A Balanced Consistency Maintenance Protocol for Structured P2P Systems*

Yi Hu, Min Feng, Laxmi N. Bhuyan
Department of Computer Science and Engineering
University of California at Riverside, CA, U.S.A
yihu, mfeng, bhuyan@cs.ucr.edu

Abstract—A fundamental challenge of managing mutable data replication in a Peer-to-Peer (P2P) system is efficiently maintaining consistency under various sharing patterns with heterogeneous resource capabilities. This paper presents a framework for balanced consistency maintenance (BCoM) in structured P2P systems. Replica nodes of each object are organized into a tree for disseminating updates, and a sliding window update protocol is developed to bound the consistency. The effect of window size in response to dynamic network conditions, workload updates and resource limits is analyzed through a queuing model. This enables us to balance availability, performance and consistency strictness for various application requirements. On top of the dissemination tree, two enhancements are proposed: a fast recovery scheme to strengthen the robustness against node and link failures; and a node migration policy to remove and prevent the bottleneck for better system performance. Simulations are conducted using P2PSim to evaluate BCoM in comparison to SCOPE [2]. The experimental results demonstrate that BCoM significantly improves the availability of SCOPE by lowering the discard rate from almost 100% to 5% with slight increase in latency.

I. INTRODUCTION

1 Structured P2P systems have been effectively designed for wide area data applications. While most of them are designed for read-only or low-write sharing contents, a lot of promising P2P applications demand support for mutable contents. This requires a consistency solution to work efficiently in such dynamic environment.

P2P systems are typically large, where peers with various resource capabilities experience diverse network latency. Also, their frequent joining and leaving make the P2P overlay failure prone. Neither sequential consistency [6] nor eventual consistency [13] individually works well in P2P environment. It has been proved [5] that among the three properties, atomic consistency, availability and partition-tolerance, only two can be satisfied at a time. Applying sequential consistency leads to prohibitively long synchronization delay due to the large number of peers and unreliable overlay. Even “deadlock” may occur when a crashed replica node causes other replica nodes wait forever. Hence, the system scalability is restricted due to the low recovery from long synchronization delay with large number of nodes. At the other extreme, eventual consistency allows replica nodes to concurrently update their local copies. It only requires that all replica copies become identical after a long enough failure-free and update-free interval. Since replica nodes are highly unreliable in P2P systems, the node issuing update may have gone offline by the time update conflicts are detected, leading to unresolvable conflicts. It is infeasible to rely on a long duration without any failure or further updates. As a result, eventual consistency fails to provide any end-to-end performance guarantee to P2P users. Wide area data sharing applications vary widely in their frequency of reads and updates among replicas, in their tolerance of stale data and handling of update conflicts [11].

This paper presents a Balanced Consistency Maintenance (BCoM) protocol for structured P2P systems to balance the consistency strictness, availability and performance. Due consideration is given to dynamic workload, frequent replica node churns, heterogeneous resource capabilities, and different application consistency requirements. BCoM protocol serializes all updates to eliminate the complicated conflict handling in P2P systems, while allowing certain obsolescence in each replica node to improve the availability and performance. A sliding window update protocol is developed to regulate the number of allowable updates buffered by each replica node. This provides bounded consistency, the performance of which falls between the sequential and the eventual consistency.

Two main categories of bounded consistency are proposed for P2P systems: probabilistic consistency [3] [15] and time-bounded consistency [10] [12], both of which have limitations. (1) In the probabilistic consistency the probability is guaranteed with regard to all replica nodes but not for an individual node. BCoM ensures node level as well as system-wide consistency bound. (2) Time-bounded consistency sets the validation timer so that the estimated number of updates within the valid duration is small. To avoid the inaccuracy in this translation, BCoM uses the sliding window to directly bound the number of updates to be buffered at each node. (3) BCoM eliminates both redundant propagations in probabilistic bounded consistency and the individual computations of the timer in time-bounded consistency. Since redundancy is not needed for consistency probability and the window size does not depend on the latency at individual nodes, it is convenient to assign one node to adjust the window size.

An update window protocol has been designed for web-server systems [16] to bound the uncommitted updates in each replica node. But update conflicts and potential cascading impacts have not been addressed when optimizing the window...
size. Moreover, in P2P systems replica nodes are highly dynamic and unreliable, and the number of replicas in P2P systems is orders of magnitude larger than that in web-server systems. This makes any optimization model, which requires information on each node, impractical for P2P systems. BCoM analyzes the window size through a queuing model based on dynamic network condition, update workload and available resources. It periodically collects the system information to guide the window size setting with extremely low overhead.

In structured P2P systems, strong consistency is provided by organizing replica nodes to an auxiliary structure on top of the overlay for update propagation. Examples are the tree structure in SCOPE [2], two-tiered structure in OceanStore [8], and a hybrid of tree and two-tiered structure in [9]. SCOPE builds the tree by recursively partitioning the identifier space and selecting a representative node as the tree node for each partition. In BCoM, replica nodes of each object are also organized into a d-ary dissemination tree (dDT) on top of the overlay structure. The system-wide consistency bound is incrementally achieved by each internal tree node through applying the sliding window update protocol to its children. This makes the consistency scalable with the total number of replica nodes and the work of consistency maintenance is evenly distributed. Even though the root is responsible for serializing updates and accepting new joining nodes, we show that it will not become a bottleneck. The overhead of dDT solves “hot spot” and “single node failure” problems as efficiently as the previous identifier space partitioning methods in SCOPE [2] without burdening non-interested nodes. Moreover, dDT is designed to reduce the latency experienced by each replica node to receive an update from the root. Hence, dDT inserts the new node or re-join nodes to the smallest subtree and tries to balance the tree to shorten the overlay distance.

BCoM presents two enhancements to further improve the performance of a dDT. One is the ancestor cache scheme, where each node maintains a cache of ancestors for fast recovery from parent node failures. This also relieves tree-structure’s “multiplication of loss” problem [14] (i.e. all the subtree nodes rooted at the crashed node will lose the updates). The other is the node migration scheme, that migrates more capable nodes to upper layers and less capable nodes to lower layers to minimize the negative effect of the bottleneck node and maximize the overall performance. If an upper layer node is slow in propagating updates, the consistency constraint blocks ancestors from receiving new updates, and all its subtree nodes do not receive updates in a timely manner. Two forms of node migration are presented, one is to remove and the other is to prevent the blocking.

The contributions of our paper are the following:

- Propose a consistency maintenance framework for structured P2P systems to balance the consistency strictness, availability and performance through a sliding window update protocol with two enhancement schemes.
- Analyze the problem of setting the window size in response to dynamic network conditions, workload updates, and resource constraints through a queuing model to serve diverse consistency requirements from various mutable data sharing applications.
- Evaluate the performance of BCoM with comparison to SCOPE [2] using the P2PSim simulation tool. SCOPE is the most representative tree structure consistency model for P2P systems.

The rest of the paper is organized as follows: Sec.II introduces the three core techniques in BCoM, and Sec.III presents the performance evaluation. Finally, Sec.IV concludes the paper. The analytical model and detailed literature review are contained in the technical report [7].

II. DESCRIPTION OF BCoM

BCoM aims to: (1) provide bounded consistency for maintaining a large number of replicas of a mutable object; (2) balance the consistency strictness, availability and performance in response to dynamic network conditions, workload updates, and resource constraints; (3) make the consistency maintenance robust against frequent node churn and failures.

To fulfill these objectives, BCoM organizes all replica nodes of an object into a d-ary dissemination tree (dDT) on top of the P2P overlay for disseminating updates. It applies three core techniques: sliding window update, ancestor cache, and tree node migration on the dDT for consistency maintenance.

In this section, we first introduce the dDT structure, and then explain the three techniques in detail.

A. Dissemination Tree Structure

For each object BCoM builds a tree with node degree d rooted at the node whose ID is closest to the object ID in the overlay identifier space. We denote this d-ary dissemination tree of object i as dDTi, which consists of only the peers holding copies of object i. We name such a peer as a “replica node” of i, or simply as a replica node. An update can be issued by any replica node, but it should be submitted to the root. The root serializes the updates to eliminate the complicated handling of update conflicts because the update-issuing nodes may have gone offline.

The dynamic node behaviors require the construction of dDT to serve two cases (1) single node joining and (2) node with subtree rejoining. The goal of tree construction is to minimize the tree height under both cases, which lowers the update propagation latency and object discard rate for consistency maintenance.

We show an example of dDTi construction for case (1) with node degree d set to 2 in Fig.1. The replica nodes are ordered by their arrival time as node 0, node 1 and so on. At the beginning when node 1 and node 2 joined, both were assigned by node 0 (i.e. the root) as a child. Then, node 3 joined when node 0's degree was full, so it passed node 3 to its child who has the smallest number of subtree nodes denoted as Subno. Since both children (i.e. node 1 and node 2) had the same Subno., it randomly selected one to break the tie, say node 1, and updated the Subno.(1) accordingly. Subno. of a new node only accounts for itself. Node 1 assigned node 3 as its child, since it had a space for a new child. When node 4 joined, node 0 did not have space for a new child and passed
node 4 to the child with smallest Sub_{no.}, node 2. Similarly, node 5 and node 6 joined. The tree construction algorithm is given in Alg.1. For case (2) when node 6 crashed, all of its children detected the crash independently and contacted other ancestors to rejoin the tree, each acting as a delegate of its subtree to save individual rejoining of subtree nodes. Sub_{no.}, counts for all its subtree nodes and itself. Sec.II-C explains how to contact an ancestor for rejoining.

**Algorithm 1** dDT Construction (p, q)

Input: node p receives node q’s join request
Output: parent of node q in dDT

if p does not have d children then
    Sub_{no.}(p) = +Sub_{no.}(q)
    return p
else
    find a child f of p s.t. f has the smallest Sub_{no.}
    Sub_{no.}(f) = +Sub_{no.}(q)
    return dDT Construction (f, q)

**dDT** directs a new node and a rejoin node with its subtree to the child with the smallest subtree nodes when the parent node degree is full. The reason for not using the tree depth as the traditional balanced tree algorithm is that rejoining with subtree may increase the tree depth by more than 1, which goes beyond the one by one tree height increase handled by them. Another important reason is that maintaining the total number of nodes in each subtree is simpler and more time efficient than the depth of each subtree. Since the internal nodes need to wait until the insertion completes, the updated tree depth can be collected layer by layer from the leaves back to the root. This makes the real time maintenance of the tree depth quite difficult and unnecessary when tree nodes are frequently joining and leaving. However, the internal nodes can immediately update the total number of nodes in the subtree after forwarding the node to a child. The tree depth is periodically collected to help set the sliding window size as discussed in Sec.II-B2, where its result does not need to be updated in real time. But using an outdated tree depth for insertions to dDT will lead to unbalanced tree and degrade the performance.

**B. Sliding Window Update Protocol**

1) **Basic Operation in Sliding Window Update:**
Sliding window regulates the consistency bound for update propagations to all replica nodes in a dDT. “Sliding” refers to the incremental adjustment of window size in response to dynamic system conditions. If dDT of object i is assigned a sliding window size $k_i$, any replica node in dDT can buffer up to $k_i$ unacknowledged updates before being blocked from receiving new updates. At the beginning, the root receives the first update, sends to all children and waits for their ACKs. There are two types of ACKs, R_ACK and NR_ACK, both indicating the successful receiving of the update. R_ACK indicates that the sender is ready to receive the next update; NR_ACK means the sender is not ready. While waiting, the root accepts and buffers the incoming updates as long as its $k_i$ size buffer does not overflow. When receiving an R_ACK from a child, the root sends the next update to this child if there is a buffered update that has not been sent to this child. When receiving an NR_ACK from a child, it will not send the next update, but the update is marked to be received by this child.

After receiving ACKs from all children, the update is removed from its buffer. There are two cases of buffer overflow: 1) when the root’s buffer is full, the new updates are discarded until there is a space; 2) when an internal node’s buffer is full, the node sends NR_ACK to its parent for the last received update. An R_ACK is sent to its parent when there is space in the buffer to resume receiving updates. A leaf node does not maintain such update buffer. After receiving an update, it immediately sends R_ACK to its parent. Fig.2 shows an example of sliding window update protocol with window size set to 8. V stands for the version number of an update, as $V_{10} - V_{13}$ means that the node keeps the updates from the 10th version to the 13th version. Each internal node keeps the next version for its slowest child up to the latest version it received, and each leaf node only keeps the latest version it received.

2) **Setting of Sliding Window Size:**
The sliding window size $k_i$ is critical in balancing the consistency strictness, the object availability and the update dissemination performance. The value of $k_i$ is an indicator of consistency strictness. A larger $k_i$ helps mask the long network latency and temporary unavailability of the replica nodes, lowers the update discard rate and improves the availability. The disadvantages of a larger $k_i$ are (1) discrepancy between the replica local view and the most updated view at the root giving rise to weaker consistency; and (2) longer queueing delay in
update propagation, thus lowering the update dissemination performance. On the extremes, infinite buffer size provides eventual consistency without discarding updates, and size zero provides sequential consistency with worst update discard rate. Please refer to [7] for the window size update.

C. Ancestor Cache Maintenance

Each replica node maintains a cache of $m$ ancestors starting from its parent leading to the root in the $dDT$. The value of $m$ is set based on the node churn rate (i.e. the number of nodes leaving the system during a given period) so that the possibility of all $m$ nodes simultaneously failing is unlikely. When the node does not have $m$ ancestors, it caches information for all the nodes beginning from the root.

A node contacts its cached ancestors sequentially layer by layer upwards when its parent becomes unreachable. This can be detected by ACK and maintenance message transmissions. The sequential contact operation will find the closest ancestor, no matter how many layers of node crashes exist. The root is finally contacted for relocation if all the other ancestors crash. We assume the root is reliable, since the overlay routing will be detected by ACK and maintenance message transmissions. The sequential contact operation will find the closest ancestor, no matter how many layers of node crashes exist. The root is finally contacted for relocation if all the other ancestors crash.

The contacted ancestor runs the tree construction Alg.1 to find a new position for this rejoining node with its subtree. BCoM does not replace the crashed node by a leaf node to maintain the original tree structure, since migration brings the bottleneck node down to the leaf layer for performance improvement. The new parent transfers the latest version of the object to this new child position if necessary. Since each node only keeps $k_i$ previous updates, content transmission is used to avoid the communication overhead for getting the missing updates from other nodes. The sliding window update propagation resumes for incoming updates.

The ancestor cache provides fast recovery from node and link failures with a small overhead and high success probability. Assuming the probability of a replica node failure as $p$, the ancestor cache with size $m$ has a successful recovery probability of $1 - p^m$. An ancestor cache is easily maintained by piggybacking an ancestor list to each update. Whenever a node receives this update it adds itself to the ancestor list before propagating the update to the children. Each node refers to the newly received ancestor list to refresh its cache. There is no extra communication for the piggyback, and the storage overhead is also negligible for keeping the information of $m$ ancestors.

D. Tree Node Migration

Any internal node with the subtree rooted at it will be blocked from receiving new updates if one of its children is too slow due to the sliding window constraint. It is quite possible that a lower layer node performs faster than the bottleneck node, so we should promote the faster node to a higher level and degrade the bottleneck node to a lower level. For example in Fig.1, assume node 1 is the bottleneck getting the root 0 blocked. The faster node may be a descendant of the bottleneck node (A) or a descendant of a sibling of the bottleneck node (B). When blocking occurs, node 0 can swap the bottleneck node 1 with a faster descendant with more recent updates, like node 4, to remove the blocking. Before blocking occurs, node 1 can be swapped with its fastest child with the same update version to prevent the blocking. The performance improvement through node migration is confirmed by our queuing model of $dDT$. There are two forms of node migration, as described below.

- Blocking triggered migration: the blocked node searches for a faster descendant, which has a more recent update than the bottleneck node and swaps them to remove the blocking.
- Non-blocking migration: when a node observes a child performing faster than itself, it swaps with this child. This migration prevents the potential blocking and speeds up the update propagation for the subtree rooted at the parent.

The swapping of (B) in Fig.1 is an example of blocking triggered migration and (A) is an example of non-blocking migration. Both forms of migration swap one layer at a time and, hence, multiple times of migrations are needed for multi-layer swapping. The non-blocking migration helps promote the faster nodes to upper layers, which makes the searching in blocking-triggered migration easier. Please refer to the technical report for the basic operations in BCoM.

III. PERFORMANCE EVALUATION

In this section, we evaluate the efficiency of BCoM with comparison to SCOPE [2], which is a seminal work of consistency maintenance in structured P2P networks. We extend the P2PSim tool [1] to simulate the heterogeneous node capacities and transmission latency. While BCoM can be applied to every type of structured P2P system, we choose Tapestry [17] as a representative network for simulations.

Simulation Setting: We simulate a network of 1000 nodes because anything larger cannot be executed stably in P2PSim. The number of objects ranges from $10^2$ to $10^3$. The object popularity follows a Zipf’s distribution, and the update arrivals are generated by a Poisson process with different average arrival rates. Network topology is simulated by two transit-stub topologies generated by GT_JTM [4] to model dense and sparse networks. The update discard rate (the ratio of the number of discarded updates to the total number of arriving updates), and update dissemination latency (the average delay for each node to receive the update) are used to measure the protocol efficiency. For detailed setting please refer to [7].

Efficiency of the Window Size: This simulation explores the efficiency of applying sliding window protocol. The curves in Fig.3 and Fig.4 show that by increasing the window size from 1 to 20, the discard rate is dropped from 80% to around 5% and the latency is increased only by 20%, which confirms that BCoM significantly improves the availability with slight sacrifice of latency performance compared to the sequential consistency.

Scalability of BCoM: This simulation verifies the scalability of BCoM with comparison to SCOPE by varying the
number of replica nodes and the update rate of each object. The results in Fig.5 and Fig.7 show that the discard rate of BCoM is maintained to less than 10% as the number of replicas per object increases from 10 to 1000 and the number of updates issued per node is increased from 1 to 200. On the other hand, applying applying the sequential consistency makes the discard rate of SCOPE almost 100%, except with a very small number of replica nodes (i.e. 10 nodes per object) or with an extremely low update rate (i.e. 1 node per object). The sliding window protocol and the adaptive window size setting contribute to good availability maintenance under dynamic system conditions.

As shown in Fig.6 the latency of BCoM is slightly higher than that in SCOPE when the number of replica nodes is large. This is due to the accumulated queuing delay at each internal node introduced by sliding window. But the increase is controlled within 1/3rd of the latency of SCOPE, which matches with the latency increase bound in window size setting for improved discard rate. The results of Fig.8 show that the latency of BCoM is similar to that of SCOPE when update rates are low, and longer than SCOPE when update rates are high. The reason is that under frequent updates, a new join or re-join node needs to have larger content transfer to get the latest version, which prolongs the average latency in BCoM. However, we do not apply this requirement in SCOPE to upgrade its discard rate to be comparable with that of BCoM. As a result, their latency results are also tuned better. In summary, BCoM achieves much higher availability than SCOPE at the cost of controlled latency increase for bounded consistency in large scale P2P systems with frequent updates. Such good balance confirms the objectives in the analytical model of the window size setting. More evaluations are given in [7] on the overhead and fault tolerance of BCoM.

IV. CONCLUSION

This paper presents a balanced consistency maintenance (BCoM) framework for improving availability, performance and consistency strictness in structured P2P systems. The simulation results from P2PSim demonstrate that BCoM out-performs SCOPE by greatly improving discard rate from almost 100% to 5% with a small sacrifice of latency under bounded consistency constraint.

REFERENCES