SCOPE: Scalable Consistency Maintenance in Structured P2P Systems

Xin Chen
Ask Jeeves Inc.
& Department of Computer Science
College of William and Mary
xinchen@cs.wm.edu

Shansi Ren, Haining Wang, Xiaodong Zhang
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187-8795, USA
{sren, hnw, zhang}@cs.wm.edu

***Abstract*— While current Peer-to-Peer (P2P) systems facilitate static file sharing, newly-developed applications demand that P2P systems be able to manage dynamically-changing files. Maintaining consistency between frequently-updated files and their replicas is a fundamental reliability requirement for a P2P system. In this paper, we present *SCOPE*, a structured P2P system supporting consistency among a large number of replicas. By building a replica-partition-tree (RPT) for each key, SCOPE keeps track of the locations of replicas and then propagates update notifications. Our theoretical analyses and experimental results demonstrate that SCOPE can effectively maintain replica consistency while preventing hot-spot and node-failure problems. Its efficiency in maintenance and failure-recovery is particularly attractive to the deployment of large-scale P2P systems.**

Keywords: structured P2P systems, replica consistency, hierarchical trees.

I. Introduction

Structured P2P systems have been successfully designed and implemented for global storage utility (such as PAST [29], CFS [9], OceanStore [17], and Pangaea [30]), publishing systems (such as FreeNet [7] and Scribe [4]), and Web-related services (such as Squirrel [14], SFR [33], and Beehive [21]). Among all these P2P-based applications, replication and caching have been widely used to improve scalability and performance. However, little attention has been paid to maintaining replica consistency in structured P2P systems. On one hand, without effective replica consistency maintenance, a P2P system is limited to providing only static or infrequently-updated object sharing. On the other hand, newly-developed classes of P2P applications do need consistency support to deliver frequently-updated contents, such as directory service, online auction, and remote collaboration. In these applications, files are frequently changed, and maintaining consistency among replicas is a must for correctness. Therefore, scalable consistency maintenance is essential to improve service quality of existing P2P applications, and to meet the basic requirement of newly-developed P2P applications.

Existing structured P2P systems rely on distributed hash tables (DHTs) to assign objects to different nodes. Each node is expected to receive roughly the same number of objects, thanks to the load balance achieved by DHTs. However, the system may become unbalanced when objects have different popularities and numbers of replicas. In a scalable replica updating mechanism, the location of a replica must be traceable, and no broadcasting is needed for the propagation of an update notification. Current structured P2P systems take a straightforward approach to track replica locations [32], [24]—a single node stores the locations of all replicas. This approach provides us with a simple solution of maintaining data consistency. However, it only works well if the number of replicas per object is relatively small in a reliable P2P system. Otherwise, several problems may occur as follows.

- *Hot-spot problem*: due to the different objects' popularities, the number of replicas per object varies significantly, making the popular nodes heavily loaded while other nodes carry much less replicas.
- *Node-failure problem*: if the hashed node fails, update notifications have to be propagated by broadcasting.
- *Privacy problem*: the hashed node knows all replicas' locations, which violates the privacy of original content holders.

To address the deficiencies in existing structured P2P systems, we propose a structured P2P system with replica consistency support, called Scalable COnsistency maintenance in structured PEer-to-peer systems (*SCOPE*). Unlike existing structured P2P systems, SCOPE distributes all replicas' location information to a large number of nodes, thus preventing hot-spot and node-failure problems. It also avoids recording explicitly the IP address or node ID of a node that stores a replica, thus protecting the privacy of the node. By building a replica-partition-tree (RPT) for each key, SCOPE keeps track of the location of replicas and then propagates update notifications. We introduce three new operations in SCOPE to maintain consistency.

- *Subscribe*: when a node has an object and needs to keep it up-to-date, it calls *subscribe* to receive a notification of the object update.
- *Unsubscribe*: when a node neither needs a replica nor keeps it up-to-date, it calls *unsubscribe* to stop receiving update notifications.
- *Update*: when a node needs to change the content of an object, it calls *update* to propagate the update notification[1] to all subscribed nodes.

[1] invalidation message or the key itself.

In SCOPE, we allow multiple writers to co-exist, since the update operation on a key can be invoked by any node keeping a replica of that key. In contrast, in some practical applications, usually only one node is authorized to update a key. SCOPE can be easily applied to single-writer applications.

Since SCOPE directly utilizes DHTs to manage object replicas, it effectively supports consistency among a large number of peers. As a general solution, SCOPE can be deployed in any existing structured P2P systems, such as CAN [24], Chord [32], Pastry [28], and Tapestry [36]. Our theoretical analyses and simulation experiments show that SCOPE can achieve replica consistency in a scalable and efficient manner. In an N-node network, each peer is guaranteed to keep at most $O(\log N)$ partition vectors for a single key, regardless of the key's value and its popularity. Due to the hierarchical management, only $O(1)$ nodes are updated when a node joins or leaves, and only $O(\log^2 N)$ messages are transmitted to recover a node failure.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 presents the RPT structure in SCOPE. Section 4 describes the operations defined in SCOPE. Maintenance and recovery procedures are introduced in Section 5. We evaluate the performance of SCOPE using Pastry routing algorithm in Section 6. In Section 7, we briefly discuss SCOPE design alternatives. Finally, we conclude the paper in Section 8.

II. Related Work

Replication is effective to improve the scalability and object availability of a P2P system. However, most proposed replication schemes are focused on how to create replicas. Maintaining consistency among a number of replicas is not fully investigated, posing a challenge for building a consistent large-scale P2P system. Different from all proposed solutions, our approach utilizes the nature of DHTs to organize the replicas in a distributed way. Therefore, it has better scalability and higher efficiency.

Some existing file-sharing P2P systems assume that the shared data are static or read-only, so that no update mechanism is needed. Most unstructured P2P systems, including centralized ones (e.g., Napster) and decentralized ones (e.g., Gnutella), do not guarantee consistency among replicas. Researchers have designed several algorithms to support consistency in a best-effort way. In [10], a hybrid push/pull algorithm is used to propagate updates to related nodes, where flooding is substituted by rumor spreading to reduce communication overhead. At every step of rumor spreading, a node pushes updates to a subset of related nodes it knows, only providing partial consistency. Similarly, in Gnutella, Lan *et al.* [18] proposed to use the flooding-based active push for static objects and the adaptive polling-based passive pull for dynamic objects. However, it is hard to determine the polling frequency, thus essentially no guaranteed consistency is provided. In [27], Roussopoulos and Baker proposed an incentive-based algorithm called CUP to cache metadata—lookup results—and keep them updated in a structured P2P system. However, CUP only caches the metadata, not the object itself, along the lookup path with limited consistency support. So, it cannot maintain consistency among the replicas of an object. Considering the topology mismatch problem between overlays and their physical layers in structured P2P systems, [25] proposed an adaptive topology adjusting method to reduce the average routing latency of a query. In [13], a network of streaming media servers is organized into a structured P2P system to fully utilize local cached copies of an object, so that the average streaming start-up time can be reduced.

For applications demanding consistency support among replicas, different solutions have been proposed in various P2P systems. Most proposed P2P-based publish/subscribe systems record paths from subscribers to publishers, and use them to propagate updates. As an anonymous P2P storage and information retrieval system, FreeNet [7] protects the privacy of both authors and readers. It uses a *content-hash key* to distinguish different versions of a file. An update is routed to other nodes based on key closeness. However, the update is not guaranteed to reach every replica. Based on Pastry, Scribe [4] provides a decentralized event notification mechanism for publishing systems. A node can be a publisher by creating a topic, and other nodes can become its subscribers through registration. The paths from subscribers to the publisher are recorded for update notifications. However, if any node on the path fails, some subscribers are not reachable unless broadcasting is used.

Being a major P2P application, a wide-area file system relies on replication to improve its performance. In [8], a decentralized replication solution is used to achieve practical availability, without considering replica consistency. PAST [29] is a P2P-based file system for large-scale persistent storage service. In PAST, a user can specify the number of replicas of a file through central management. Although PAST utilizes caching to shorten client-perceived latency, it does not maintain consistency of cached contents. Similarly, CFS [9] is a P2P read-only storage system, and avoids most cache inconsistency problems by content hashes. Each client has to validate the freshness of a received file by itself, and stale replicas are removed from caches by LRU replacement. OceanStore [17] maintains two-tier replicas: a small durable primary tier and a large, soft-state second tier. The primary tier is organized as a Byzantine inner ring, keeping the most up-to-date data. The replicas in the second tier are connected through multicast trees, i.e., dissemination trees (d-tree). Periodic heartbeat messages are sent for fault resilience, which incurs significant communication overhead. Similar solutions have been used in P2P-based real-time multimedia streaming (e.g., Bayeux [37] and SplitStream [5]). Pangaea [30] creates replicas aggressively to improve overall performance. By organizing all replicas of a file in a strongly-connected graph, it propagates an update from one server to the others through flooding, which does not scale well with a large number of replicas. Automatic replica regeneration [35] has been proposed to provide higher availability with a small

number of replicas, which are organized in a lease graph. A two-phase write protocol is used to optimize reads and linearize the read/write process.

Most newly-proposed Web services on P2P structures still employ the time-to-live (TTL) mechanism to refresh their replicas. For example, Squirrel [14] is such a system based on the Pastry routing protocol. The freshness of a cached object is determined by the Web cache expiration policy (e.g., TTL field in response headers). In order to facilitate Web object references, *Semantic Free Reference* (SFR) [33] has been proposed to resolve the object locations. Based on DHTs, SFR utilizes the caches of different infrastructure levels to improve the resolving latency. Beehive, designed for domain name systems [21], [22], provides O(1) lookup latency. Different from widely used passive caching, it uses proactive replication to significantly reduce the lookup latency. In [12], Gedik *et al.* used a dynamic passive replication scheme to provide reliable service for a P2P Internet monitoring system, where the replication list is maintained by each Continual Queries (CQ) owner.

III. The Base of SCOPE Protocol

The SCOPE protocol specifies: (1) how to record the locations of all replicas; (2) how to propagate update notifications to related peers; (3) how to join or leave the system as a peer; and (4) how to recover from a node's failure. This section describes how to record the replica locations by building a *replica-partition-tree* (RPT)—a distributed structure for load balancing in SCOPE. The operation algorithms and maintenance procedures will be presented in Sections 4 and 5, respectively.

A. Overview

In DHTs each key is assigned to a node according to its identifier, and we call this original key-holder the *primary node* of the key. To avoid the primary node becoming the hot spot, SCOPE splits the whole identifier space into partitions and selects one representative node in each partition to record the replica locations within that partition. Each partition may be further divided into smaller ones, in which child nodes are selected as the representatives to take charge of the smaller partitions. As the root of this partition-tree, the primary node only records the key existence in the partition one level beneath, while its child representative nodes record the key existence in the partitions two levels below the root; and so on and so forth. In this way, the overhead of maintaining consistency at one node is greatly reduced and undertaken by the representative nodes at lower levels. Since the hash function used by DHTs distributes keys to the whole identifier space, the load of tree maintenance is balanced among all nodes at any partition level. Note that the location information at any level is obtainable from representative nodes at lower levels, the partition-tree also provides a recovery mechanism to handle a node failure.

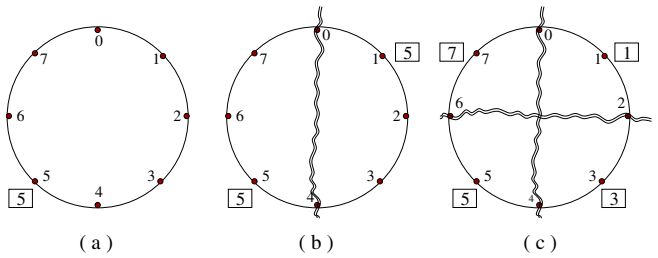

Fig. 1. (a) A 3-bit identifier space; (b) The same identifier space with two partitions; (c) The same identifier space with four partitions.

B. Partitioning Identifier Space

A consistent hash function (e.g. SHA-1) assigns each node and each key an m-bit identifier, respectively. If we use a smaller identifier space, the key identifier can be easily calculated by keeping a certain number of least significant bits. By adding different most significant bits, the same key can be mapped to multiple smaller equally-sized identifier spaces with different identifier ranges. A partition can be further divided into smaller ones, and it records the existence of all keys in its sub-partitions. Figure 1(a) shows an identifier space with $m = 3$. Suppose there is a key 5 in the space. If the original space is split into two partitions as shown in Figure 1(b), one with space $[0, 3]$ and the other with space $[4, 7]$, the key can be hashed to 1 in the first partition and 5 in the second partition, respectively. If we further split each partition into two sub-partitions as Figure 1(c) illustrated, the identifiers of the same key can be located in the smaller spaces at 1, 3, 5, and 7, respectively. Figure 2 shows the root of key 5 (101) in the original 3-bit identifier space and its representative nodes in the two-level partitions. At the top level, the root node is located at 5 (101). At the intermediate level, the two least significant bits (01) are inherited from the root, while the different value (0 or 1) is set at the most significant bit to locate the representative nodes R1 and R2 in the two partitions, respectively. At the bottom level, only the least significant bit (1) is inherited from the root but two most significant bits are set to four different values (00/01/10/11) in order to determine the locations of representative nodes R11, R12, R21, and R22, respectively. Note that the partitioning is logical and the same node can reside in multiple levels. For example, the root node (101) is used as the representative node in all partition levels.

C. Building Replica-Partition-Trees (RPTs)

1) Basic Structure: After partitioning the identifier space as mentioned above, we build an RPT for each key by recursively checking the existence of replicas in the partitions. The primary node of a key in the original identifier space is the root of RPT(s). The representative node of a key in each partition, recording the locations of replicas at the lower levels, becomes one intermediate node of RPT(s). The leaves of RPT(s) are those representative nodes at the bottom level. Each node of RPT(s) uses a vector to record the existence of replicas in its sub-trees, with one bit for each child partition.

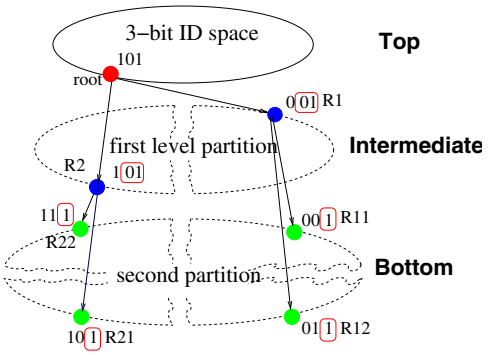

Fig. 2. Key 5 (101) in a 3-bit identifier space and its representative nodes at different levels of partitions.

Figure 3(a) shows an example with the identifier space of $[0, 7]$. The nodes 0, 4, and 7 in the space keep a replica of key 5. The RPT for key 5 is shown in Figure 3(b). At the top level, a 2-bit vector is used to indicate the existence of replicas in the two sub-trees. At the bottom level, four 2-bit vectors are used to indicate the existence of key 5 in all eight possible positions from 0 to 7. In general, if the identifier space is 2^M, the height of RPT(s) for any key is $O(M)$. Consider that most DHTs use a 160-bit SHA-1 hashing function, which may result in tens of partition levels. For example, if we split each partition into 64 (2^8) pieces, we will have 20 levels. Obviously, too many levels of partitions would make the RPT construction and the update propagation inefficient.

2) Scalable RPT: Since the number of nodes is much smaller than the identifier space, our goal is to reduce the heights of RPTs to the logarithm of the number of nodes. In the partitioning algorithm presented above, each partition is recursively divided into smaller ones until only one identifier remains. The leaf nodes of RPTs record the existence of keys at the corresponding identifiers. However, if a partition only contains one node, there is no need for further partitioning to locate the node. For example, as shown in Figure 3(a), only node 0 exists in the partition of $[0, 3]$. During subscribe/unsubscribe operations, node 0 only needs to inform the primary node of key 5, which records the first level partition $[0, 3]$ and $[4, 7]$. When the key is modified, it can directly notify node 0 by sending an invalidation message to the first identifier in $[0, 3]$, which is 0. By removing the redundant leaf nodes, we can build a much shorter RPT. The RPT after the removal of redundant leaf nodes, is shown in Figure 3(c).

The method given above can significantly reduce the partition levels if nodes are distributed sparsely. However, even if the total number of nodes is small, the number of partition levels could still be large when most nodes are close to each other. Figure 4(a) shows an example with the identifier space of [0,7], where two nodes 6 and 7 subscribe key 5. The RPT is illustrated in Figure 4(b). Both nodes are in the same partition until the identifier space is decreased to 1—the bottom level of the partition. The height of this RPT is 3, and it cannot be condensed by reducing leaf nodes. In general, if the nodes' identifiers happen to be consecutive and we only remove the leaf nodes as above, the height of RPT(s) will still be $O(M)$.

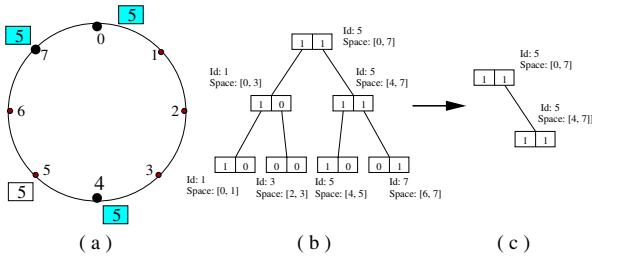

Fig. 3. (a) In the identifier space of [0,7], nodes 0, 4, and 7 subscribe key 5; (b) The RPT of key 5; (c) The RPT after removing redundant leaf nodes.

We resolve this problem by removing the redundant intermediate nodes. If all nodes in a partition are clustered in one of its lower-level partitions, it is possible to reduce the intermediate nodes. Figure 4(c) shows one optimized RPT. The intermediate node for the partition $[4, 7]$ is removed since only one of its lower-level partition $[6, 7]$ has nodes. Thus, the height of the RPT is decreased from 3 to 2.

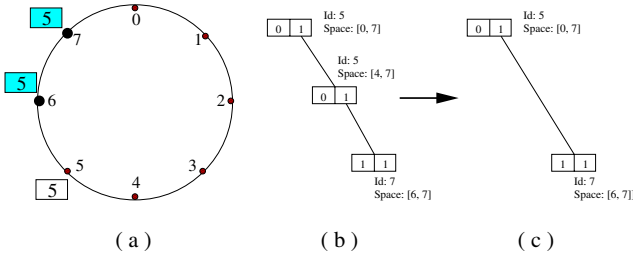

Fig. 4. (a) Nodes 6 and 7 subscribe key 5; (b) The RPT for key 5 after removing redundant leaf nodes; (c) The RPT after removing both redundant leaf nodes and intermediate nodes.

Theorem 1: For an N-node network with partition size of 2^m, the average height of RPTs is $O(\frac{\log N}{m})$, regardless of the size of an identifier space.

Proof: Suppose the whole identifier space is 2^M. Every partitioning generates 2^m smaller equally-sized partitions, each with size of $1/2^m$ of the previous partition range. After $\frac{\log N}{m}$ time partitioning, the identifier range of each partition is reduced to $2^M/2^{\log N} = 2^M/N$. The height of RPT grows to $\frac{\log N}{m}$, with maximal height $\log N$ at $m = 1$. Note that the expected number of node identifiers in the range of this size is 1. Due to identifier randomness induced by SHA-1 hash function, the average height of all RPTs is $O(\frac{\log N}{m})$. ■

D. Load Balancing

RPT effectively balances the load across the network, disregarding the key values and their popularities. By using RPT, we conclude that:

Theorem 2: In an N-node network with partition size of 2^m, for a key with C subscribers, the average number of partition vectors in its RPT is $O(\log N \cdot C)$.

Proof: In the RPT of the key, only one root is located at the top level. At the second level, at most $\min(2^m, C)$ representative nodes have one partition vector. At the xth level

of the RPT, at most $\min(2^{xm}, C)$ representative nodes are involved. The total number of vectors S of the RPT is:

$$\begin{aligned} S &= 1 + \min(2^m, C) + \min(2^{2m}, C) \\ &\quad + ... + \min(2^{\frac{\log N}{m} m}, C) \\ &= \sum_{i=0}^{a-1} 2^{im} + \sum_{i=a}^{\frac{\log N}{m}} C \\ &= (\log_m N - a)C + \frac{2^{am} - 1}{2^m - 1}, \\ &\quad \text{for } 2^{(a-1)m} < C \leq 2^{am}, 1 < a \leq \tfrac{\log N}{m} \end{aligned}$$

Compared with the number of subscribers C, the number of vectors is increased to $(\log N - a) + \frac{2^{am}-1}{(2^m-1)C}$. Since $\frac{2^{am}-1}{(2^m-1)C} < \frac{2^{am}}{(2^m-1)2^{(a-1)m}} = \frac{2^m}{2^m-1}$, which is less than 2 ($m \geq 1$), the maximal value is achieved when $a = 1$, and the total number of vectors in the RPT is $O(\log N)$ times of the number of subscribers.

■

IV. Operation Algorithms

A. Subscribe/Unsubscribe

The subscribe/unsubscribe procedures are initiated by subscribers and proceed toward the root of an RPT. The process can be implemented in an iterative or recursive way. With iteration, the subscriber itself has to inform all representative nodes one by one. With recursion, each representative node is responsible for forwarding the subscriptions to the next higher level until the root node is reached. In SCOPE, we implement the subscribe/unsubscribe operations recursively for routing efficiency.

At the beginning, each subscriber locates its immediate upper-level partition from its predecessor's and successor's identifiers. Then, the node sends subscribe/unsubscribe requests to the upper-level representative node. The representative node checks if it has a vector allocated for the key. If so, it sets/unsets the corresponding bit, and the subscribe/unsubscribe procedure terminates there. Otherwise, it creates/deletes the vector of the key, sets/unsets the bit, and continues forwarding subscribe/unsubscribe requests to the representative node at the next upper-level partition. This process proceeds until it reaches the root of the RPT. The routing algorithms of the operations depend on the type of the specific structured P2P systems. In this section, we use Pastry as the base routing scheme for the purpose of analysis. Note that similar analysis is applicable to other hypercube routing algorithms as well.

Figure 5 illustrates a subscribe/unsubscribe process in a 3-bit identifier space, where node 2 (010) subscribes/unsubscribes key 5 (101). At first, node 2 notifies the representative node 3 (011) at the bottom level, then node 3 informs the representative node 1 (001) at the intermediate level. Finally, node 1 informs the root node 5, which is the representative node of the whole space.

In order to improve routing efficency, every node maintains level indices to indicate the node's partitions at different

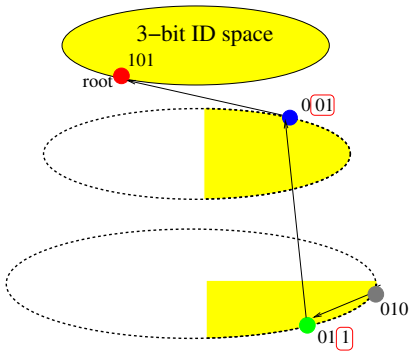

Fig. 5. Node 2 (010) subscribe Key 5 (101) in a 3-bit identifier space.

levels. As we have pointed out before, reducing intermediate partitions makes the height of RPT different from the depth of partitioning. The level index is used to record the change of the RPT height with the increase of partitioning. Its maximal length is equal to M for an identifier space with size of 2^M. The i^{th} entry in a level index is the height of the RPT at i^{th} partition level.

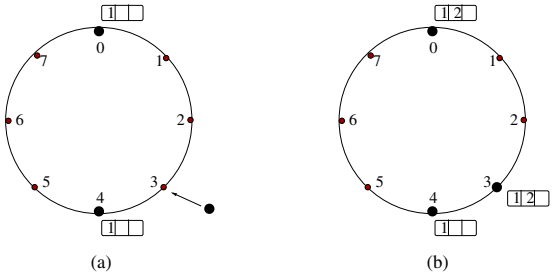

Fig. 6. Level index changes after node 3 joins in a 3-bit space.

Figure 6 shows an example of the level index in a 3-bit identifier space. Before node 3 joins, nodes 1 and 4 are identified after first-time partitioning. Both of them have the same level index $\{1, \sqcup, \sqcup\}$, where $\sqcup$ represents an empty space. With the participation of node 3, the whole space needs to be partitioned twice to identify nodes 0 and 3, and no redundant intermediate partition exists. The RPT grows as partitioning proceeds, and the level indices of nodes 0 and 3 become $\{1, 2, \sqcup\}$. Comparatively, node 4 is identified after the first-time partitioning and its level index is $\{1, \sqcup, \sqcup\}$.

With the Pastry routing table and leaf set, a node can reach any other node in a range of 2^{xm} within $O(\log(\frac{N}{2^{M-xm}}))$ hops. When a node initiates a subscribe/unsubscribe operation, it also forwards its level index to the representative nodes at upper levels. Each intermediate representative uses the level index to derive the location of its higher level representative.

Lemma 1: For an N-node network with partition size of 2^m in a 2^M identifier space, in any range of 2^{xm}, on average a node can find the successor of a key in $O(\log(\frac{N}{2^{M-xm}}))$ hops.

Proof: We use Pastry as the base routing algorithm, and assume that a node's routing table is organized into $\lceil log_{2^b} M \rceil$ levels with $2^b - 1$ entries each. Due to the usage of the SHA-1 hash function, node and key identifiers are randomly

distributed and the number of nodes in a range of 2^{xm} is $\frac{N}{2^{M-xm}}$ on average. Suppose that node n wants to resolve a query for the successor of k, $k > n$ and $0 < k - n < 2^{xm}$. Node n forwards its query to the close predecessor of k in its routing table. Suppose key k is in the i^{th} $(0 < i <= xm/b)$ level j^{th} $(0 < j < 2^b)$ interval of node n. If this interval is empty, node n has found the successor of k. Otherwise, it fingers some node f in this interval. The distance between n and f is at least $2^{xm-i\cdot b}$. Since f and k are both in i's level, the distance between them is at most $2^{xm-i\cdot b}$, less than half of the distance from n to k.

After $\log(\frac{N}{2^{M-xm}})$ forwardings, the distance between the current query node n and key k is reduced to $2^{xm}/2^{\log(\frac{N}{2^{M-xm}})} = 2^M/N$. Because the expected number of node identifiers in this range is 1, the successor of key k can be reached in $O(\log(\frac{N}{2^{M-xm}}))$ hops on average. If $n > k$ and $0 < n - k < 2^{xm}$, we can obtain the same result. ■

Theorem 3: For an N-node network with partition size of 2^m, on average the subscribe/unsubscribe operations can be finished in $O(\frac{\log^2 N}{2m})$ hops.

As we have learned from the previous section, the average height of RPT(s) is $O(\frac{\log N}{m})$. In order to finish a subscribe/unsubscribe operation, $O(\frac{\log N}{m})$ levels are traversed on average. From Lemma 1, at each level, on average a query node can reach the successor of a key in the same partition at level l in $\log N/2^{lm}$ hops, the average routing length of a subscribe/unsubscribe operation ($hop(sub/unsub)$) is:

$$\begin{aligned} hop(sub/unsub) &= \log N + \log\frac{N}{2^m} + \log\frac{N}{2^{2m}} \\ &\quad +... + \log\frac{N}{2^{\log N}} \\ &= \log N + (\log N - m) + (\log N - 2m) \\ &\quad +... + (\log N - \frac{\log N}{m}m) \\ &= \frac{\log^2 N}{2m} - \frac{\log N}{2}. \end{aligned}$$

Therefore, on average a subscribe/unsubscribe operation can be finished in $O(\frac{\log^2 N}{2m})$ hops.

While the upper bound of subscribe path length is longer than that of the centralized solution, the subscribe/unsubscribe operations in SCOPE are efficient if multiple subscribers exist. The subscribe/unsubscribe process terminates at any representative node that has recorded the same key at the same level. When a node subscribes a key with C replicas, the average length of the subscribe/unsubscribe process can be expressed as follows:

$$\begin{aligned} hop(sub/unsub) &= \log N + \log\frac{N}{2^m} + ... + \log\frac{N}{2^{\log N}} \\ &\quad -(\log N + \log\frac{N}{2^m} + ... + \log\frac{N}{2^{\log C}}) \\ &= \sum_{i=0}^{\frac{\log N}{m}} \log\frac{N}{2^{im}} - \sum_{i=0}^{\frac{\log C}{m}} \log\frac{N}{2^{im}} \\ &= \sum_{i=\frac{\log C}{m}+1}^{\frac{\log N}{m}} \log(N/2^{im}) \end{aligned}$$

B. Update

The update procedure is launched by the root node after it receives update requests from a replica, and proceeds toward every subscriber. The root node first checks its vector of the key. Then, it sends notifications to the representative nodes of the partitions with corresponding bits set. Every intermediate representative is responsible for delivering the notifications to its lower-level representatives. When the notification reaches a leaf node, it is forwarded to the subscribers directly.

Theorem 4: For an N-node network with partition size of 2^m, on average, update operations can be finished in $O(\frac{\log^2 N}{2m})$ hops.

Although the update path to a single subscriber in SCOPE is longer than that in the centralized solution, the average number of update routing hops in SCOPE is smaller. This is because in SCOPE all replicas can be notified in $O(\log^2 N)$ hops, but the centralized solution needs $O(C)$ hops to finish. For a sufficiently large C, the latter incurs much longer delay.

Similar to subscribe/unsubscribe operations, the update operation in SCOPE is efficient if multiple subscribers exist. The total number of hops ($hop(update)$) for an update of a key with C replicas is:

$$\begin{aligned} hop(update) &= C\log N - (C - 2^m)\log N + C\log\frac{N}{2^m} \\ &\quad -2^m(\frac{C}{2^m} - 2^m)\log\frac{N}{2^m} + ... + C\log\frac{N}{2^{\frac{\log C}{m}m}} \\ &\quad -2^{\frac{\log C}{m}m}(C/2^{\frac{\log C}{m}m} - 2^m)\log\frac{N}{2^{\frac{\log C}{m}m}} \\ &\quad +C\log\frac{N}{2^{(\frac{\log C}{m}+1)m}} + ... + C\log\frac{N}{2^{\frac{\log N}{m}m}} \\ &= C\sum_{i=\frac{\log C}{m}+1}^{\frac{\log N}{m}} \log\frac{N}{2^{im}} + \sum_{i=0}^{\frac{\log C}{m}} 2^{(i+1)m}\log\frac{N}{2^{im}} \end{aligned}$$

Comparatively, the centralized solution needs $O(C\log N)$ hops to conduct an update operation. Our analysis above is based on the base SCOPE protocol, in which we do not record the IP addresses of descendant nodes explicitly. To further reduce the latency of update operations, a parent node may directly record the IP addresses of its RPT children. If so, SCOPE can reduce the average update hops to $O(\log N)$, which is much smaller than that of the centralized solution for a sufficiently large C.

V. Maintenance and Recovery

A. Node Joining/Leaving

This section describes how to maintain the RPT when a single node joins. A similar method can be applied to the situation where a node leaves.

Besides maintaining the predecessor/successor and routing tables, a newly-joining node in SCOPE needs to take two actions to maintain the RPT : transferring partition vectors and updating level indices. With a node joining, part of RPT vectors under the charge of the node's successor should be transferred to the newly-joined node, similar to transferring

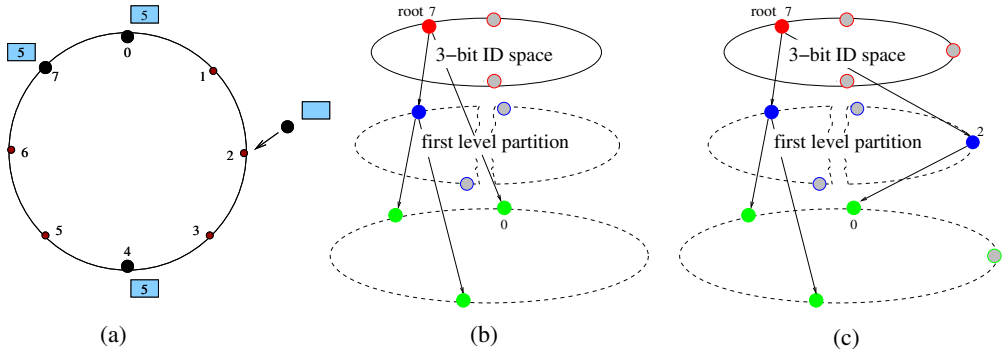

Fig. 7. (a) Node 2 (010) joins in a 3-bit identifier space; (b) The RPT of key 5 before node 2 joins; (c) The RPT of key 5 after node 2 joins.

keys. The operation is straightforward: the newly-joined node n informs its successor n', then node n' moves the related vectors at every level to node n.

A node joining may cause further partitioning to distinguish itself from other existing nodes. If node n changes the structure of any RPTs, it also needs to inform the affected nodes to update their level indices accordingly. Figure 7 illustrates the level index maintenance when node 2 joins a 3-bit identifier space. Figure 7(a) shows the node distribution. Originally, only node 0 is in the partition $[0, 3]$, and its level index is $\{1, \sqcup, \sqcup\}$. Figure 7(b) shows the RPT of key 5, where node 7, the primary node of key 5, can read the first bit of its vector and know the existence of key 5's replica at node 0. After node 2 joins, node 0 is no longer the only node in the partition $[0, 3]$, thus further partitioning within [0, 3] becomes necessary to differentiate node 0 from node 2. Subsequently, node 0 updates its level index to $\{1, 2, \sqcup\}$ and subscribes key 5 via node 2—the representative of the new partition. The modified RPT of key 5 is illustrated in Figure 7(c). Note that a newly-joined node triggers at most one partitioning. The newly-generated partition consists of the newly-joined node and at least one existing node.

Lemma 2: In an N-node network, when a node joins, on average only $O(1)$ nodes need to update their level indices.

Proof: According to the design of RPTs, we do not need to partition at a level if only one of its lower partitions has nodes. If the newly-joined node makes the partitioning a necessity, only the existing nodes in that lower partition need to update their level indices. Considering that nodes are randomly distributed among partitions, the average number of nodes in that partition is $O(1)$. ■

Theorem 5: In an N-node network with partition size of 2^m, on average, any node joining or leaving requires $O(1)$ messages to update the corresponding RPTs and level indices.

When a node joins/leaves, both RPTs and level indices can be updated with $O(1)$ messages. The total number of maintenance messages is still $O(1)$.

B. Node Failure

One advantage of multi-level partitioning is fault tolerance. The records at any level partition can be restored from its lower-level partitions. SCOPE has a recovery process invoked periodically after the stabilization process. When one peer fails, the recovery process is executed by the new node that takes over the failed one. The new node sends queries to its lower-level partitions, then restores every key's vectors based on their responses.

Suppose a node fails in an N-node network with partition size of 2^m. In order to recover the RPTs, another node taking over the failed node needs to collect all subscription information from all 2^m lower-level partitions at all $\frac{\log N}{m}$ levels. On average, the total number of messages ($Message$) is:

$$\begin{aligned} Message &= 2^m(\log N + \log \frac{N}{2^m} + \log \frac{N}{2^{2m}} \\ &\quad +\ldots + \log \frac{N}{2^{\log N}}) \\ &= 2^m \frac{\log^2 N}{2m} - 2^m \frac{\log N}{2} \end{aligned}$$

Thus, when a node fails, the recovery process only needs $O(2^m \frac{\log^2 N}{2m})$ messages.

Comparatively, a centralized approach can recover the replica locations by broadcasting the re-subscription requests to all nodes or by requiring all subscribed nodes to periodically communicate with the hashed nodes. Obviously, neither of these methods is as effective as the method that SCOPE uses.

VI. Performance Evaluation

In this section, we validate the efficacy of SCOPE through simulations. In our experiments, all nodes and keys are randomly-selected integers. They are hashed to a 160-bit identifier space via SHA-1. The number of partitions at each level is 16. Specified as the Pastry default parameters, the routing table of each node has 40 levels and each level consists of 15 entries; the leaf set of each node has 32 entries.

A. Structure Scalability

A scalable P2P system should distribute the whole storage load to a large number of nodes to avoid the hot-spot problem. We consider a network consisting of 10^4 nodes, and vary the total number of replicas of a key at 1, 10, 10^2, 10^3 and 10^4. In order to record a key and its locations, a record [*key_id, partition_level, partition_vector*] is kept at each representative node. We measure the number of nodes involved and the

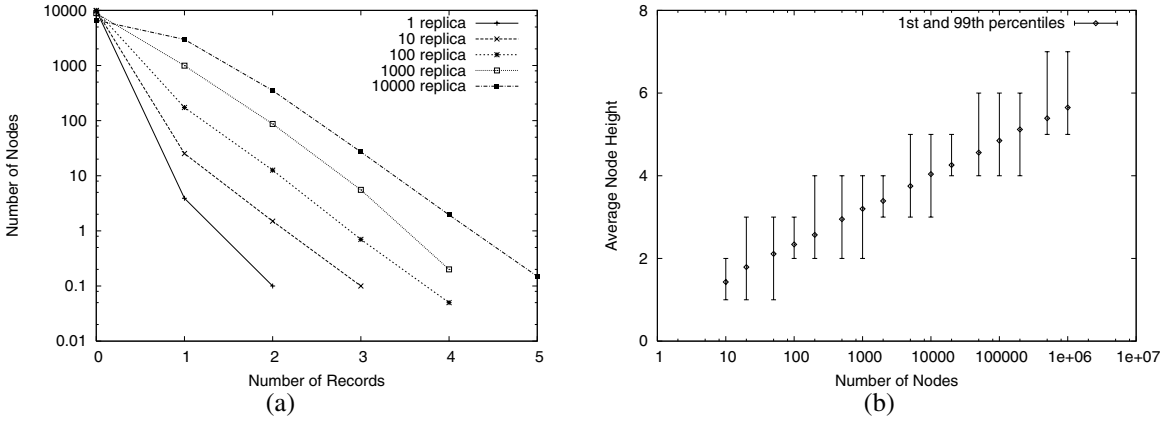

Fig. 8. (a) The storage distribution of different nodes; (b) The average RPT height with the change of number of nodes.

number of records stored on each node to evaluate scalability. The experiments are repeated 20 times and the mean values are plotted in Figure 8(a). With the increase of total replicas, the number of records on a node is slowly increased. For example, when there are 10^4 replicas, it is rare for a node to have more than three records.

The height of an RPT determines the latency of operations in SCOPE. By varying the number of nodes, we measure the level of partitions for each node, which is equal to the height of RPTs of all replicas on one node. Figure 8(b) plots the mean, the 1st, and 99th percentiles of the height of RPTs with the increase of number of nodes. The RPT heights of all keys residing in different nodes exhibit small variations, and they grow logarithmically with the increase of the number of nodes.

Assume subscribers follow a Zipf's distribution, and there are 10^4 and 10^5 keys in a 10^4-node network. The number of subscribers of i^{th} most popular key is equal to $1/i$ of total number of nodes. Figures 9(a) and (b) plot the storage load on each node when the number of total keys is 10^4 and 10^5, respectively. The storage load is measured by the number of records kept on one node. Compared with the centralized solution, SCOPE can effectively distribute the storage load to all nodes, thus avoiding the hot-spot problem. In these two experiments, the maximal records on a single node are reduced from 10004 and 10078 in the centralized solution to 105 $(1/95)$ and 387 $(1/26)$ in SCOPE, respectively.

Next we consider a query distribution obtained from Web proxy logs [1]. We randomly selected 10^4 and 10^5 hostnames, and the distribution of the number of subscribers is equal to that of requests collected during one week period (Nov. 02 - Nov. 08, 2003). Figures 9(c) and (d) plot the storage load on each node when the number of total keys is 10^4 and 10^5, respectively. Again, the maximal number of records on a single node are reduced to 418 and 1937 in SCOPE, which are 24 and 10 times less than 10098 and 19548 in the centralized solution, respectively.

B. Operation Effectiveness

In this section, we evaluate the effectiveness of new operations in SCOPE. We focus on subscribe operations only, since the other two kinds of operations (unsubscribe/update) are similar to subscribe operations. Figure 10(a) plots the dynamics of the subscribe path length with the increase of total number of nodes. SCOPE has longer paths to finish a subscribe operation than the centralized solution, because multiple representative nodes should be contacted before the primary nodes are reached. In practice, when numerous subscribers exist, the subscribe operation may terminate at a representative node, leading to a reduced path. Figure 10(b) illustrates the effects of multiple subscribers on the path length. If the number of subscribers is larger than 200 in a 10^4-node network, the path length is shorter than the centralized solution.

Assume subscribers follow a Zipf's distribution. Figures 11(a) and (b) plot the distribution of the messages sent/received by each node in a 10^4-node network with 10^4 and 10^5 keys, respectively. Although the maximal path length in SCOPE is longer than that of the centralized solution, the messages from the subscribe operations are much more evenly distributed among all nodes, instead of clogging at a few nodes. As the simulation results shown, when the number of keys is 10^4, the maximal number of messages on a single node is 412 in SCOPE, only about 1/25 of 10445 in the centralized solution. When the number of keys is 10^5, the maximal number of messages on a single node increases to 1410 in SCOPE, but still only about one seventh of 10406 in the centralized solution.

When the subscriber distribution is obtained from proxy logs [1], Figures 11(c) and (d) illustrate the distribution of the messages sent/received by each node in a 10^4-node network with 10^4 and 10^5 keys, respectively. The maximal number of messages on a single node in SCOPE is only one sixth (1906 vs. 11218) and one fourth (7925 vs. 33068) of the centralized solution in these two cases, respectively.

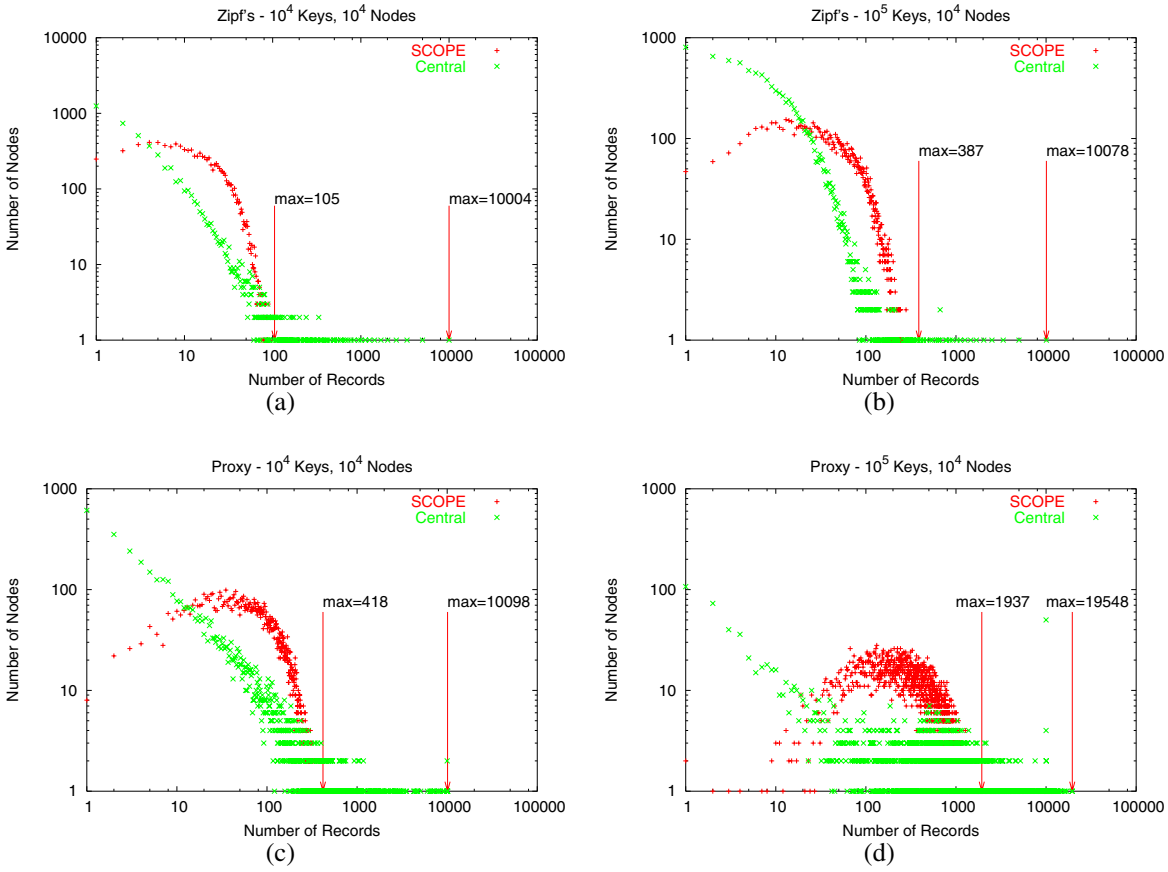

Fig. 9. With the Zipf's distribution, the number of records kept by each node in: (a) 10^4-key, 10^4-node network; (b) 10^5-key, 10^4-node network; with the distribution obtained from proxy logs, the number of records kept by each node in: (c) 10^4-key, 10^4-node network; (d) 10^5-key, 10^4-node network.

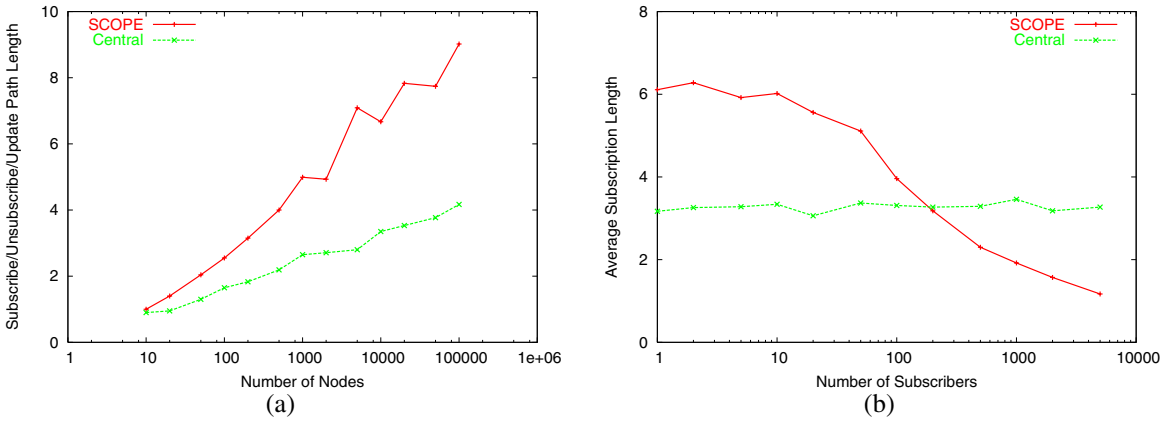

Fig. 10. (a) The subscribe operation path length in SCOPE compared with the centralized solution; (b) The changes of subscribe operation path length with the variance of number of subscribers.

C. Maintenance Cost

The maintenance cost includes node joining and leaving, and node failure recovery. Besides the maintenance routines in Pastry, node joining/leaving needs additional operations to maintain an RPT. We focus on the additional overhead induced by SCOPE. The first part of maintenance, *transferring RPT*, can be completed through a regular operation in Pastry. The second part of maintenance, *further partitioning*, needs additional operations. Figure 12(a) illustrates the number of affected nodes when a new node joins. On average, a newly-joined node only invokes 0.5 node to update its level index,

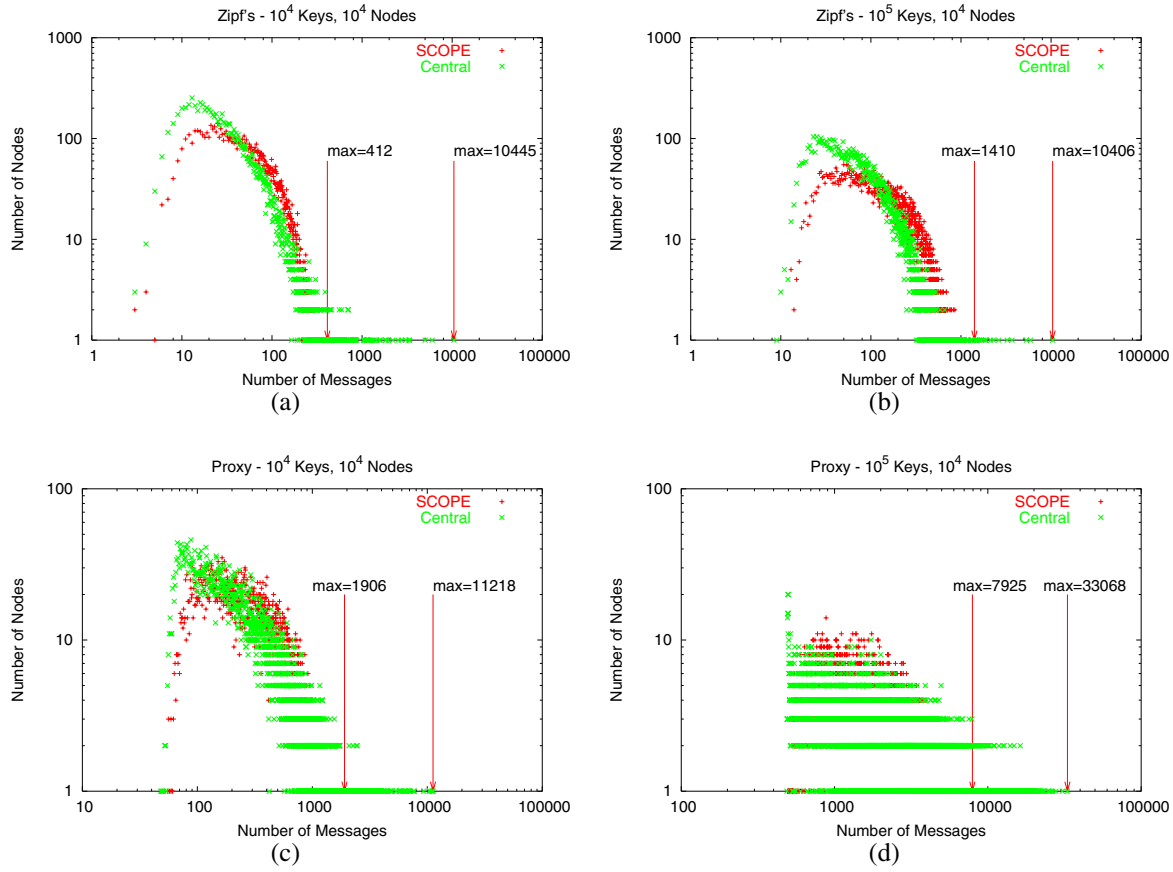

Fig. 11. With the Zipf's distribution, the distribution of messages sent/received by each node in: (a) 10^4-key, 10^4-node network; (b) 10^5-key, 10^4-node network; with the distribution obtained from proxy logs, the distribution of messages sent/received by each node in: (c) 10^4-key, 10^4-node network; (d) 10^5-key, 10^4-node network.

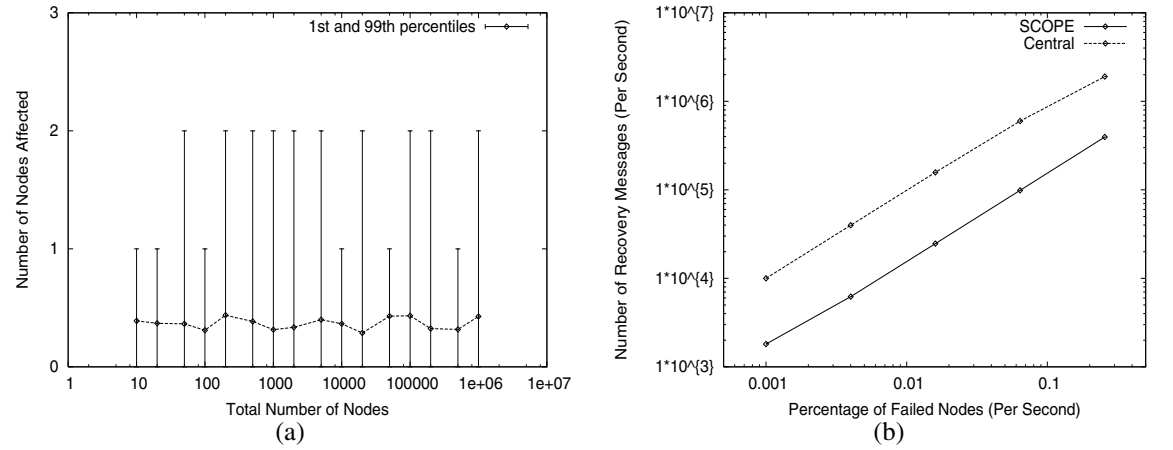

Fig. 12. (a) The number of nodes to update their level indices at a node joining; (b) The number of maintenance messages in a network with certain node failure rate.

disregarding the size of a P2P system.

One important feature in SCOPE is its efficient recovery mechanism when a node failure is detected. As in Pastry, in order to detect node failures, the neighboring nodes periodically exchange keep-alive messages. In our experiments, we only count the additional messages in the network to recover from node failures. In SCOPE, the new representative nodes communicate with the lower-level representatives to recover RPTs in case of a node failure. On the contrary, the centralized solution has to broadcast the recovery information to all nodes.

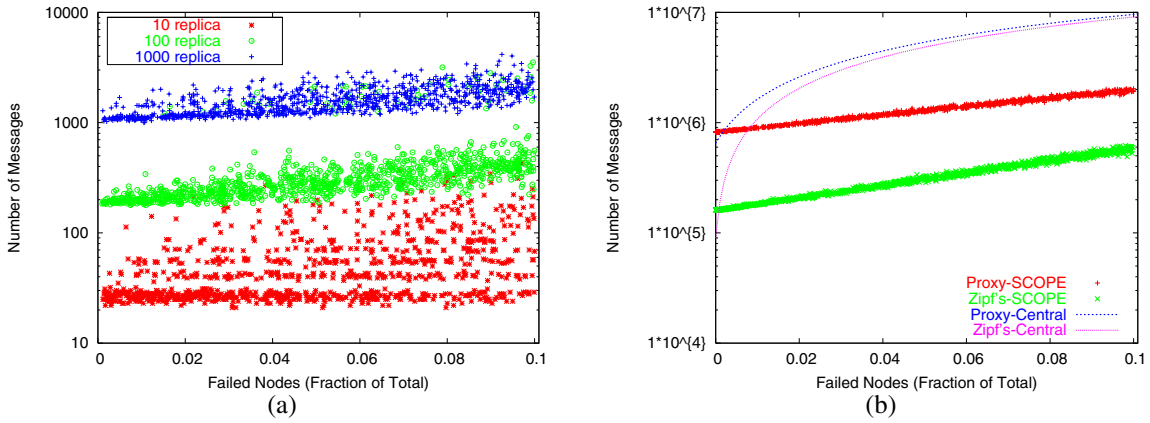

Fig. 13. (a) The number of messages for an update in a 10^4-node network; (b) The total messages for updates in a 10^4-key, 10^4-node network for both Zipf's and proxy log-based distributions.

Figure 12(b) shows the message rate in a 10^4-node network with given node failure rates. Compared with the centralized solution, SCOPE only consumes about one fifth of messages to recover under different failure rates.

D. Fault Tolerance

When a node fails, in order to propagate the update, a centralized scheme relies on broadcasting to reach the destination nodes. In contrast, SCOPE only needs to send the update notifications to representative nodes at the lower levels. We simulate a network with 10^4 nodes and 10^4 keys. Figure 13(a) plots the total number of messages of an update operation with the increase of failed nodes. Note that the update is made on an object with 10, 100, or 1000 replicas, respectively. In all cases, the number of messages for the update is proportionally increased with the fraction of failed nodes. Assume subscribers follow either Zipf's or proxy log-based distribution, Figure 13(b) plots the total number of messages if all keys are updated once. The number of messages in SCOPE is about 5%-10% of the message overhead in the centralized scheme for both Zipf's and proxy log-based distributions.

VII. Design Alternatives

Selected Representative Nodes

Frequent node joins and leaves, which is not uncommon in practice [2], could significantly degrade the performance of SCOPE. However, we can mitigate this performance degradation by pre-selecting representative nodes in RPTs. Not all nodes are eligible for being representative nodes; only those trusted and stable ones are selected as representatives for each partition. Since transient nodes join at the bottom level, the higher-level partitions are relatively stable and the cost of maintaining RPTs is minimized.

Direct Notification

In the base SCOPE protocol, the update process needs to traverse multiple partitions even if only one replica remains in the P2P system. If privacy is not a concern, SCOPE can be easily extended to record subscribers' IP addresses to shorten the latency of update notifications. If a node/partition is the first one to subscribe a key in the upper-level partition, this partition records its IP address in addition to the partition vector. When the upper level receives an update notification, it directly forwards the message to the node with the corresponding IP address without traversing the partition tree.

Dynamic Partitioning

The partition number at each level is predefined and fixed in the base SCOPE. A large number of partitions may reduce the total number of levels, thus lower subscribe/unsubscribe/update latency, but at the expense of high space overhead caused by a large number of partition vectors. On the other hand, a small number of partitions may increase the number of levels with low space overhead. Considering the tradeoff between routing latency and storage overhead, our partitioning scheme could be dynamic, in which the number of partitions is adaptively changed with respect to the popularity of a key. In this scheme, the root of the RPT for a key decides the appropriate number of partitions for that key. Since subscribers of the key do not know the number of partitions, subscribe/unsubscribe operations always start from the root RPT vectors.

VIII. Conclusion

The challenges to building a consistent P2P system are twofold: large scale and high failure rates. In this paper, we proposed a scalable, consistent structured P2P system, called SCOPE. Based on structured DHTs, SCOPE builds a replica-partition-tree for each key to distribute its replica location information among peers. In an N-node network, each peer is guaranteed to keep at most $O(\log N)$ partition vectors for a single key, regardless of the key's value and its popularity. Three new primitives, subscribe/unsubscribe/update, are introduced specifically to maintain the replica consistency. Due to hierarchical management, these operations can be committed efficiently with minimal maintenance cost. Only

$O(1)$ nodes are updated when a node joins or leaves, and only $O(\log^2 N)$ messages are transmitted to recover a node failure. Our theoretical analyses and simulation experiments have shown that SCOPE scales well with the number of nodes, maintains consistency effectively, and recovers from node failures efficiently.

Acknowledgments

We thank the anonymous referees for their constructive comments. We appreciate Bill Bynum for reading the paper and his comments. This work is supported in part by the National Science Foundation under grants CNS-0098055, CCF-0129883, and CNS-0405909, and by Ask Jeeves.

REFERENCES

[1] http://www.ircache.net.

[2] R. Bhagwan, S. Savage, and G. Voelke. Understanding availability. In *Proceedings of IPTPS'03*, Berkeley, CA, USA, Feb. 2003.

[3] M. Castro, M. Costa, and A. Rowstron. Performance and dependability of structured peer-to-peer overlays. In *Proceedings of DSN'04*, 2004.

[4] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. In *IEEE Journal on Selected Areas in Communications*, volume 20, Oct. 2002.

[5] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth multicast in a cooperative environment. In *Proceedings of the 19th ACM SOSP'03*, Lake Bolton, NY, USA, Oct. 2003.

[6] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like p2p systems scalable. In *Proceedings of ACM SIGCOMM'03*, Karlsruhe, Germany, Aug. 2003.

[7] I. Clarke, O. Sandberg, B. Wiley, and T.W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, 2001.

[8] F. M. Cuenca-Acuna, R. P. Martin, and T.D. Nguyen. Autonomous replication for high availability in unstructured p2p systems. In *Proceedings of IEEE SRDS'03*, Florence, Italy, Oct. 2003.

[9] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM SOSP'01*, Banff, Alberta, Canada, Oct. 2001.

[10] A. Datta, M. Hauswirth, and K. Aberer. Updates in highly unreliable, replicated peer-to-peer systems. In *Proceedings of IEEE ICDCS'03*, Providence, RI, USA, May 2003.

[11] S. El-Ansary, L.O. Alima, P. Brand, and S. Haridi. Efficient broadcast in structured p2p networks. In *Proceedings of IPTPS'03*, 2003.

[12] B. Gedik and L. Liu. Reliable peer-to-peer information monitoring through replication. In *Proceedings of IEEE SRDS'03*, Florence, Italy, Oct. 2003.

[13] L. Guo, S. Chen, S. Ren, X. Chen, and S. Jiang. Prop: a scalable and reliable p2p assisted proxy streaming system. In *Proceedings of IEEE ICDCS'04*, Tokyo, Japan, March 2004.

[14] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proceedings of ACM PODC'02*, Monterey, California, USA, July 2002.

[15] D. Karger, E. Lehman, F. 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 *Proceedings of ACM STOC'97*, El Paso, TX, USA, May 1997.

[16] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proceedings of ACM SOSP'03*, 2003.

[17] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, and D. Geels. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS-IX*, Cambridge, MA, USA, Nov. 2000.

[18] J. Lan, X. Liu, P. Shenoy, and K. Ramamritham. Consistency maintenance in peer-to-peer file sharing networks. In *Proceedings of IEEE WIAPP'03*, San Jose, CA, USA, June 2003.

[19] P. Maniatis, M. Roussopoulos, T.J. Giuli, D.S.H. Rosenthal, M. Baker, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proceedings of ACM SOSP'03*, 2003.

[20] A. Mohan and V. Kalogeraki. Speculative routing and update propagation: A kundali centric approach. In *Proceedings of IEEE ICC'03*, 2003.

[21] V. Ramasubramanian and E.G. Sirer. Beehive: Exploiting power law query distributions for o(1) lookup performance in peer to peer overlays. In *Proceedings of USENIX NSDI'04*, San Francisco, CA, USA, Mar. 2004.

[22] V. Ramasubramanian and E.G. Sirer. The design and implementation of a next generation name service for the internet. In *Proceedings of ACM SIGCOMM'04*, Portland, OR, USA, Aug. 2004.

[23] A. Rao, K. Kakshminarayanan, S. Surana, R. Karp, and I. Stoica. 'load balancing in structured p2p systems. In *Proceedings of IPTPS'03*, 2003.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, USA, Aug. 2001.

[25] S. Ren, L. Guo, S. Jiang, and X. Zhang. Sat-match: A self-adaptive topology matching method to achieve low lookup latency in structured p2p overlay networks. In *Proceedings of IEEE IPDPS'04*, Santa Fe, NM, USA, April 2004.

[26] M. Rodrig and A. LaMarca. Decentralized weighted voting for p2p data management. In *Proceedings of ACM MobiDE'03*, 2003.

[27] M. Roussopoulos and M. Baker. Cup: Controlled update propagation in peer-to-peer networks. In *Proceedings of 2003 USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 2003.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware'01*, Heidelberg, Germany, Nov. 2001.

[29] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale persistent peer-to-peer storage utility. In *Proceedings of ACM SOSP'01*, Banff, Alberta, Canada, Oct. 2001.

[30] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proceedings of USENIX OSDI'02*, Boston, Massachusetts, USA, Dec. 2002.

[31] C.A. Stein, M.J. Tucher, and M.I. Seltzer. Building a reliable mutable file system on peer-to-peer storage. In *Proceedings of SRDS'02*, 2002.

[32] 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. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, USA, Aug. 2001.

[33] M. Walfish, H. Balakrishnan, and S. Shenker. Untangling the web from dns. In *Proceedings of USENIX NSDI'04*, San Francisco, CA, USA, Mar. 2004.

[34] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. In *ACM Transactions on Computer Systems*, Aug. 2002.

[35] H. Yu and A. Vahdat. Consistent and automatic service regeneration. In *Proceedings of USENIX NSDI'04*, San Francisco, CA, USA, Mar. 2004.

[36] B.Y. Zhao, L. Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. In *IEEE Journal on Selected Areas in Communications*, volume 22, Jan. 2004.

[37] S.Q. Zhuang, B.Y. Zhao, A.D. Joseph, R.H. Katz, and J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proceedings of ACM NOSSDAV'01*, Port Jefferson, NY, USA, June 2001.