Locality-Aware Consistency Maintenance for Heterogeneous P2P Systems

Zhenyu Li[1,2], Gaogang Xie[1,3], Zhongcheng Li[1]
[1]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 100080, China
[2]Graduate School of the Chinese Academy of Sciences, Beijing, 100080, China
[3]INRIA-Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay, France
{zyli, xie, zcli}@ict.ac.cn

Abstract

Replication and caching have been deployed widely in current P2P systems. In update-allowed P2P systems, a consistency maintenance mechanism is strongly demanded. Several solutions have been proposed to maintain the consistency of P2P systems. However, they either use too much redundant update messages, or ignore the heterogeneity nature of P2P systems. Moreover, they propagate updated contents on a locality-ignorant structure, which could consume unnecessary backbone bandwidth and delay the convergence of consistency maintenance. This paper presents a locality-aware consistency maintenance scheme for heterogeneous P2P systems. Taking the heterogeneity nature, we form the replica nodes into a locality-aware hierarchical structure: the upper layer is DHT-based and a node in the lower layer attaches to a physically close node in the upper layer. An efficient update tree is built dynamically upon the upper layer to propagate the updated contents. Theoretical analyses and simulation results demonstrate the effectiveness of our scheme. Specially, experiment results show that, compared with gossip-based scheme, our approach reduces the cost by about one order of magnitude.

1 Introduction

In P2P systems, shared resources are replicated on several nodes to improve system reliability and availability. In addition, query results are always cached along the query path to reduce the response time of subsequence queries. Hence, replication and caching are such two common ways to improve system performance that they have been widely deployed in current P2P systems. However, existing works mainly focus on replica creation, paying little attention on the consistency maintenance. Although data objects in P2P file sharing systems, such as Gnutella [2] and KaZaA [3], are always consistent, some P2P systems, such as OceanStore [6] and Publius [1], allow users to modify their own data, which causes replicas of modified data inconsistent. On the other hand, with the rapid evolution in P2P-based applications, P2P systems will support frequent updates for contents, such as online auction, trust management [8] and remote collaboration. Inconsistency in these systems would deteriorate the system performance, or even attaint the systems. Thus, a cost-effective consistency maintenance mechanism with less convergence time is highly demanded by P2P systems.

In P2P systems, we call the node having a replica of the data indexed by key *k* as a replica node of *k*. These replica nodes constitute a group called *group_k*. When the data are modified legally on a replica node, the updated content should be delivered to all the members in *group_k* as soon as possible. The group management protocol used here should have three characteristics: 1) supporting nodes churn; 2) fault tolerance; 3) scalable. And the update method based on this protocol should satisfy following terms: 1) guaranteeing strong consistency; 2) propagating updated contents as fast as possible, or with shorter convergence time; 3) cost-effective, or with less overhead.

Centralized scheme is a straightforward way to maintain replica consistency. However, it suffers from notorious scalability and brings a single point of failure problem. Gossip-based scheme has a good quality of scalability and fault tolerance. However, it can only offer probabilistic consistency and bring a lot of redundant update messages. Another feasible way is tree-based scheme. This method has shorter convergence time and less redundant update messages. How to maintain the tree structure and improve the fault tolerance are the two most important problems in this scheme.

While all the existing works [13, 14, 15] may partially solve the consistency problem in P2P systems, in our opinion, they have one or all of the following limitations: 1) They use a lot of redundant update messages; 2) they propagate updated contents on a locality-ignorant structure; 3) They assume a homogeneous environment. Thus, these methods always have longer convergence time and consume lots of unnecessary bandwidth (e.g., bisection

1-4244-0910-1/07/$20.00 ©2007 IEEE

backbone bandwidth). In this paper, we propose a scalable, locality-aware consistency maintenance method for heterogeneous P2P systems. Replica nodes of key k are organized in a two-layer fashion: the upper layer is DHT (Distributed Hash Table) based, and a replica node in the second layer attaches to a physically close node in the upper level. An update tree is built dynamically on top of the upper layer by partitioning the identifier space and the updated content is propagated along this tree. In particular, we make the following contributions:

1) Relying on the efficient update tree based on DHT and with the aid of a scalable failure recovery method, replica consistency is achieved by propagating updated contents along the tree with less redundant update messages.
2) Taking network locality information and the heterogeneity of node capacity into consideration, our method not only has shorter convergence time but also reduces the bandwidth consumption.
3) Simulation experiments show that, compared with gossip based scheme, our approach reduces the update cost by about one order of magnitude.

The rest of the paper is organized as follows. Section 2 provides a survey of related work. Section 3 gives an overview of our scheme, followed by a detailed description of its design in section 4. In section 5, we analyze the performance of our scheme theoretically. And in section 6, we evaluate our scheme through extensive simulations. Finally, we conclude our work in section 7.

2 Related Work

Replication and caching have been adequately deployed in distributed systems. In Gnutella [2], query results are cached on the nodes along the query path to reduce response time of subsequence queries of similar objects. In CFS [10], in order to increase availability, a data block is replicated on k nodes. And in CAN [7], the popular data objects are replicated on the neighborhood nodes to achieve a load balancing. All these systems resort to a naive centralized method to maintain consistency. The consistency of web proxy caching is studied in [4] and [5]. However, in their context, the proxies are always available. Therefore, these methods are not applicable for dynamic environment, which is a prominent characteristic of P2P systems.

A. Datta et al propose a hybrid push and pull consistency maintenance scheme for highly unreliable P2P systems [13]. They take advantage of gossip as a group management protocol and update messages are rumored to other replica nodes. This is called push. And when a new replica node joins, it fetches the latest content from other nodes actively. This is called pull. While suitable for unreliable P2P systems, this method only offers probabilistic guarantee of replica consistency. In addition, it is locality ignorant and brings a large number of redundant update messages. In [14], a flooding based scheme is proposed for Gnutella-like file sharing systems. Compared with [13], this approach uses even more redundant update messages.

SCOPE, proposed in [11], is a scalable scheme for structured P2P systems. Each key is associated with a replica-partition-tree (*RPT*) for updated content delivering. Update operation can be completed in $O(\log^2 N)$ hops, and a node stores $O(\log N)$ partition vectors for a single key. In SCOPE, a node (e.g., the root node) may reside at several levels in *RPT*, which would make this node overloaded or vulnerable. Moreover, SCOPE is also locality ignorant. In addition, the maintenance cost of *RPT* is non-negligible. Our method is similar to SCOPE in tree building (i.e. by partitioning the identifier space). However, in our scheme, a replica node only appears once in the update tree and the tree is built only when an update operation is needed.

Hierarchical architecture has been implemented in Gnutella [2] and also used in [12] and [16]. In Gnutella, nodes with higher reliability and capacity are elected as super nodes. In [12], super nodes in Gnutella are organized into a structured P2P fashion to increase the hit rates of rare data objects. However, they mainly focus on locating rare data items and ignore the network locality. Cluster based scheme are proposed in [16]. However, both the upper layer and second layer are organized in a structured P2P way. And the performance of their method under node failure is not analyzed or simulated.

S. Tewari et al. [17] analyses the benefits of proportional replication in P2P networks. They focus on replica creation. In [18], a distributed membership management service is proposed for QoS sensitive P2P applications. In [15], we propose a distributed load balancing algorithm for structured P2P systems. These works are largely complementary to the work presented in this paper.

3 Overview

In this section we present a brief overview of our solution, deferring a detailed description to the next section.

In the following parts, we use Chord [9] as a representative DHT protocol for analysis and description, but it is straightforward for other DHT protocols.

Fig.1 captures the hierarchical model in our scheme. Replica nodes are organized in a two-layer architecture. The upper layer is Chord-based and consists of more reliable and powerful nodes. We call the replica nodes in the upper layer as *Chord Replica Nodes* (*CRN*s). Each node at the second (lower) layer attaches to a physically close CRN, and nodes at this layer are denoted as *Ordinary Replica Nodes* (*ORNs*).

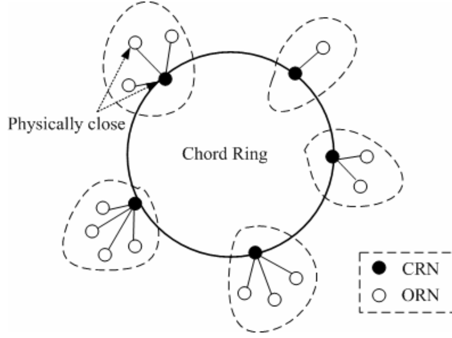

Fig.1: Hierarchical structure. Powerful replica nodes, labeled as CRNs (Chord Replica Nodes), are organized in a Chord ring. Ordinary replica nodes (ORNs) attach to physically close CRNs.

An update operation initiated by an ORN is first submitted to its corresponding CRN through an update request message. When the CRN receives the update request message or initiates an update operation by itself, it dynamically builds an update tree on the upper layer, rooted by the CRN itself, by partitioning the Chord identifier space. Then updated content is delivered along this update tree in a top-down fashion. In addition to transferring the updated content to the child replica nodes on the update tree, a CRN also delivers the updated content to the ORNs attached to it. Since CRNs are physically close to the ORNs attached to it, this enables fast convergence on consistency maintenance and saves the bandwidth consumption, or our scheme is *locality aware*. After the update operation completes, the update tree is destroyed in a bottom-up fashion.

From above description, we find that the two key issues of our scheme are: 1) how to generate locality information and how to use this information to cluster replica nodes; 2) how to build an efficient update tree and how to recover from a failure. Some notations used in later sections are showed in Table I.

4 Design

In this section we describe the detail of our consistency maintenance scheme. We begin with locality information generation and hierarchy architecture construction based on this locality information. Then we describe how to build the update tree and how to propagate updated content. We end this section with the maintenance and failure recovery mechanism of our scheme.

4.1 Generating locality information

We take advantage of landmark clustering scheme [19] to generate locality information. In this algorithm, m nodes are picked up randomly from the Internet as landmark nodes. A replica node measures the distance to these landmark nodes and obtains a landmark vector $< d_1, d_2, ..., d_m >$. Replica nodes use their landmark vectors as coordinates in an *m-dimension* Cartesian space, which is called landmark space. The intuition behind is that physically close replica nodes are likely to have similar or close landmark vectors, and thus close to each other in the landmark space. As pointed out in [19], landmark clustering is a coarse-grain approximation and not very effective in differentiating nodes within close distance. However, we just simply use landmark clustering in our scheme. This is mainly because coarse-grain information is adequate for our scheme and the simulation experiments also show that approximate information works well.

As described later, we only need a one dimension key to represent node's position in the landmark space. This key is denoted as *landmark number*. Thus, another challenge is related to map m-dimension landmark vectors to one dimension landmark numbers while preserving the network locality. Space-filling curves [20] are good choices for this problem. Space-filling curves map an m-dimension point to a one-dimension point without loss of proximity, or points that are close in m-dimension space are also close in one-dimension space. One example of space-filling curves is Hilbert curve.

We partition the landmark space into 2^{mx} smaller grids with equal size, where x is the order of Hilbert curve and controls the number of grids used to partition the landmark space. Then we fit a Hilbert curve on the landmark space to

TABLE I NOTATIONS

CRN	Chord Replica Nodes, replica nodes in the upper layer.
ORN	Ordinary Replica Nodes, replica nodes in the second layer.
d	Average out degree of CRN in the update tree.
C_n	Capacity of node n, defined as the maximum number of replica nodes to which n is able to send updated content concurrently.
l_n	Landmark number of replica node n.
CS	The maximum number of ORNs that a CRN can be attached to, or the maximum cluster size.
w	The number of backup CRNs stored on a ORN
R_0	The basic search radius for locating nearby CRNs
N	Total number of replica nodes of key k
N_{CRN}	Number of Chord replica nodes of key k
h	Height of an update tree.

number each grid. Replica node whose landmark number falls into grid l has the landmark number as l. Due to the proximity preserving property of Hilbert curve, closeness in landmark number indicates physical closeness.

4.2 Constructing Hierarchical Architecture

Each CRN publishes its landmark number on the upper layer, based on Chord protocol. That is to say the information of a CRN and its landmark number l is stored on the successor of identifier l. Recall that, in a DHT, if two objects have similar/close DHT keys, then these two objects will be stored close to each other in the DHT overlay. Therefore, thanks to the proximity preserving property of Hilbert curve, the information of two physically close nodes is stored closely in the Chord ring.

As in [9], we assume replica nodes learn the information of an existing CRN (denoted as n_0) in the upper layer by some external mechanism. And replica nodes can evaluate their capacities by their selves. When joining existing group, a new replica node of key k first tries to find a close CRN to attach, or become an ORN. If this operation fails, it then joins as a CRN. In detail, each new replica node runs the routine *join_group()* as described in Fig.2.

To find nearby replica nodes, a new joining replica node search a range, through a well-known replica node n_0, with its landmark number as centre and T as radius. And

$$T = \begin{cases} R_0 \times \alpha / c & c \geq d \\ \infty & otherwise \end{cases} \quad (1)$$

where R_0 is the basic search radius, α is a design parameter and c is the capacity of the new join node. In practice, α can be set as the expectation value of node capacity. The intuition behind formula (1) is that the bigger a node's capacity is, the higher probability it is a CRN. Recall that node capacity is measured by the maximum number of replica nodes to which it is able to send updated content concurrently, and a replica node at least should propagate updated content to d (on average) child replica nodes along the update tree. Thus, when a node's capacity

rn.**join_group**()
1: *rn* measures the distance to landmark nodes, and computes its landmark number l_{rn}
2: $crn = n_0.find_successor(l)$ /* *find_successor* is provided by Chord[9]*/
3: vector *v_CRN* ← *crn*.GetCRNs(l_{rn}, C_{rn})
4: **while**(!*v_CRN*.empty())
5: crn_1 ← the node with landmark number closest to l_{rn}
6: **if**(crn_1.Can_Attach())
7: *rn* attaches to crn_1 and selects other *w* CRNs randomly from *v_CRN* as backup CRNs
8: **return**
9: **else**
10: *v_CRN*.erase(crn_1)
11: **end while**
12: **if**(*v_CRN*.empty()) /*join as an CRN*/
13: *rn* joins the upper layer as a CRN based on Chord protocol and publishes its landmark number.

crn.**GetCRNs**(landmark_number *l*, capacity *c*)
1: $T = \begin{cases} R_0 \times \alpha / c & c \geq d \\ \infty & otherwise \end{cases}$
2: *crn* finds the CRNs whose landmark number is in the range $R = \{r : |r - l| < T\}$, and pushes these CRNs back to the vector *v_CRN*
3: **return** *v_CRN*

crn.**Can_Attach**()
1: **if** (the number of ORNs attached to *crn* < *CS* and (*d* + the number of ORNs attached to *crn*) < C_{crn})
2: **return** 1
3: **else**
4: **return** 0

Fig.2: New replica node joining algorithm

is smaller than d, it is not capable for a CRN. We achieve this by having the search range infinite. Since each CRN in the Chord ring maintains a continuous identifier space and the information of CRNs with close landmark numbers is stored closely in Chord ring, finding nearby CRNs should be fast. Note that although we only use capacity as a metric for replica node joining, it is easy to combine history uptime and other metrics.

Following above joining mechanism, a more powerful node still has a probability to become an ORN of a close CRN with less capacity. To this end, each ORN periodically evaluates itself to decide if it should become a CRN based on some criterions, such as CPU speed, bandwidth and uptime. If an ORN is CRN capable, it rejoins as a CRN.

Intuitively, a bigger cluster size (*CS*) will help the nearby replica nodes being grouped into one cluster with higher probability. However, a bigger cluster size may also increase the number of ORNs attached to a CRN. This not only overloads the CRN, but also increases the cost of recovering from a failure of the CRN. Thus, we need a compromise here. Another thing worth pointing out is that when a new replica node joins as an ORN, it stores other w close CRNs as backup CRNs. This is used for increasing the system fault tolerance. We defer the detail in later sections. Maintenance and failure recovery of this hierarchical structure are described later as well.

4.3 Propagating Updated content

As mentioned before, an ORN submits the update operation to its corresponding CRN. When a CRN receives an update request message or initiates an update operation by itself, it dynamically builds an update tree on the upper layer, rooted by the CRN itself, by partitioning the Chord identifier space. Initially, the CRN, crn_i, holds the whole identifier space. This identifier space is partitioned into d parts with equal size. We choose the first CRN of each part as the representative node to hold the identifier space of this part and set these d representative CRNs as the children of crn_i. Each part is further partitioned into d parts with equal size, and so on, until there is only one CRN in this identifier part. The pseudo code is listed in Fig.3. The function *find_successor(id)*, provided by the Chord protocol, is used to find the successor node with the *id*. The function of *get_rpn(region)* is to get the representative node of *region*.

In Fig.4, we show a Chord ring consisting of 10 CRNs and the corresponding update tree.

After building the update tree on top of the Chord ring, updated content is propagated along this tree in a top-down fashion. In addition to transferring the updated content to the child replica nodes on the update tree, a CRN also delivers the updated content to the ORNs attached to it. When receiving a latest updated content, a replica node

X.**region_partition**(*region_x*)

1: if (X.id + 1 > *region_x*.end)

/*There is only one node (say X) in this region.*/

2: **return**;

/*There is no node between [regon_x.start, X.id). And we prune node X from further partition */

3: *region* ← (X.*id* + 1, *region_x*.end);

4: Split *region* into d partitions with equal size

5: **for** $i=1$ **to** d{

6: *region[i]* ← *the i-th partition;*

7: $RPN_{region[i]} = X.get_rpn(region[i])$;

8: **if** ($RPN_{region[i]} != NULL$){

9: $X.children = X.children \cup RPN_{region[i]}$;

10: $RPN_{region[i]}$.*region_partition* (*region*[i]);

11: }

12: } /* *end of for i=1…* */

X.**get_rpn**(*region*)

1: *id* ← *first ID of this region;*

2: *node* ← X.*find_successor* (*id*);

3: **if** (node.*id* $\notin$ *region*)

4: **return** *NULL*;

5: **return** *node*;

Fig. 3: Algorithm for building update tree

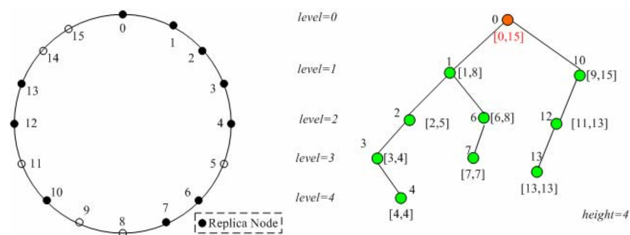

Fig.4 Chord ring and a update tree built based on this ring

checks and verifies the data, updates its content and forwards the updated content if necessary. Due to the tree structure, each replica nodes will receive $O(1)$ update messages. Moreover, since CRNs are physically close to the ORNs attached to it, consistency maintenance converges fast and bandwidth is greatly saved. After the update operation completes, the update tree is destroyed in a bottom-up fashion. Failure recovery mechanism during the update period is detailed later.

The reason why we build the update tree dynamically and destroy it after the update operation is that, to fully take advantage the system resources, we should use multiple update trees to propagate updated contents.

Otherwise, if we use only one update tree, most nodes are leaf nodes which would make no contribute for content delivering. In our opinion, the maintenance cost of multiple update trees is larger than the cost of building an update tree while necessary, especially in dynamic P2P systems.

4.4 Maintenance of Hierarchical Structure

If a replica node joins after a period of leaving, it may miss updated contents during this period. As in [13], we use a pull scheme: after rejoining the hierarchical structure, CRNs ask its successor for latest content and ORNs ask its CRN for latest content.

When a replica node does not need to keep the replica up to date any more, it leaves. Leaving mechanism for ORNs is really simple: just sends a *Leave Message* to the corresponding CRN, and then leaves. When receiving the *Leave Message* from the ORN attached to it, the CRN deletes the stored information for this ORN. If a CRN leaves, it selects the most reliable, powerful replica node from its ORNs to replace itself. The selected ORN rejoins as a CRN and takes over other ORNs attached to the leaving CRN. If there is no ORN powerful enough to replace the leaving CRN, the cluster is split into several small groups and powerful nodes are chosen from each group to act as CRNs for these groups. Then the leaving CRN leaves the upper layer based on the Chord protocol and unsubscribe its landmark number information.

Failure of an ORN can be detected by its corresponding CRN and this has little effect on the structure. When a CRN fails, we resort to Chord protocol to recover the upper layer. Failures of CRNs can also be detected by their ORNs by periodically message exchanging. The ORNs previously attached to the failing CRN first try to attach to one of its backup CRNs. If all the backup CRNs of an ORN fail, the ORN rejoins the hierarchical structure. Note that if the failures of CRNs are independent, this scenario is rare. It is worth pointing out that attaching to a backup CRN may impair the proximity effect as the new CRN may not be the closest CRN in the upper layer. However, the simulation results show that this has little effect and our scheme is as effective as usual.

To ensure that the backup CRNs are always the available ones in the upper layer, ORNs checks the availability of their backup CRNs periodically. If $w/2$ its backup CRNs are not available any more, an ORN sends a *Backup CRNs Query Message* through its CRN to find nearby CRNs filling up its backup CRNs list. This query message is similar to the joining message in terms of locating nearby CRNs.

4.5 Maintenance when Propagating Updated Content

Recall that the update tree is built dynamically when there is an updated content needed to be disseminated, and this update tree is destroyed when the update operation completes. Although the convergence time of our scheme is really short, it still has a probability that a replica node leaves and fails during this short period. Because ORNs are not in the update tree, leaving and failing of ORNs have no effect on propagating updated content. When a CRN in an update tree leaves, in addition to leaving the group according to the leaving scheme, it asks its parent to rebuild the sub tree rooted by the leaving CRN and leaves according the leaving mechanism introduced earlier. Given that CRNs have a good quality of availability and the convergence time of our scheme is short, we think our leaving scheme is reasonable.

To improve the fault resilience of our scheme, we use an acknowledgement scheme. A CRN acknowledges its parent CRN as soon as it has received acknowledgements from all its children CRNs. When receiving an acknowledgement from a child CRN, the parent CRN deletes the information (e.g. ID, address and region information) related to this child CRN. This is a recursive process from bottom to top. And the leaf nodes acknowledge their parents as soon as sending the updated content to their ORNs. To ensure a strong consistency, each CRN in the update tree sets a timeout when it forwards the updated content to their child CRN. If a CRN does not receive all the acknowledgements when the timeout expires, it rebuilds the sub tree rooted by the failure child node and retransmits the updated content along that sub tree.

To isolate node faults, the timeout intervals of CRNs decrease exponentially with the increasing of the level at which the CRNs reside.

$$timeout_{crn} = T_0 \times e^{-level_{crn}}, \quad 0 \le level_{crn} \le h, \quad (2)$$

where h is the height of update tree and T_0 is a design parameter. In this way, a CRN failure is restricted to the sub tree rooted by the parent of failure CRN with high probability.

We can deduce that CRNs' leavings, especially failures, cost more than the ORNs' as it plays a more important role. Therefore, CRNs must be not only more powerful but also more available.

5 Analysis

In this section, we analyze the performance of our scheme from several perspectives, such as the maintenance cost, the performance and cost of update tree and the efficiency of failure recovery mechanism.

5.1 Analysis of Replica Nodes Joining and Leaving

When a new replica node joins, to search the close CRNs on the upper level, $O(\log N_{CRN})$ Chord query messages are required. If the new node joins as a CRN, another $O(\log^2 N_{CRN})$ Chord joining messages are required. Thus, when a new replica node joins, on average,

$$\#Msg_{join} = \log N_{CRN} + p_{CRN} \times \log^2 N_{CRN} \quad (3)$$

messages are required, where p_{CRN} is the probability that a new replica node joins as an CRN.

According to the leaving mechanism mentioned before, an ORN leaving only uses $O(1)$ message (i.e. notifying its CRN). Suppose that a CRN leaving causes its cluster split into s smaller group, then a CRN leaving uses $O(\log^2 N_{CRN})$ Chord leaving messages and causes s nodes to rejoin as CRNs. The average number of messages used by a replica node leaving is

$$\begin{aligned}&\#Msg_{leaving}\\ &= 1\times\frac{N-N_{CRN}}{N}+\frac{N_{CRN}}{N}\times(\log^2 N_{CRN}+s\times\log^2 N_{CRN}) \quad (4)\\ &= 1+\frac{N_{CRN}}{N}\times\left[(s+1)\log^2 N_{CRN}-1\right]\end{aligned}$$

5.2 Analysis of Update tree

In our scheme, to propagate the updated content to all the alive replica nodes, an update tree should be built first, and then the updated content is disseminated along with the update tree. Thus, the update time needed by an update operation consists of two parts: the time for building update tree and the time for propagating updated content.

Theorem 1: The average height of a d-ary update tree is $O(\log_d N_{CRN})$, and a CRN resides only in one level on the update tree.

proof: Suppose the identifier length is v, that is to say the whole identifier space is 2^v. Each partition generates d smaller equally-sized regions, each with size of 1/d of the previous partition region. After $\log_d N_{CRN}$ time partition, the generated region is reduced to $2^v / d^{\log_d N_{CRN}} = 2^v / N_{CRN}$. In Chord, nodes are distributed on the ring randomly. Thus, the average number of nodes in the region with size of $2^v / N_{CRN}$ is 1. This is the termination condition of our partition method. So, the average height of Chord tree is $\log_d N_{CRN}$.

Note that current node ID is excluded from the region for further partition (line 3 in function *region_partition* illustrated in Fig.3). Thus, every node resides at only one level in the updating tree.■

Note that a CRN delivers the updated data to its child CRNs and its ORNs concurrently. Therefore, after building the update tree, updated content can be propagated to all the replica nodes in $O(\log_d N_{CRN})$.

Lemma 1: If an M-node Chord ring is partitioned into r regions with equal size, then the successor of the first ID of the partitioned region can find the successor of a key in this region in $O(\log \frac{M}{r})$ logical hops, on average.

proof: Suppose the identifier length is v, that is to say the whole identifier space is 2^v and node n_i is successor of the first ID region r_i. Now, we analyze the logical hops required to find the successor of key k, where k is in region r_i and $k > n_i$. According to the Chord protocol, each hop halves the distance from the query node to the successor of k. Thus, after $\log \frac{M}{r}$ hops, the distance between the query node to the successor of k is at most $\frac{2^v / r}{2^{\log(M/r)}} = 2^v / M$. Due to the random distribution of nodes in the Chord ring, the average number of nodes in the region with size of $2^v / M$ is 1. Thus, n_i has found the successor of key k.■

Theorem 2: A d-ary update tree can be built in $O(\log^2 N_{CRN})$ logical hops, on average.

proof: See [22] for details. ■

Thus, an update operation completes in $O(\log^2 N_{CRN} + \log N_{CRN})$ time. Noting that N_{CRN} is much smaller than the total number of replica nodes, we think our scheme is time-effective and have a good quality of scalability as well.

Now, we analysis the Chord query messages used by building an update tree.

Theorem 3: With N_{CRN} Chord replica nodes, to build a d-ary update tree, $O(N_{CRN} \times \log d)$ Chord query messages will be used.

proof: See [22] for details. ■

From Theorem 3, the average number of query messages per CRN to build a d-ary update tree is $O(\log d)$. Therefore, a smaller d is preferred. However, from Theorem 1 and Theorem 2, decreasing d will increase the convergence time of the update operation. We need a trade-off on the selecting of d.

5.3 Analysis of Failure Recovery

Theorem 4: If failures of replica nodes are independent and random, then a failure of a replica node would cause $O(\log N_{CRN})$ redundant updated messages at most, on average.

proof: A failure of an ORN does not cause redundant updated messages. If a CRN fails, we should rebuild the sub tree and retransmit the updated content. In the worst case, all the descendant CRNs of the failing CRN and their ORNs have received updated content before, thus all the retransmitted messages are redundant. Suppose that there

are $\frac{N - N_{CRN}}{N_{CRN}}$ ORNs attached to a CRN on average, and the failure CRN resides in the *i*-th level in the updating tree with probability of $p_i = \frac{d^i}{N_{CRN}}$. Then, on average, the total number of redundant updated messages caused by a failure of a replica node is $\#Msg_{rdt}$ at most.

$$\begin{aligned}\#Msg_{rdt} &= \frac{N_{CRN}}{N} \times \sum_{i=1}^{h-1}\left[p_i \times (1 + \frac{N - N_{CRN}}{N_{CRN}}) \times \sum_{j=1}^{h-i} d^j\right] \\ &= \sum_{i=1}^{h-1}\left[\frac{d^i}{N_{CRN}} \times \frac{d\left[d^{h-i} - 1\right]}{d-1}\right] \\ &= \frac{d}{d-1} \times \frac{1}{N_{CRN}} \times \sum_{i=1}^{h-1}\left[d^h - d^i\right] \\ &< \frac{d}{d-1} \times \frac{1}{N_{CRN}} \times \sum_{i=1}^{h-1} d^h \\ &= \frac{d}{d-1} \times (\log_d N_{CRN} - 1) \\ &\approx O(\log_d N_{CRN} - 1)\end{aligned}$$

■

Thus, our failure recovery scheme also has a good quality in terms of scalability.

6 Performance Evaluation

We evaluate our consistency maintenance method by extensive simulation experiments. In the simulations, we randomly choose 15 landmark nodes from the internet topology. Node capacities are generated using a Pareto distribution with the shape parameter $a = 2$ and the scale parameter $b = 16$. Thus, the expectation value of node capacity is 32 and the standard deviation is infinite. The basic search radius for locating nearby CRNs (R_0) is set to 20 and the design parameter α is set to 32. The number of backup CRNs of a ORN is 4, or $w = 4$. Finally, we set the average out degree of the update tree to 8, or $d = 8$, and the size of update message is set to 1K bytes by default.

In our simulations, if an ORN is two times more powerful than its CRN, it is promoted as a CRN. And each ORN determines if it is CRN capable every 10 unit time. In practice, we should take uptime and other metrics into consideration.

To evaluate the efficacy of our proximity-aware scheme, two different transit-stub topologies are generated by GT-ITM [21]. Both topologies have about 2,500 nodes. We set the number of nodes in the P2P system as 2,400, and replica nodes are chosen randomly from these nodes.

- ts2.5k-small: 4 transit domains each with 4 transit nodes, 5 stub domains attached to each transit node, and 30 nodes in each stub domain on average.
- ts2.5k-large: 70 transit domains each with 4 transit nodes, 4 stub domains attached to each transit node, and 2 nodes in each stub domain on average.

Intuitively, "ts2.5k-large" has a larger backbone and sparser edge network (stub) than "ts2.5k-small". And "ts2.5k-large" represents a situation in which the replica nodes scattered in the entire Internet. We assign different distance to the edge according to the edge type: the distance of intra-domain edge is 1 hop of unit of latency; the distance of the edge between transit and stub domain is 5 hops of units of latency; and the distance of inter-transit edge is 25 hops of units of latency.

We also compare our scheme with the gossip-based hybrid push and pull scheme [13]. To achieve a fair comparison, the node *fanout* in gossip-scheme is set to 8, and updated content stop rumoring when 95% replica nodes have received updated content. And the last thing worth pointing out is that each data point in our plots represents the average value of 10 trials.

6.1 Number of Chord replica nodes

The number of Chord replica nodes determines the convergence time and failure recovery cost of an update operation. Intuitively, increasing max cluster size (*CS*) will increase the number of ORNs attached to a CRN and decrease the number of CRNs. When *CS* is equal to 0, the number of CRNs is equal to the number of total replica nodes. Fig.5 illustrates the number of Chord replica nodes while varying *CS*. The number of CRNs (N_{CRN}) decreases with the increasing of *CS*. However, increasing *CS* from 16 to 32 has little effect on N_{CRN}. This is mainly because there are not enough replica nodes to be grouped in one cluster. We also see N_{CRN} is about one order of magnitude less than the total replica nodes and increases slowly with the total number of replica nodes of a key. By default, in our experiments, *CS* is set to 16.

6.2 Number of Messages for an update operation

Fig.6 shows the average number of messages per replica node used for an update operation. For gossip-based scheme, only the update messages are counted in. While for our scheme, the Chord query messages for building the update tree is also included. The gossip-based scheme uses about 5.5 update messages per node, because a replica node may receive update messages from several other replica nodes. When *CS* is 0, or without hierarchical structure, the number of messages used is about 3.5 on average, two times more than the number of messages used in our hierarchy scheme, which uses only about 1.5 on average. This is mainly because much more Chord query messages are used for building the update tree when *CS* is 0. Thus, our locality-aware scheme is more efficient than gossip-base scheme in terms of the number of messages used for an update operation.

6.3 Cost for an update operation

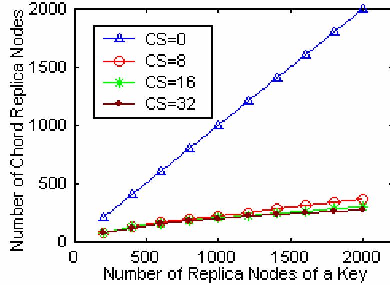

Fig.5: N_{CRN} with different scale and different max cluster size.

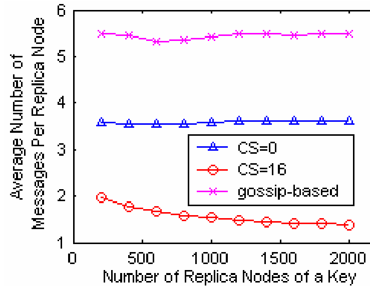

Fig.6: Average number of messages for an update operation. Including the Chord query messages for building the update tree.

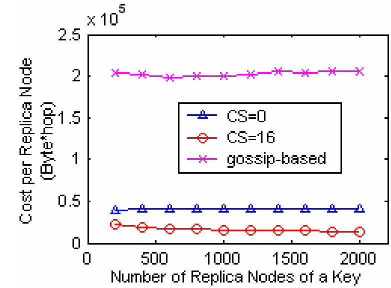

Fig.7: Average cost per replica node for an update operation in "ts2.5k-small" (byte*hop)

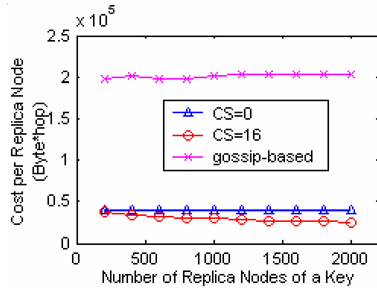

Fig.8: Average cost per replica node for an update operation in "ts2.5k-large" (byte*hop)

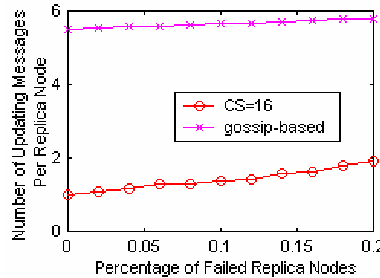

Fig.9: Number of update messages per node for an update operation with 1000 replica nodes

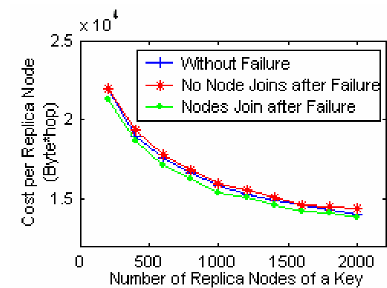

Fig.10: Cost per replica node for an update operation under churn ($CS = 16$)

Fig.7 and Fig.8 illustrate the average cost per replica node for an update operation in "ts2.5k-small" and "ts2.5k-large", respectively. The cost of an update operation *Cost(update)* is defined as follows:

$$Cost(update) = \sum_{i=1}^{u} sizeof(message) \times dst ,$$

where u is the number of messages for an update operation and *dst* denotes *message* delivered at the distance of *dst* hops. We set the size of update message as 1k and the size of query message as 27 bytes (20 bytes for querying ID, 6 bytes for address information of source node and 1 byte for marking). For our scheme, the Chord query messages for building the update tree are also included when computing the cost. We can see that for both schemes, the average cost per replica node is almost unchanged for different scales.

Compared with gossip-based scheme, our locality-aware scheme ($CS = 16$) reduces the average cost per node by about one order of magnitude, for both "ts2.5k-small" and "ts2.5k-large". When considering the scheme without hierarchical structure ($CS = 0$), the locality-aware scheme ($CS = 16$) is also more effective in terms of the update cost. However, the cost reduction in "ts2.5k-large" is not as significant as in "ts2.5k-small". This is explained by the fact that nodes are scattered in the entire network in "ts2.5k-large", and the number of replica nodes belonging to the same domain is relatively small.

6.4 Fault Tolerance

When a CRN fails, we need to retransmit the updated content. This may bring some redundant update messages. From Fig.9, we see that the number of update messages increases proportionally with the percentage of failed replica nodes. Although the failures of replica nodes have a relative small effect on the gossip-based scheme, the number of update messages used by their scheme is still two times more than ours, even when 20% replica nodes fail.

6.5 Impact of replica node churn

In this set of experiments, we evaluate our scheme under churn. First, there is no node failure. Then, we have 10% replica nodes fail. Finally, we have some other replica nodes join and the number of joining replica nodes is the same as the number of failing replica nodes. For each circumstance, the costs of update operations are computed and the average values are plotted in Fig.10. We see that after 10% replica nodes failing, the cost per replica node is slightly higher than the cost in other two cases. This is mainly because, when a CRN fails, its ORNs select other CRNs as their new CRNs, and the new CRN is always farther away than the original CRN. After replica nodes

rejoin, the cost per node drops back, even smaller than the case without any failure. We can deduce that our scheme is resilient to replica nodes churn.

7 Conclusions

This paper presents a novel, scalable consistency maintenance scheme for heterogeneous P2P systems. Replica nodes of a key are organized in a two-layer locality-aware hierarchy model and the upper layer is DHT-based. We mainly focus on fast delivering updated contents to all the replica nodes with low cost consumption. To achieve these goals, we group nearby replica nodes into a cluster and build update trees dynamically to propagate the updated contents. An efficient failure recovery mechanism is also proposed to improve fault tolerance. On average, for N replica nodes of a key with N_{CRN} upper-layer replica nodes, an update operation completes in $O(\log^2 N_{CRN})$ time and only $O(N)$ updated messages are required. Theoretical analyses and simulation results have shown that our scheme has a good quality in terms of scalability and fault tolerance. And specially, compared with gossip-based scheme, our scheme reduces the cost consumption by about one order of magnitude.

In future work, we will examine our scheme with other capacity distributions and do more comparative studies with other mechanisms.

Acknowledgement

This work is supported by the National Natural Science Foundation of China under Grant No.60403031 and No. 90604015, by the National High-Tech Research and Development Plan of China under Grant No. 2005AA121560, and funded by France Telecom R&D and Research Institute of China Mobile.

References

[1] Waldman M, Rubin AD, Cranor LF. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proceedings of the 9th USENIX Security Symposium*, 2000.

[2] Gnutella, http://gnutella.wego.com/

[3] KaZaA, http://www.kazaa.com

[4] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom* 2000.

[5] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of USEITS*, 1999.

[6] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance-free global data storage. *IEEE Internet Computing*, 5(5), 2001

[7] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A scalable content-addressable network. In *Proceedings of SIGCOMM 2001*, pages 161–172, San Diego, CA, USA, August 2001.

[8] K. Aberer and Z. Despotovic. Managing Trust in a Peer-2-Peer Information System. In *CIKM*, 2001.

[9] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM 2001*, pages 149–160, San Deigo, CA, USA, August 2001

[10] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica, Wide-Area Cooperative Storage with CFS, in *Proc. 18th ACM Symp. Operating Systems Principles (SOSP)*, pp. 202-215, Oct. 2001.

[11] X. Chen, S. Ren, H. Wang, X. Zhang, SCOPE: Scalable consistency maintenance in structured P2P systems. *In Proceedings of the IEEE INFOCOM 2005*.

[12] Boon Thau Loo, Ryan Huebsch, Ion Stoica, Joseph M. Hellerstein. The case for a hybrid P2P search infrastructure. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS04)*, San Diego, CA, February 2004.

[13] 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.

[14] 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.

[15] Zhenyu Li, Gaogang Xie, "A Distributed Load Balancing Algorithm for Structured P2P Systems", *11th IEEE Symposium on Computers and Communications (ISCC'2006)*, Italy, June 26-29, 2006.

[16] Ruixiong Tian, Yongqiang Xiong, Qian Zhang, Bo Li, Ben Y. Zhao and Xing Li. Hybrid Overlay Structure Based on Random Walk. *4th International Workshop on Peer-To-Peer Systems*. Ithaca, New York, USA. February 2005.

[17] Saurabh Tewari and Leonard Kleinrock. Proportional Replication in Peer-to-Peer Networks. In *Proceedings of IEEE INFOCOM 2006*, April 2006

[18] Jin Liang and Klara Nahrstedt, RandPeer: Membership Management for QoS Sensitive Peer-to-Peer Applications, in *Proceedings of IEEE Infocom 2006,* April, 2006

[19] Z. Xu, C. Tang, and Z. Zhang. Building Topology-Aware Overlays using Global Soft-State. In ICDSC'2003, May. 2003.

[20] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmaier, Space Filling Curves and Their Use in Geometric Data Structures , Theoretical Computer Science, 181, 1997, pp. 3-15.

[21] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In Proceedings of INFOCOM 1996, volume 2, pages 594–602, San Francisco, CA, USA, March 1996

[22] Zhenyu Li, Gaogang Xie, Zhongcheng Li, Locality-aware consistency maintenance for heterogeneous P2P systems. *Technical Report*, Oct. 2006