistency strictness for various application requirements. One group of methods propagate updates along routing paths. CUP [32] 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. CUP has performance limitations because it pushes the update along the query path, and intermediate nodes along the path receive the updated index even if they do not need it. To improve CUP, Yin and Cao proposed Dynamic-tree based Update Propagation (DUP) scheme [33], which builds a dynamic update propagation tree that consists of nodes interested in update. In Freenet [21], an update is routed to other nodes based on key closeness.

The structure-based pushing methods also have a number of problems. First, decentralized pushing cannot guarantee all replica nodes receive the update with node departures and failures, hence it cannot always ensure high fidelity of consistency of queried results. Second, proximity unawareness in most methods prevents further improvement in consistency maintenance efficiency. In contrast, GeWave can guarantee that replica nodes receive updates by polling other nodes or the file owner. It also achieves proximity-aware consistency maintenance by enabling nodes to poll their geographically close nodes. Compared to UMPT, GeWave is fully decentralized and fully locality-aware. Also, GeWave is not limited by the physical topology of a P2P network.

Push-based methods are advantageous in real-time update propagation in low-churn systems while GeWave is proposed for high-churn and low-resource systems that can tolerate inconsistency for a limited time period. GeWave considers both proximity and churn in order to provide highly efficient and reliable consistency maintenance. With active polling, it provides high consistency fidelity even in churn and reduces structure construction and maintenance cost. GeWave shares similarity with the works in [36, 37, 30] in terms of polling employment and polling rate determination. However, the novelty of GeWave lies in the decentralized and proximity-aware polling.

3 Overview of GeWave

In GeWave, each replica node has a periodical polling time interval for its replica, denoted by time-to-refresh (TTR). GeWave forms replica nodes of a file into a structure based on their TTR values and their physical locations. An example of the GeWave structure is shown in Figure 1. The root node is the file owner, and the replica nodes are organized in ascending order of their TTR levels from the top to the bottom. That is, the first level nodes are the nodes that have the least TTR level, and the TTR level of the dth level nodes is smaller than that of the $(d+1)^{th}$ level nodes, i.e., the polling rate of the nodes in the dth level is faster than that of the nodes in the $(d+1)^{th}$ level. The children are physically close to their parents. In each level, a node connects to its predecessor and successor (sibling neighbors), which are physically close to the node.

In GeWave, rather than polling the file owner, each replica node polls its parent for an update. The file update is conducted in the fashion of a wave. After the original file is updated, the nodes in the first level poll the file owner for updates before their replicas are queried. Later, the nodes in the second level poll the nodes in the first level before their replicas are queried, and so on. Therefore, a file's updating process is like a wave from the top to the bottom between geographically close nodes. In this way, GeWave is distinguished by a number of features: (1) Depending on active polling rather than pushing, GeWave enhances the fidelity of consistency. (2) Polling within geographically close nodes dramatically improves the efficiency of consistency maintenance. (3) Relying on decentralized polling instead of centralized polling, GeWave achieves a high scalability by distributing the overhead among replica nodes. (4) Compared to the structure-based pushing methods, GeWave avoids update failures due to node departures and failures and helps to ensure file consistency, even in churn. Also, the direct node-to-node communication enables distant nodes to achieve consistency and is less sensitive to churn, network size and P2P structure. Specifically, GeWave addresses the following challenges:

(1) How can the GeWave structure be constructed to considering file update and query rates and node physical proximity?
(2) How can consistency maintenance in the GeWave structure be conducted, in a decentralized manner?
(3) How can the GeWave structure be maintained in churn in order to enhance the fidelity of file consistency?

Replica nodes are allowed to modify the file but need to notify the file owner. Techniques for file replication are orthogonal to our study in this paper. As previously mentioned, though GeWave can adjust its file polling frequency in response to the file update rate, it is not suitable for a scenario where a replica file's query rate or update rate experiences significant fluctuations very frequently and abruptly. P2P systems with relatively stable update and query rates are not uncommon. Gnutella [38] shows that the popularity of transiently popular files is stable over a short time period and the popularity of persistently popular files remains stable. The work in [39] assumes that a file's update rate (i.e., number of updates over a certain period of time) is relatively stable. A scenario with

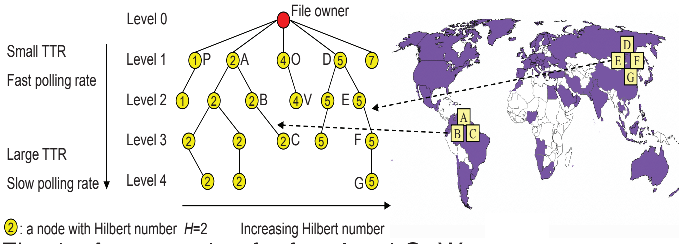


Fig. 1. An example of a four-level GeWave structure.

relatively stable query and update rates in a certain time period can be found in a P2P trust management system [27], an e-commerce catalogue and a file sharing system where nodes' activities are relatively stable in a certain time period.

4 The Design of GeWave

4.1 Adaptive and Lightweight Polling

GeWave uses adaptive polling, in which a replica node polls a node with an updated file to determine if its replica is stale. GeWave is developed based on the work in [37] that considers cases when replica nodes directly poll the file owner for updates. Specifically, GeWave uses the strategy in the work to determine the polling frequency. We describe this strategy in the following.

Consider a file's maximum update rate is $1/\triangle$, which means it updates every $\triangle$ time units, say seconds, in the highest update frequency. Therefore, a replica node can ensure that a replica is never outdated by more than $\triangle$ seconds by simply polling the owner every $\triangle$ seconds. A TTR value is associated with each replica that represents the time between two successive polls. We use $\triangledown$ to denote the lowest update frequency. A file owner can find $\triangle$ and $\triangledown$ based on its own historical updating activities. When it replicates its file to other nodes, it also sends $\triangle$ and $\triangledown$ to the replica nodes. The polling algorithm begins by initializing $TTR = TTR_{min} = \triangle$ and $TTR_{max} = \triangledown$. It then uses a linear increase multiplicative decrease algorithm (LIMD) [36] to adapt the TTR value (and hence, the polling frequency) to the file update rate. Particularly, the TTR value is increased by a linear factor (resulting in less frequent polls) if the file does not change between successive polls. That is,

$$TTR = TTR + a \ (a > 0), \tag{1}$$

where a is an additive constant. Otherwise, the TTR value is reduced by a multiplicative factor (causing more frequent polls). That is,

$$TTR = TTR/b \ (b > 1), \tag{2}$$

where b is the multiplicative decrease constant. The parameters of a and b are determined by the real system file updating frequency and the requirement of the tradeoff of polling overhead and the fidelity of consistency maintenance. A system with a higher file update frequency should have a larger b and smaller a, while a system with a lower file update frequency should have a larger a and smaller b. Also, larger a or smaller b leads to lower polling overhead but lower fidelity of consistency maintenance and vice versa. Considering the various factors to determine a and b is a non-trivial task. However, it is not the focus of this work and we leave it as our future work. The algorithm takes as input two parameters: $TTR_{\min}$ and $TTR_{\max}$, which represent lower and upper bounds on the TTR values. Values that fall outside these bounds are set to

$$TTR = \max(TTR_{\min}, \min(TTR_{\max}, TTR)). \tag{3}$$

This ensures that the TTR computed by the algorithm is neither too large nor too small. Thus, each replica node probes the file owner at a rate at which the file changes and sets the TTR value accordingly.

The ultimate objective of file consistency maintenance is to ensure a replica is not outdated when requested. Therefore, it is not necessary to update all file replicas once a file is changed. Thus, the file query rate is further considered in TTR determination in order to reduce the overhead of consistency maintenance. That is,

$$TTR = \begin{cases} T_{query} & TTR \leq T_{query} \\ TTR & TTR > T_{query}, \end{cases} \tag{4}$$

where T_{query} denotes the time interval between two successive queries. A replica node periodically calculates T_{query} of its replica. For example, during a time period, a replica is visited m times with T_{query_i} $(1 \leq i \leq m)$. Then, $T_{query} = \sum_{i=1}^{m} T_{query_i}/m$. When $TTR \leq T_{query}$, that is, when the file update rate is higher than the file query rate, there is no need to update the replica at the update rate. On the other hand, when $TTR > T_{query}$, that is, the replica is visited at a higher rate than its update rate, then the replica should be updated based on TTR. This polling frequency determination method avoids unnecessary overhead for file updates while achieving high fidelity of consistency of queried results.

4.2 Geographically-aware Wave

All replica nodes polling a file owner for consistency maintenance may overload the file owner, leading to delayed update message response. The TTR-based and proximity-aware GeWave structure conducts file consistency maintenance in a decentralized manner.

Before we present the details of the GeWave structure, let us introduce a landmarking method to represent node closeness on the Internet by indices. Landmark clustering has been widely adopted to generate proximity information [40, 41]. It is based on the intuition that nodes close to each other are likely to have similar distances to a few selected landmark nodes, although details may vary from system to system. We assume m landmark nodes that are randomly scattered in the Internet. Each node measures its physical distances to the m landmarks, and uses the vector of distances $< d_1, d_2, \ldots, d_m >$ as its coordinate in Cartesian space. Two physically close nodes have similar landmark vectors. Note that a sufficient number of landmark nodes are needed to reduce the probability of false clustering where nodes that are physically far away have similar landmark vectors. We use space-filling curves [42], such as Hilbert curve [41], to map m-dimensional landmark vectors to real-numbers, called Hilbert numbers. As a result, physically close nodes will have similar Hilbert numbers. We use H_i to denote the Hilbert number of node i.

A GeWave structure is built dynamically based on the node proximity and polling rate. GeWave takes the file owner as its root and places nodes into different levels based on polling rate levels with nodes in the upper levels having higher polling frequencies than those in the lower levels. A GeWave

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.

structure has L levels, where L is determined by the file owner according to the actual file query rate, TTR distribution and an estimated number of replica nodes. L should be set to an appropriate value. If L is too large, it will adversely affect the effectiveness of the GeWave structure for file updating. If it is too small, it may overload some nodes with many children. A real trace from Gnutella [38] shows that the file query rate can be modeled as a Zipf distribution. Therefore, TTR values are most likely to be varied in a range based on Formula (4). No matter whether the TTR values are varied in a range or not, the file owner partitions the entire TTR range and assigns each partition to each tree level so that the replica nodes are not distributed in different levels in imbalance (i.e., too many replica nodes are assigned to a level while very few replica nodes are assigned to another level). The entire TTR range does not have to be partitioned evenly, more even TTR distribution leads to move even partitions. We use L_{TTR} to denote a node's TTR level. For instance, if the possible TTR is within [0,100] seconds, $L = 10$ and each level evenly holds 10 seconds, then the nodes with TTR in [0,10) are on level 1, the nodes with TTR in [10,20) are on level 2, and so on. The TTR range on level 2 can be [10,30] if there are few replica nodes with TTR in [10,20) or [20,30). After replica nodes join in the tree, their TTRs may change over the time leading to an imbalanced distribution of nodes in different tree levels. In order to avoid this problem, if many replicas' TTRs become very close, the file owner further partitions this TTR range into m levels. When a replica in this TTR range is created, the file owner assigns this replica to one of m levels that has the fewest nodes. If there are very few nodes in a level, the file owner can combine the range of this level with the range in the level above or below. After the tree reconstruction, the file owner notifies corresponding nodes to change their neighbors in the tree. To further achieve load balance, in Section 7.2, we will introduce a load balancing algorithm that avoids overloading a replica node due to many connected children.

As shown in Figure 1, the root node is the file owner, and the replica nodes are organized in ascending order of their TTR levels from the top to the bottom. That is, the first level nodes are the nodes that have the least TTR level, and the TTR level of the dth level nodes is smaller than that of the $(d+1)^{th}$ level nodes. In other words, the polling rate of nodes in the dth level is faster than that of nodes in the $(d+1)^{th}$ level.

GeWave uses the Hilbert number [41] to represent node proximity closeness. Physically close nodes have similar Hilbert numbers. We use H_i to denote the Hilbert number of node i. In one level, nodes are arranged in ascending order of their Hilbert numbers. Nodes in the $(d+1)^{th}$ level choose the geographically closest nodes in the dth level as their parents. In one level, a node connects to its predecessor and successor, which are relatively physically close neighbor(s) whose polling rate is at the same level as itself. The neighbor links help to enhance the efficiency of structure maintenance for replica node joins and departures. As a result, a node has links to its parent, two neighbors (i.e., predecessor and successor) and its children. The root node does not have a parent link and the first node and last node in each level only have one neighbor.

In GeWave, rather than polling the file owner, each replica node polls its parent for an update. For example, in Figure 1 nodes A, B and C are physically close nodes, and node A is relatively close to its neighbors P and O. During file updating, after A polls the file owner, B polls A for an update, and after that, C polls B for an update. A file has a new version number after it is updated. When a child polls its parent for an update, it also piggybacks its own version number on the request. If the parent finds that the piggybacked version is the same as its own version number, i.e., it has not received latest update, it does not respond until it has received a new update with a newer version number. If a file is not updated at its update rate within a time period, the file owner replies to the polling node that the file has not been updated during this update period. The parents subsequently reply to their children that the file has not updated during this update period. This policy prevents nodes from waiting for an update infinitely before replying to file requests.

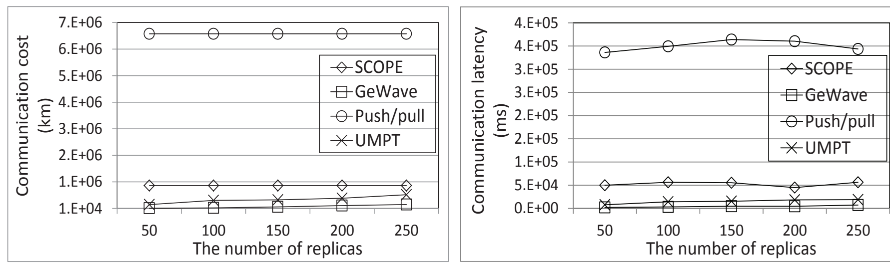
The parent assignment algorithm in GeWave ensures that a child's parent has the most up-to-date file when the child polls it for an update and that child is physically close to its parent. Communication between geographically close nodes reduces bandwidth consumption and improves efficiency. From the perspective of the entire GeWave structure, the update process is modelled as a wave from the top to the bottom between geographically close nodes, ensuring nodes receive updates timely and efficiently. Such a decentralized update pattern distributes updating overhead among replica nodes, thus significantly improving the scalability of the polling method.

Section 7 in the supplementary file presents other mechanisms of GeWave including those for structure construction and maintenance, reliability, and load balancing.

5 Performance Evaluation

We designed and implemented the systems on the real-world PlanetLab testbed [43]. We compared the performance of *GeWave* with the *SCOPE* [31], *UMPT* [26] and *Push/pull* [29] methods. We chose *SCOPE* as a representative of system-wide structure-based pushing methods that constitute all nodes in a P2P system to a structure for consistency maintenance. We chose *UMPT* as a representative of per-file structure-based proximity-aware pushing methods that bulids all replica nodes of a file into a system for consistency maintenance and propagate updates between geographically close nodes. We chose *Push/pull* as a representative of message spreading methods. In *UMPT* and *SCOPE*, we set the k in the propagation k-nary tree to 2. Below, we briefly introduce each of the methods.

***SCOPE* [31].** By building a replica-partition-tree (RPT), SCOPE keeps track of the locations of replicas and then propagates update notifications. Specifically, each node has a consistent hash value (e.g. SHA-1) [44] as its m-bit identifier. RPT is built by recursively splitting the identifier space. The primary node of a key in the original identifier space is the root of RPT. The representative node of a key in each partition, recording the locations of replicas at the lower levels, becomes one intermediate node of RPT. The leaves of RPT are those representative nodes at the bottom level. When an update is propagated from the root to the bottom, an intermediate node decides



(a) Communication cost in distance (b) Communication cost in latency

Fig. 2. Communication cost.

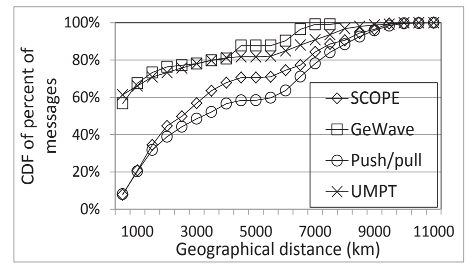


Fig. 3. Proximity-aware performance.

whether it needs to forward the update to each of its child partition. In the experiments, we built one RPT in a P2P system.

***UMPT* [26].** UMPT is a proximity-aware hierarchical structure. Its upper layer is P2P-based and consists of powerful and stable replica nodes, while a replica node at the lower layer attaches to a physically close upper layer node. When a file is updated, a k-ary update message propagation tree (UMPT) is built dynamically and the update is propagated in the top-down manner. Each upper-layer node also forwards the update to its connected lower-layer nodes. Nodes periodically exchange messages to maintain the connections between lower-layer nodes and physically close upper-layer nodes.

***Push/pull* [29].** In this hybrid push/pull algorithm, the update messages are propagated in a rumor spreading style. A replica node randomly selects a set of replica nodes and forwards the message to them with a probability. In our experiments, a node selects its neighbors with probability 1 to spread the updates. In order to reduce the redundancy and help replica nodes discover replicas unknown to them, a partial list of replica nodes to which the same message has been sent is enclosed with the update message. The update message will not be forwarded to the replica nodes in this partial list.

In the PlanetLab experiments, we randomly chose 256 nodes on PlanetLab worldwide, and distributed 10 files (each of which is 1MB) to each node. We randomly chose 15 landmark nodes all over the world, and used the method introduced in Section 4.2 to generate the proximity of each node. Each node randomly chose nodes from the other 255 nodes to replicate its files. The size of an update was set to a value randomly chosen from [1,5]Kb. The file query rate of each replica file was randomly generated from [0,100]s. Considering the small scale of this P2P network, we did not limit the TTL of update transmission hops in *Push/pull*. Unless otherwise specified, each file has 100 replicas.

5.1 Consistency Maintenance Cost

Communication cost constitutes a major part of file consistency maintenance overhead. The cost is directly related to the update message size and physical path length or latency of the message travelled. We use the produce of these two factors of all messages to represent the communication cost.

5.1.1 Single File Update

We first test the communication cost of one file's consistency maintenance. In this experiment, the file update rate was set to 20s. In *Push/pull*, each node pushes an update to 18 neighbors, 8 neighbors in its finger table and 10 in its successor list. Figure 2(a) and Figure 2(b) show the average update communication cost measured by distance and latency for all replicas in every 20s during the total 500s with different number of replicas, respectively. We see that the result follows *Push/pull*>*SCOPE*>*UMPT*>*GeWave*. From the figures, we can see that *Push/pull* generates prohibitively high cost. In *Push/pull*, upon receiving an update message, a node forwards the update to its 18 neighbors. Thus, the number of messages increases exponentially. In addition, *Push/pull* does not consider proximity in message pushing and a node's neighbors may be physically far away from itself. Therefore, a large amount of messages travel long distances in file updates, leading to a dramatically high cost. We also observe that *GeWave* and *UMPT* incur much lower communication cost than *SCOPE*. Because *SCOPE* propagates an update along a tree consisting of all nodes in the system, though some nodes are not replica nodes, the considerably larger number of messages increases the communication cost. Also, *SCOPE* does not take proximity into account in update propagation, resulting in long message travel distances. Furthermore, we can see that the communication cost of *Push/pull* and *SCOPE* remains almost constant regardless of the number of replica nodes. This is because *Push/pull* employs rumor spreading and *SCOPE* propagates the update among all system nodes.

In contrast, *GeWave* and *UMPT* constrain the propagation scope only within replica nodes. Also, they consider proximity by guiding update messages to travel between physically close nodes. Therefore, *GeWave* and *UMPT* generate significantly lower communication costs than *Push/pull* and *SCOPE*, and their communication costs increase as the number of replicas grows. We observe that *GeWave* produces lower costs than *UMPT*. This is because *GeWave* does not update replicas if they are visited less frequently than their update rates. This result shows the advantage of relaxing the strict consistency requirement that all replicas should be updated once the source file is updated. the result confirms the low overhead and latency of *GeWave* in consistency maintenance measured by the message transmission cost in the real-world testbed.

5.1.2 Proximity-aware Performance

This experiment shows the proximity-aware performance to map physically close nodes for communication. Figure 3 shows the proximity-aware performance in propagating update messages between physically close nodes. It shows that more update propagations in *GeWave* are within short distances than in other methods. Specifically, there are 88%, 82%, 71%, and 57% of all updates that are within 5000km in *GeWave*, *UMPT*, *SCOPE* and *Push/pull*, respectively; and

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.

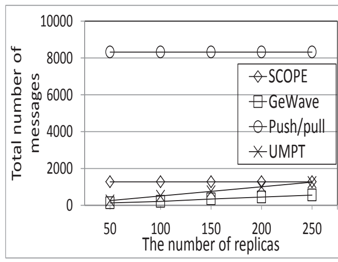
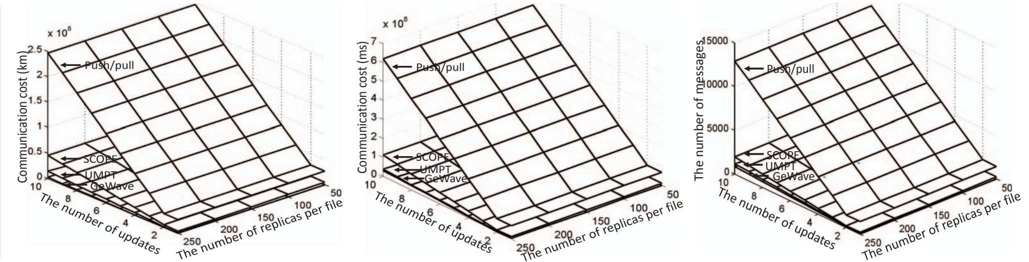


Fig. 4. Total number of update messages.



(a) Communication cost in distance (b) Communication cost in latency (c) The number of update messages

Fig. 5. Performance of file consistency maintenance schemes.

there are 99%, 91%, 84%, and 78% of all updates that are within 7000km in *GeWave*, *UMPT*, *SCOPE* and *Push/pull*, respectively. This result confirms that *GeWave* and *UMPT* have higher proximity-aware performance than *SCOPE* and *Push/pull*, and that *GeWave* outperforms *UMPT* in propagating more updates between physically close nodes when nodes do not form clusters in a real-world network due to the same reasons as in Section 5.1.1. The result confirms that *GeWave* is effective in guiding update messages to travel between physically close nodes, leading to fast and lightweight update propagation.

5.1.3 The Number of Generated Messages

Figure 4 demonstrates the total number of update messages for all replicas during 100s versus the number of replicas in the experiment. The result follows *Push/pull*>*SCOPE*>*UMPT*>*GeWave*. *Push/pull* and *SCOPE* are stable, and *UMPT* and *GeWave* increase in proportion to the number of replicas. The results are consistent with Figure 2(a) and Figure 2(b). In *Push/pull*, each node forwards an update message to 18 neighbors. Thus, *Push/pull* generates the highest number of messages in total for a file's update regardless of the number of replicas. *SCOPE* propagates the update messages along the tree formed by all nodes in the system. Therefore, *SCOPE* produces 255 messages for each file update regardless of the number of replicas. *GeWave* and *UMPT* propagate update messages only among replica nodes. Thus, they produce more update messages as the number of replicas increases. Further, *GeWave* leads to much fewer messages due to the same reason as in Figure 2.

5.1.4 Multiple File Updates

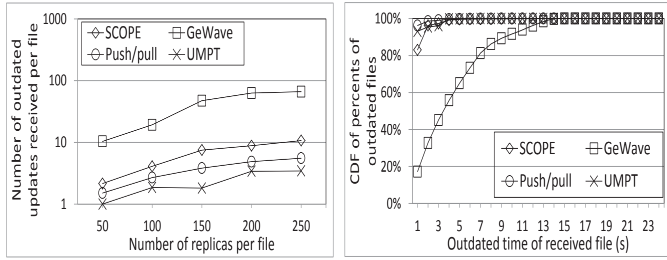
In this experiment, we consider 200 files that are classified into four categories, and the percentage of the files in each category and their update rates were set to (0.5%, 10s), (2.5%, 20s), (7%, 30s) and (90%, 50s). Figure 5(a), Figure 5(b) and Figure 5(c) show the total communication cost in distance and latency, and the number of file update messages versus the number of updates and number of replicas per file, respectively. In both figures, the results follow *Push/pull*>*SCOPE*>*UMPT*>*GeWave*. Also, *Push/pull* increases much faster than all other methods. We see that *UMPT* and *GeWave* exhibit much lower costs than *Push/pull* and *SCOPE*, the costs of which grows quickly with the number of updates. Comparing Figure 5(a) and Figure 5(b) and Figure 5(c) with Figure 2, we observe that *Push/pull* and *SCOPE* produce much higher communication costs in updating 200 files than updating 1 file. This is because they already incur very high costs for updating 1 file due to the involvement of all nodes in the system. Updating more files generates a cost that is multiple times higher. *UMPT* and *GeWave* incur much lower costs in updating 1 file. Then, updating more files brings about incremental cost growth. The figure demonstrates that the costs of *Push/pull* and *SCOPE* are not affected by the number of replicas per file. This is because no matter how many replicas a file has, *Push/pull* and *SCOPE* conduct the same operation involving all nodes in the system. The communication costs of *UMPT* and *GeWave* increase proportionally with the number of replicas per file and the number of updates, though it is not obvious in the figure. Recall that *UMPT* and *GeWave* only propagate update messages among replica nodes. Therefore, more replicas of a file or more updates generate more messages and hence higher communication costs. The three figures confirm the low cost and high efficiency of *GeWave*.

5.2 Fidelity of File Consistency

Recall that *GeWave* aims to ensure that each replica is never outdated by more than Δ time units. We define such a replica file as a soft up-to-date file, otherwise a soft outdated file. Let $v_0, v_1, v_2 \ldots$ denote a file's version after each update. After a file owner sends out an update for file version v_i, if a replica node provides the file with a version $v_j (j < i)$, the provided file is defined as a hard outdated file.

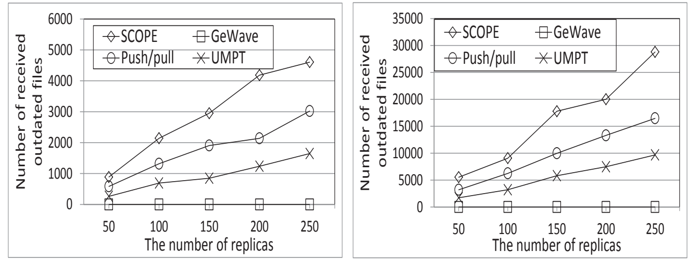
Since an update could occur at any time spot during the unit time interval, we randomly chose the time spot for each update in an update interval when evaluating the fidelity of consistency. Our experimental results show that all methods generate 0 soft outdated files received in a system without churn. Figure 6(a) shows the number of received hard outdated files. We see that the result follows *GeWave*>SCOPE>Push/pull>UMPT. *GeWave* aims to reduce soft outdated files rather than hard outdated files, hence it generates most hard outdated files.

The update propagation speed of the push-based methods follows SCOPE>Push/pull>UMPT. With faster speed, more replica nodes can receive updates in time, leading to less hard outdated files. To further investigate the delay in updating, we plot Figure 6(b) that shows the CDF of the received hard outdated files versus the outdated time, defined as the time difference between when the file is received and when the file is hard outdated for the first time. We see that the outdated time for most received hard outdated files in the push-based methods is within 5 seconds, while that of *GeWave* is within



(a) Received hard outdated files (b) CDF of received hard outdated files

Fig. 6. Performance in achieving goals of consistency.



(a) Slow file update (b) Fast file update

Fig. 7. Performance of the fidelity of consistency.

13 seconds. This result confirms that *GeWave* can always constrain the outdated time to a minimum update interval, and that the push-based methods are advantageous for real-time updates.

We then measure the fidelity of consistency of *GeWave* using the soft outdated files, and measure that of other methods using the hard outdated files. In Section 8.1 in the supplementary file, we measures the number of soft outdated files for all methods in churn, which shows the higher consistency fidelity of *GeWave* than other structure-based methods in churn. We measured the average number of received outdated files every 100s out of the total 1000s experiment time. Figure 7(a) shows this average number of received outdated files in 100s with different numbers of replicas per file. The result follows *SCOPE>Push/pull>UMPT>GeWave*. There are no received outdated files in *GeWave*, and the number of outdated files in the other three methods increases gradually. *UMPT* generates fewer outdated files than *SCOPE* because update propagation between physically close nodes in *UMPT* enables replicas to receive updates faster than in *SCOPE*. *SCOPE* produces fewer outdate files than *Push/pull* because *SCOPE* propagates updates faster along the tree. More replicas lead to longer latencies in update propagation, and consequently more outdated files received. Using polling, each node in *GeWave* knows whether it has the updated file and only responds with the updated file, leading to 0 received outdated files.

In order to see the fidelity of consistency provided by the schemes when most files are updated more frequently, we change the percentage of the files and their update rates to (90%, 10s), (7%, 20s), (2.5%, 30s) and (0.5%, 50s). Figure 7(b) shows the average number of received outdated files in 100s during all 1000s, which also follows *SCOPE>Push/pull>UMPT>GeWave*. There are also no received outdated files in *GeWave*, and the number of outdated files in the other three increases gradually. The reasons for the results are the same as explained in Figure 7(a). Comparing Figure 7(b) with Figure 7(a), we see that the number of received outdated files in each method increases significantly due to the faster update rate. *GeWave* still produces 0 received outdated files in fast update rate, which confirms its performance in maintaining high fidelity of file consistency.

GeWave is distinguished by lower communication costs for updates, and higher fidelity of soft consistency in churn. Also, it achieves better proximity-aware performance than *UMPT*. *Pull/push* provides high consistency fidelity but at a cost of high message spreading overhead. We present additional experimental results in Section 8 in the supplementary file.

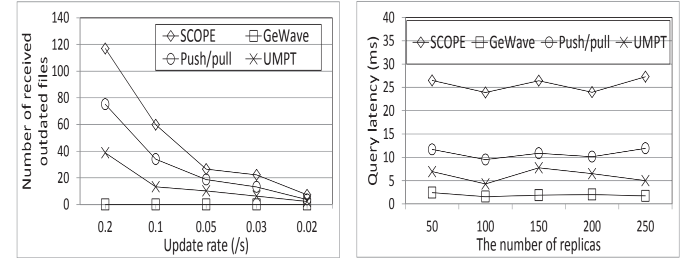


Fig. 8. Num. of received outdated files.

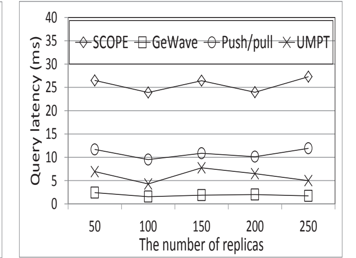


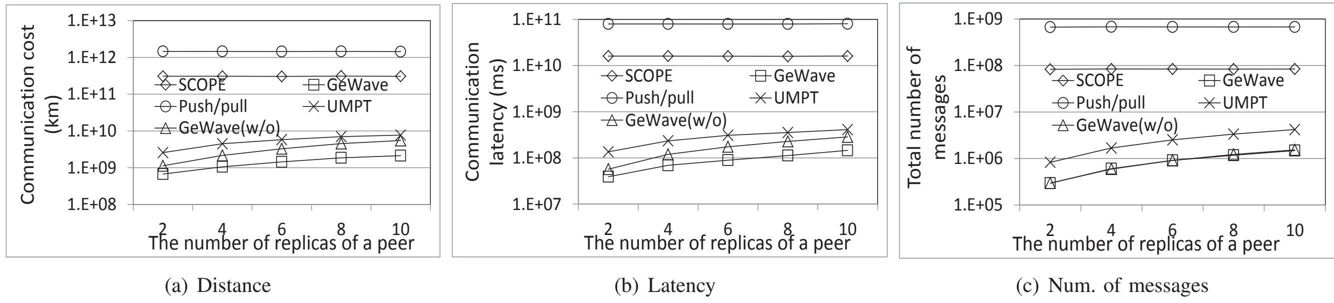
Fig. 9. Ave. query latency per query.

5.3 Different Update Rates

In this experiment, we evaluated the *GeWave* fidelity performance over different update rates of a file with one randomly chosen file owner and 100 replica nodes. We conducted 20 experiments, each lasting 100 seconds, and then calculated the average as the final result of each method. Figure 8 shows the number of outdated files received in different methods versus different update rates. The number of outdated files received follows *SCOPE>Push/pull>UMPT>GeWave* due to the same reason as Figure 7(a) and Figure 7(b). The figure also shows that the number of received outdated files decreases as the update rates decrease. Decreasing update rate reduces the number of updates during each 100 second period, thus generating fewer outdated files received in each method. The figure also shows that the decreasing speed follows *SCOPE>Push/pull>UMPT>GeWave*; one reason for this behavior is that longer update paths lead to longer latencies, and the average path length from a file owner to a replica node follows *SCOPE>Push/pull>UMPT*. Updates are simultaneously pushed to more children in *Push/pull* than in *SCOPE*, so that more nodes in *Push/pull* receive updates with short path lengths. *UMPT* clusters physically close nodes for one-hop updates to reduce the updating latency and builds the heads of clusters into a tree. In the tree, children are not physically close their parents. Thus, an update may be propagated along physically far-away nodes along the tree. In *GeWave*, children are physically close to their parent in the tree. Children poll their parents for updates and poll the file owner after a timeout. Therefore, a replica node can always provide a soft up-to-date file.

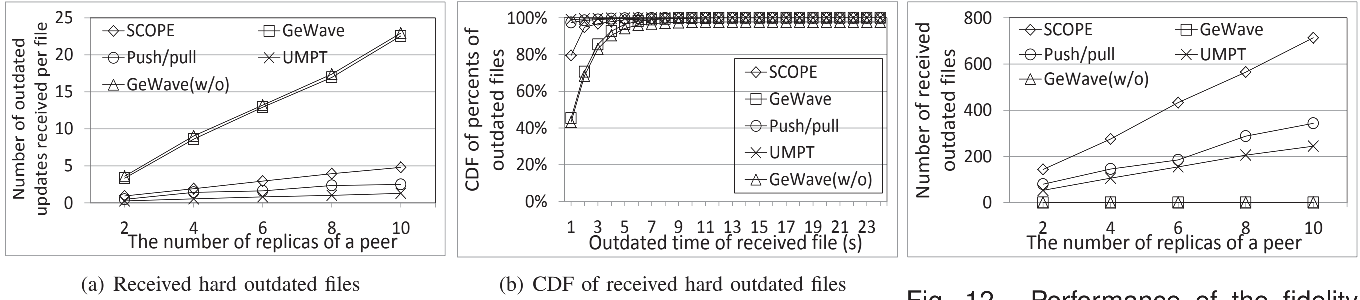
5.4 File Query Latency

In this experiment, we evaluated the average query latency of a file. To make other methods comparable to *GeWave*, we assume that replica nodes in other methods also know whether their replicas are up-to-date when receiving a file



(a) Distance (b) Latency (c) Num. of messages

Fig. 10. Communication cost in trace-driven experiments.



(a) Received hard outdated files (b) CDF of received hard outdated files

Fig. 11. Performance in achieving goals of consistency in trace-driven experiments.

Fig. 12. Performance of the fidelity of consistency in trace-driven experiments.

query. The query latency is defined as the time period from the arrival of a query at a replica node (or file owner) to the time when an up-to-date file is ready to respond. When a node's replica is not updated, the node delays the response for file queries until it receives the update or a "no update" message. Figure 9 shows the average query latency per query in 2000 seconds for different methods. We see that the average latency follows *SCOPE>Push/pull>UMPT>GeWave*, and the query latency is independent of the number of replicas per file for all methods. The query rate is uniformly distributed during [0,100]s, which means that the query distribution over a time interval between two update arrivals is also uniform. As explained in Figure 8, the update latency follows *SCOPE>Push/pull>UMPT>GeWave*. Therefore, the query latency also exhibits the same tendency.

5.5 Trace-driven Experiments

We used a 4-hour I/O (read/write) trace from a large-scale application known as CTH [45] for file reads and writes in our experiments. The trace has 16,566 files, 3,972,284 file I/O calls and 3,300 clients. The query rate varies in [0,315] visits/hour, and update rate varies in [0, 4889] writes/hour. We selected 200 active files with the highest visit rates.

The BitTorrent User Activity Trace [46] traced the downloading status of 3,570,587 peers in 242 countries or districts all over the world. The number of peers of each country varying in [1,498,238] follows a power-law distribution. As the CTH trace has 3,300 clients, we simulated 3,300 peers following the same geographical distribution as the trace. That is, a peer is hosted in a randomly selected PlanetLab node in the same country as the peer among all 330 PlanetLab nodes.

We assigned the 200 files to randomly selected pees. In the experiment, first, each peer of the 3,300 peers randomly selected m files to visit, and m was varied from 2 to 10 in an increase step of 2. Each node replicates its visited file. Then, file updating and file visiting start at the same time. The files' update rate and visit rate follows the rate distributions in the trace. We conducted each experiment lasting one hour to measure the update communication cost, communication latency, and the total number of messages for each m. We also measured the number of hard outdated files and soft outdated files to show the performance of the fidelity of consistency in different methods.

We use *GeWave(w/o)* to denote *GeWave* without proximity-awareness. Figure 10(a) and Figure 10(b) and Figure 10(c) show the total update communication cost measured by distance and latency and the number of update messages, respectively. These figures show the same tendency as Figure 2(a), Figure 2(b) and Figure 4, respectively due to the same reasons. *GeWave* produces the highest performance in all methods, which saves 74.2%, 69.1% and 64.4% more than *UMPT* in distance, latency and the number of update messages, respectively. Compared to *GeWave(w/o)*, *GeWave* saves 53.5% and 44.5% more update communication cost in distance and latency, respectively. This indicates that the necessity of proximity consideration in *GeWave* for parent node selection, which saves around half of the communication load.

Figure 11(a) shows the number of received hard outdated files. Due to the same reason as Figure 6(a), *GeWave* and *GeWave(w/o)* generate most hard outdated files. *GeWave(w/o)* does not consider the proximity, which makes the update communication latency larger. Thus, it receives around 3.9% more hard outdated files than *GeWave*. Figure 11(b) shows the CDF of the received hard outdated files versus the outdated time, which shows the same tendency as Figure 6(b) due to the same reasons. It also shows that *GeWave(w/o)* has 2.8% more outdated files with outdated time larger than 3 seconds than *GeWave* due to the same reason as Figure 11(a). Both of the two figures indicate the necessity of the proximity considera-

tion in *GeWave*. Figure 12 shows the number of received soft outdated files. It exhibits the same tendency as Figure 7 due to the same reasons, which confirms GeWave's high performance in maintaining high fidelity of file consistency.

6 Conclusions

The challenges for consistency maintenance schemes in P2P systems are twofold: large scale and churn. To handle these challenges, we propose a Geographically-aware Wave consistency maintenance scheme (GeWave) that conducts consistency maintenance efficiently at a significantly lower cost. Without relying on a static structure, GeWave is highly resilient to P2P churn by adaptive polling with direct node-to-node communication. It avoids unnecessary file updates by dynamically adapting to time-varying file update and query rates. Also, it transmits update messages in a decentralized manner between physically close nodes. Extensive experimental results with PlanetLab demonstrate the effectiveness of GeWave in comparison to other consistency maintenance schemes. Its low overhead, high efficiency, and churn-resilience are particularly attractive to the deployment of large-scale and dynamic P2P file sharing systems.

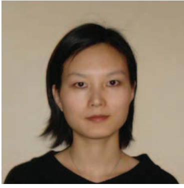
Acknowledgements

This research was supported in part by U.S. NSF grants CNS-1249603, OCI-1064230, CNS-1049947, CNS-1156875, CNS-0917056 and CNS-1057530, CNS-1025652, CNS-0938189, CSR-2008826, CSR-2008827, Microsoft Research Faculty Fellowship 8300751, and U.S. Department of Energy's Oak Ridge National Laboratory including the Extreme Scale Systems Center located at ORNL and DoD 4000111689. An early version of this work was presented in the Proceedings of ICPP'08 [47].

References

[1] J. Zhou, L. N. Bhuyan, and A. Banerjee. An Effective Pointer Replication Algorithm in P2P Networks. In *Proc. of IPDPS*, 2008.
[2] T. Liu, M. Bao, G. Chang, and Z. Tan. The Improved Research of Chord Based on File-Partition Replication Strategy. In *Proc. of HIS*, 2009.
[3] J. Kageyama, M. Kobayashi, S. Shibusawa, and T. Yonekura. A file replication method based on demand forecasting in P2P networks. In *Proc. of Second ICADIWT*, pages 268–74, 2009.
[4] W. K. Lin, C. Ye, and D. M. Chiu. Decentralized Replication Algorithms for Improving File Availability in P2P Networks. In *Proc. of IWQoS*, 2007.
[5] S. L. Monni. Adaptive Media Replication in Unstructured P2P File Sharing Systems Based on Geographical Properties and Query Distributions. In *Proc. of AXMEDIS*, pages 171–179, 2008.
[6] M. Takaoka, M. Uchida, K. Ohnishi, and Y. Oie. Thermal Diffusion-based Access Load Balancing for P2P File Sharing Networks. In *Proc. of ICCGI*, pages 284–290, 2008.
[7] Y. Fang, L. Huo, and H. Hu. Research of Replication in Unstructured P2P Network. In *Proc. of WiCom*, pages 1–4, 2009.
[8] L. Guo, S. Yang, R. Zhang, X. Niu, and H. Song. RBMA: Replication Based on Multilevel-Agent for P2P Systems . In *Proc. of CNMT*, 2009.
[9] J. Ni, S. J. Harrington, and N. Sharma. Designing File Replication Schemes for Peer-to-Peer File Sharing Systems. In *Proc. of ICC*, 2008.
[10] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of SIGCOMM*, 2001.
[11] 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.
[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stocia. Wide-area cooperative storage with CFS. In *Proc. of SOSP*, 2001.
[13] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer Caching Schemes to Address Flash Crowds. In *Proc. of IPTPS*, 2002.
[14] M. Theimer and M. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proc. of ICDCS*, 2002.
[15] Gnutella home page. http://www.gnutella.com.
[16] E. Cohen and S. Shenker. Replication strategies in unstructured peer-to-peer networks. In *Proc. of ACM SIGCOMM*, 2002.
[17] S. Tewari and L. Kleinrock. Analysis of Search and Replication in Unstructured Peer-to-Peer Networks. In *Proc. of SIGMETRICS*, 2005.
[18] S. Tewari and L. Kleinrock. Proportional Replication in Peer-to-Peer Network. In *Proc. of INFOCOM*, 2006.
[19] D. Rubenstein and S. Sahu. Can Unstructured P2P Protocols Survive Flash Crowds? *IEEE/ACM Trans. on Networking*, (3), 2005.
[20] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. Adaptive Replication in Peer-to-Peer Systems. In *Proc. of ICDCS*, 2004.
[21] 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.
[22] Kazaa, 2001. Kazaa home page: www.kazaa.com.
[23] Morpheus home page. http://www.musiccity.com.
[24] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.
[25] M Waldman, AD Rubin, and LF Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. of the 9th USENIX Security Symposium*, 2000.
[26] Z. Li, G. Xie, and Z. Li. Efficient and Scalable Consistency Maintenance for Heterogeneous Peer-to-peer Systems. *TPDS*, 2008.
[27] R. Zhou and K. Hwang. PowerTrust: A Robust and Scalable Reputation System for Trusted Peer-to-Peer Computing. *TPDS*, 2007.
[28] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM TON*, 2003.
[29] A. Datta, M. Hauswirth, and K. Aberer. Updates in Highly Unreliable, Replicated Peer-to-Peer Systems. In *Proc. of ICDCS*, 2003.
[30] J. Lan, X. Liu, P. Shenoy, and K. Ramamritham. Consistency maintenance in peer-to-peer file sharing networks. In *Proc. of WIAPP*, 2003.
[31] X. Chen, S. Ren, H. Wang, and X. Zhang. SCOPE: scalable consistency maintenance in structured P2P systems. In *Proc. of INFOCOM*, 2005.
[32] M. Roussopoulos and M. Baker. CUP: Controlled Update Propagation in Peer-to-Peer Networks. In *Proc. of USENIX ATC*, 2003.
[33] L. Yin and G. Cao. DUP: Dynamic-tree Based Update Propagation in Peer-to-Peer Networks. In *Proc. of ICDE*, 2005.
[34] R. Shoup and F. Travostino. ebay's scaling odyssey, growing and evolving a large ecommerce site. Invited Industrial Talk in LADIS'08, http://www.slideshare.net/ebayworld/ebays-scaling-odyssey.
[35] Y. Hu, M. Feng, and L. N. Bhuyan. A Balanced Consistency Maintenance Protocol for Structured P2P Systems. In *Proc. of INFOCOM*, 2010.
[36] M. Raunak P. Shenoy B. Urgaonkar, A. Ninan and K. Ramamritham. Maintaining Mutual Consistency for Cached Web Objects. In *Proc. of ICDCS*, 2001.
[37] H. Shen. IRM: Integrated File Replication and Consistency Maintenance in P2P Systems. *TPDS*, 2009.
[38] W. Acosta and S. Chandra. On The Need For Query-Centric Unstructured Peer-To-Peer Overlays. In *Proc. of IEEE IPDPS*, 2008.
[39] C. Zhang and Z. Zhang. Trading Replication Consistency for Performance and Availability: an Adaptive Approach. In *Proc. of ICDCS*, 2003.
[40] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Proc. of INFOCOM*, 2002.
[41] Z. Xu, M. Mahalingam, and M. Karlsson. Turning heterogeneity into an advantage in overlay routing. In *Proc. of INFOCOM*, 2003.
[42] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier. Space filling curves and their use in geometric data structure. *TCS*, 1997.
[43] PlanetLab. http://www.planet-lab.org/.
[44] 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.
[45] E. S. Hertel, Jr., R. L. Bell, M. G. Elrick, and et al. CTH: A Software Family for Multi-Dimensional Shock Physics Analysis. Technical report, Sandia National Laboratories, Albuquerque, New Mexico, USA.
[46] BitTorrent User Activity Traces. http://www.cs.brown.edu/p̃avlo/torrent/.
[47] H. Shen. GeWave: Geographically-aware Wave for File Consistency Maintenance in P2P Systems. In *Proc. of ICPP*, 2008.
[48] 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.

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.



Haiying Shen Haiying Shen received the BS degree in Computer Science and Engineering from Tongji University 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 ECE Department at Clemson University. Her research interests include P2P networks, mobile computing, and cloud computing. She is a Microsoft Faculty Fellow of 2010 and a member of the IEEE and 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 distributed networks, with an emphasis on Peer-to-Peer, data center and online social networks. He is a student member of IEEE.