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 how to efficiently maintain 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 queueing 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 [24]. 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

Structured P2P systems have been effectively designed for wide area data applications [21] [10] [16] [18] [22] [20]. While most of them are designed for read-only or low-write sharing contents, a lot of promising P2P applications demand for supporting mutable contents, such as modifiable storage systems (e.g. OceanStore [16], Publius [19]), mutable content sharing (e.g. P2P WiKi [13]), even interactive ones (e.g. P2P online games [2] [5] and P2P collaborative workspace [12]). P2P organization improves availability, fault tolerance, and scalability for static content sharing. But mutable content sharing raises issues of replication and consistency management. P2P dynamic network characteristics combined with the diverse application consistency requirements and heterogeneous peer resource constraints also impose unique challenges for P2P consistency management. This requires a consistency solution to work efficiently in such dynamic conditions.

P2P systems are typically large scale, where peers with various resource capabilities experience diverse network latency. Also, their dynamic joining and leaving make the P2P overlay failure prone. Neither sequential consistency [15] nor eventual consistency [8] individually works well in P2P environment. It has been proved [14] 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 makes other replica nodes wait forever. Hence, the system scalability is restricted due to the lowered availability from long synchronization delay for a large number of nodes. At the other extreme, eventual consistency allows replica nodes concurrently update their local copies and only requires that all replica copies become identical after a long enough failure-free and update-free interval. Since in P2P systems replica nodes are highly unreliable, the update-issuing node 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, due to which eventual consistency fails to provide any end-to-end performance guarantee to P2P users. As surveyed in [23], 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.

This paper presents a Balanced Consistency Maintenance (BoM) protocol for in structured P2P systems for balancing 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 obsoleteness in each replica node to improve the availability and performance. A sliding window update protocol is used to specify the number of allowable updates buffered by each replica nodes. 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 [4] [30] and time-bounded consistency [25] [26], both of which have main limitations, but are relaxed with BCoM. (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 timer valid duration is small. To avoid the inaccuracy in this translation, BCoM uses the sliding window to directly bound the number of updates allowed 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 set and adjust the window size.

An update window protocol has been designed for web-server systems [31] to bound the uncommitted updates in each replica node. But update conflicts and potential cascading impacts can hardly been addressed when optimizing the window size. Moreover, there are two challenges for applying this technique to P2P systems: (1) unlike the web-servers, P2P replica nodes are highly dynamic and unreliable; (2) the number of replicas in P2P systems is orders of magnitude larger than that in web-server systems. (1) and (2) together make any optimization model impractical for P2P systems because it requires information on each node's update rate, propagating latency, etc. BCoM analyzes the window size through a queueing model based on dynamic network condition, update workload and available resources. It periodically collects the general system information, such as the total layers of replica node and the bottleneck latency, and guides the window size setting with extremely low overhead. In this way, the consistency maintenance and performance optimization in BCoM scale well with the P2P systems and adapt promptly to the dynamic conditions.

In BCoM, replica nodes of each object are 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. Since each replica node takes charge of its children in update propagation and consistency maintenance, the work of consistency maintenance is evenly distributed. Even though the root is responsible for serializing updates and accepting new joining node, we show that it will not become a bottleneck.The overhead of dDT is lightweight and evenly distributed to prevent "hot spot" and "single node failure" problems as efficiently as the previous identifier space partitioning methods in [24] [29]. Another primary goal of constructing a dDT is to reduce the latency experienced by each replica node to receive an update from the root. Thus dDT inserts the new join 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 [11] (i.e. all the subtree nodes rooted at the crashed node will lose the updates), which is especially critical in P2P systems. Maintaining the ancestor cache does not introduce extra overhead since the needed information conveniently piggybacks on update propagation. A small size of cache can also significantly improve the robustness against node failures. The other is the *node migration* scheme, that is to migrate more capable nodes to upper layers and less capable nodes to lower layers to minimize the side 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 the blocking and the other is to prevent the blocking so that unnecessary performance and availability degradations are removed.

The contributions of our paper are the following:

- Propose a consistency maintenance framework in structured P2P systems for balancing the consistency strictness, availability and performance through a sliding window update protocol with two enhancement schemes.
- Analyze the problem of optimizing the window size in response to dynamic network conditions, update workload, and resource constraints through a queueing model to serve diverse consistency requirements from various mutable data sharing applications.
- Evaluate the performance of BCoM with comparison to SCOPE using the P2PSim simulation tool.

The rest of the paper is organized as follows: Sec.II introduces the three core techniques in BCoM and the protocol deployment. Sec.III presents the analytical model for window size setting. The performance evaluation is given in Sec.IV and the existing literature is reviewed in Sec.V. The paper is concluded in Sec.VI.

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, update workload, and resource constraints; (3) make the consistency maintenance robust against frequently node churns 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 dDT_i, 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 behavior requires 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 dDT_i construction for case (1) with node degree d set to 2 in Fig.1. The replica nodes are ordered by their joining 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 as $Sub_{no.}$. Since both children (i.e. node 1 and node 2) had the same $Sub_{no.}$, it randomly selected one to break the tie, say node 1, and updated the $Sub_{no.}(1)$ accordingly. $Sub_{no.}$ of a join node is one standing 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 ancestor 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.

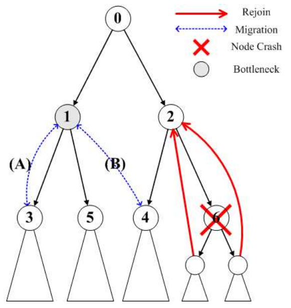

Fig. 1. Dissemination Tree Example

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 join node and a rejoin node with its subtree to the child node with the smallest subtree nodes when the parent node degree is full. The reason for not using the tree depth as the traditional tree balanced algorithm is that rejoining with subtree may increase the tree depth by more than 1, which is 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 joining 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 dDT construction 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 condition. If dDT_i of object i is assigned a sliding window size k_i, any replica node in dDT_i can buffer up to k_i unacknowledged updates before being blocked from receiving new updates. At the beginning, 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. 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 window size set to 8, V stands for the version number of the update, as $V10 - V13$ means the node keeps the updates from 10th version to 13th version. Each internal node keeps the next version for its slowest child until 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 plays a critical role in balancing the consistency strictness, the object availability and the update

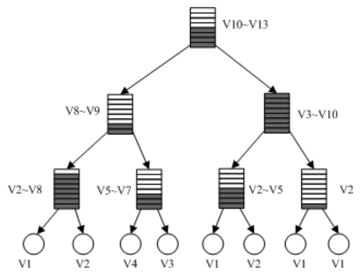

Fig. 2. An example of sliding window update protocol

dissemination performance. The value of k_i is an indicator of consistency strictness. The larger k_i helps mask the long network latency and temporary unavailability of the replica nodes, lowers the update discards 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 buffer size zero provides sequential consistency with worst update discards.

We explain here how the root updates the window size with the analytical model in Sec.III giving the specific formula to guide the update. The root measures input metrics every T seconds and adjusts the k_i value only when the metrics stable and the old k_i violates the constraint in Eq.7. In this way, the unnecessary changes due to the temporary disturbances are eliminated to keep the dDT_i stable. In case k_i needs to be adjusted, it is incrementally increased or decreased one by one until the constraints are satisfied.

The computation of k_i requires the information on the update arrival rate λ, the tree height L, and the bottleneck service time μ_L. The arrival rate is directly measured by the root. The tree height and bottleneck service time are collected periodically from leaf nodes to the root in a bottom-up fashion. The two metrics are aggregated at every internal node, so that the maintenance message always keeps the same size. The aggregation is performed as follows: each leaf node initializes the tree height to zero ($L = 0$) and the bottleneck service time μ_L to its update propagation time. Each node sends the maintenance message to its parent. Once an internal node receives the maintenance messages from all children, it updates L as the maximum value of its children's tree height plus 1 and μ_L as the maximum value among its and every child's service time. If its service time is longer than a child's, a non-blocking migration is executed to swap the parent with the child. This aggregation continues until the root is reached.

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 automatically handle the root failure by letting the node with the nearest ID to replace the crashed root of dDT.

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 $1 - p^m$. It is very unlikely that all of the m cached ancestors fail simultaneously; even if it occurs, the root can be contacted for the relocation. 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 slowest child is blocked 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 in Fig.3. 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 (A) in Fig.1 is an example of blocking triggered migration and (B) 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. Since the overlay DHT routing in structured P2P networks relies on cooperative nodes, we assume BCoM is run by these cooperative P2P nodes transparent to the end users. Tree node migration uses only the local information and improves the overall system performance.

E. Basic Operations in BCoM

BCoM provides three basic operations:

- Subscribe: when a node p wants to read the object i and keep it updated, p sends the subscription request to the root of dDT_i by overlay routing. After receiving the request, the root runs Alg.1 to locate a parent for p in dDT_i, who will transfer its most updated version to p. The subsequent updates are received under sliding window protocol. The message overhead for a subscription is $O(\log_d N)$, since locating a new node at most searches along a path from the root to a leaf in dDT_i.
- Unsubscribe: when a node p does not want object i anymore, it promotes its fastest child as the new parent and transfers its parent and other children's information to the newly promoted node. p also notifies them of the newly promoted node to update their related maintenance information. The message overhead for a node leaving is $O(1)$, since the number of the affected node is no more than d, and each has constant overhead to update the related maintenance information.
- Update: after subscribing, if a node p wants to update the object, it sends the update request directly to the root using IP routing. The root's IP address is obtained through the subscription or the ancestor cache. If the root crashes, p submits the update to the new root through overlay routing. Updates are serialized at the root by their arrival time. The specific policy for resolving conflicts is application dependent. The message overhead of an update is $O(1)$ for the direct submission to the root.

III. Analytical Model for Sliding Window Setting

The unstableness of P2P systems forbids us to use any complicated optimization techniques that require several hours of computation at workstations (e.g. [28]) or every node information in the entire system (e.g. [31]). BCoM adjusts the sliding window size timely to dyanmic P2P systems relying on limited information.

This section presents the analytical model of the sliding window size k_i of object i, where the update propagation to all replica nodes is modeled by a queuing system. We first analyze the queueing behavior when an update is discarded, then calculate the update discard probability and the expected latency for a replica node to receive an update, finally, we set k_i to balance the availability and latency constrained by consistency bounds.

A. Queueing Model

Assuming the total number of replica nodes as N, the node degree as d, and there are L ($L = O(\log_d N)$) layers of internal nodes with update buffer size k_i (i.e. layer $0 \ldots L-1$ nodes with sliding window k_i). The leaf nodes are in layer-$_L$ and do not need buffer. The update arrivals are modeled by a Poisson process with average arrival rate λ_i (simply as λ), as each update is issued by a replica node independently and identically at random. The latency of receiving an update from the parent and acknowledged by the child is denoted as the service time for update propagation. The service time for one layer to its adjacent layer below is the longest parent-child service time in these two layers. μ_l denotes the service time for update propagation from layer-$_l$ to layer-$_{l+1}$. For examples, μ_0 is the service time from the root to its slowest child, μ_{L-1} is the longest service time from a layer-$_{L-1}$ node to its child (i.e. a leaf node). The update propagation delay is assumed to be exponential distributed. The update propagations in dDT_i are modeled as a queuing process shown in Fig.3 (a): The updates arrive with average rate λ at the root, then go to the layer-$_0$ buffer with size k_i. The service time for propagating from layer-$_0$ to layer-$_1$ is μ_0. After that, the updates go to layer-$_1$ nodes' buffer of size k_i with service time as μ_1 for propagating to layer-$_2$ nodes. The propagations end when updates are received by the leaves in the layer-$_L$.

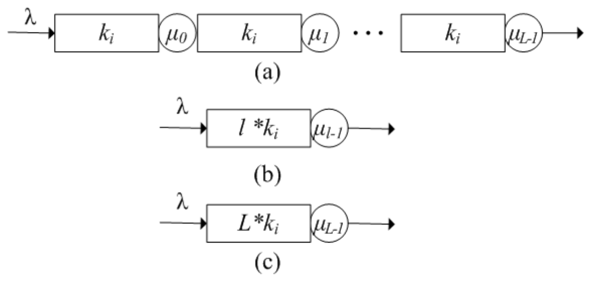

Fig. 3. Queuing Model of Update Propagation

An update may only be discarded by the root when its buffer overflows. This happens when the root is waiting for R_ACK from the slowest child in layer-$_1$, who is waiting for R_ACK from its slowest child in layer-$_2$. The waiting cascades until the bottleneck node of the dDT_i is reached, say in the layer-$_l$, $0 \leq l \leq L$. The nodes in layers $l+1 \ldots L$ (if $l < L$) do not receive any update even when their buffers are not full.

All the nodes in the path from the root to the bottleneck node have buffer overflow. The nodes along the path are denoted by $p_0, p_1 \dots p_l$, where p_0 is the root and p_l is the bottleneck node. After the bottleneck node p_l receives an update, it sends an R_ACK to its parent. The R_ACK is then propagated to the root p_0, such that the root can purge the update from its buffer and accept a new one. The update propagation from $p_0 \to p_1, p_1 \to p_2, \dots p_{l-1} \to p_l$ is in parallel and the service time μ_{l-1} between p_l and p_{l-1} should be the longest along this path (i.e. $\mu_{l-1} > \mu_j, 0 \le j < l-1$). Therefore, the queuing model of update discarding is transformed to a queue with the effective buffer size $l * k_i$ for the dDT_i, and the service time is $\mu_(l-1)$, as shown in Fig.3 (b).

This queuing model of the update discard explains that given a k_i, the effective buffer size $l * k_i$ is determined by l, which is the layer the bottleneck node resides. The larger the effective buffer size, the lower the discard probability. So when the bottleneck node is a leaf ($l = L$), buffer resources of dDT_i is maximal used with effective buffer size $L * k_i$. This inspires the Tree Node Migration techniques presented in Sec.II-D, which help to move down the bottleneck node to the leaf layer to boost the overall performance of the dDT_i. The discard probability of an update is computed based on the queuing model of dDT_i after optimized by tree node migrations as in Fig.3 (c). The queue becomes a $M/M/1/$ queue with buffer size $L * k_i$, arrival rate λ and service time μ_{L-1}.

B. Availability and Latency Computation

Define the update request intensity as ρ.

$$\rho = \frac{\lambda}{\mu_{L-1}} \tag{1}$$

Define the probability of n updates in the queue as π_n. Based on the queueing theory for $M/M/1$ finite queue [6], π_n is represented as Eq.2.

$$\pi_n = \rho^n \pi_0 \tag{2}$$

The discard probability is π_{L*k_i}, which indicates the buffer overflow. From $\Sigma_{n=0}^{L*k_i} \pi_n = 1$, we get $\pi_0 = \frac{1-\rho}{1-\rho^{L*k_i}}$. And the discard probability is computed in Eq.3.

$$\pi_{L*k_i} = \frac{1-\rho}{1-\rho^{L*k_i}} \rho^{L*k_i} \tag{3}$$

The expected number of packets in the queue $E[N_{L*k_i}]$ is calculated in Eq.4.

$$E[N_{L*k_i}] = \sum_{0 \le n \le L*k_i} n * \pi_n \tag{4}$$

Plug in the Eq.2 for π_n, the final form of $E[N_{L*k_i}]$ is given in Eq.5.

$$E[N_{L*k_i}] = \frac{(L*k_i+1)\rho^{L*k_i+1}}{(\rho^{L*k_i+1}-1)} + \frac{\rho}{(1-\rho)} \tag{5}$$

The expected delay $E[T_{L*k_i}]$ is calculated by Little's law in Eq.6, where $E[N_{L*k_i}]$ is the expected number of packets in the queue and $\lambda(1-\pi_{L*k_i})$ is the arrival rate of the accepted updates.

$$E[T_{L*k_i}] = \frac{E[N_{L*k_i}]}{\lambda(1-\pi_{L*k_i})} \tag{6}$$

C. Window Size Setting

The effectiveness of a consistency protocol is measured by three attributes: consistency strictness, object availability and latency for receiving an update, and the three are in subtle tension towards each other. Given the update arrival rate and the service time, increasing the window size k_i lowers the discard probability, while prolongs the expected latency and weakens the consistency strictness. It is hard to accurately model the delay for an update to be received by each replica node, since besides the queueing delay at each node, the dynamic node joining and leaving cause disturbance on the update propagation process. The expected latency $E[T_{L*k_i}]$ indicates the average delay for an update to arrive at a replica node, which serves as a simple and approximate indicator. The consistency strictness is measured by the number of updates a replica node has not yet received, which is at most $L * k_i$ in dDT_i.

BCoM sets the window size to balance among the three attributes by maximizing the object availability under the constraints that the number of not-yet-received updates is bounded to K_m and latency for receiving the update is no worse than the sequential consistency for a small bound T_s in Eq.7. $E[T_{L*k}]$ is the expected latency with a window size k and $E[T_L]$ is the expected latency for sequential consistency for this dDT_i, which serves as the baseline for bounding the latency performance. The latency threshold T_s and the consistency strictness threshold K_m are set according to application requirements. In our simulation, empirically setting T_s to 1.3 achieves good results shown in Fig.6 and Fig.7, the discard probability is improved from almost 100% to 5% at the cost of latency increases less than one third most of the time. K_m is set to 60 based on the network size of 1000.

$$k_i = arg \min \ \pi_{L*k} \ s.t. \ \frac{E[T_{L*k}]}{E[T_L]} \le T_s, L*k \le K_m \tag{7}$$

IV. Performance Evaluation

In this section, we evaluate the efficiency of BCoM with comparison to SCOPE [24], 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 [7] as a representative network for simulations.

A. 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 10^4. The object popularity follows a Zipf's distribution, and the update arrivals are generated by a Poisson process with different average arrival rates. By default each node issues 200 updates during a simulation cycle, which is $7.2 * 10^6$ time slots. This setup is aimed at simulating the situation where frequent updates may overload the servers, which motivates the use of P2P systems. Given that transmitting one update uses only 10 to 100 slots, the number of time slots covered in a simulation cycle (i.e. $7.2 * 10^6$) is large enough to generate sustainable results. The data points in our figures are the average values of 20 trials.

The heterogeneity of node capacities follows a Pareto distribution [29]. We set the shape parameter $a = 1$ and scale parameter $b = 900$ to get 900 different node capacities. Network topology is simulated by two transit-stub topologies generated by GT_ITM [9] to model dense and sparse networks: (1) ts1k-small (dense) - 2 transit domains each with 4 transit nodes, 4 stub domains attached to each transit node, and 31 nodes in each stub domain. (2) ts1k-large (sparse) - 30 transit domains each with 4 transit nodes, 4 stub domains attached to each transit node, and 2 nodes in each stub domain.

The node degree is set to 5, since the average Gnutella node degree is 3 to 5. To have a fair comparison, we also set the vector degree of each SCOPE node to 5. 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.

B. Efficiency of the Window Size

This simulation explores the efficiency of applying sliding window protocol. The curves in Fig.4 and Fig.5 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.

C. 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.6 and Fig.8 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.7 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 $1/3$rd of the latency of SCOPE, which matches with the latency increase bound in window size setting for improved discard rate. The results of Fig.9 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.

D. The Overhead of BCoM

This simulation compares the overhead of BCoM with that of SCOPE as shown in Fig.10. The consistency maintenance overhead of each object consists of three parts: subscribe overhead, update overhead, and crash/migrate overhead, which account for subscribing, updating the object and recovering from crashes, respectively. We use the label ”migrate” to indicate the migration and crash recovery overhead in BCoM. BCoM keeps the overhead at the same level as that in SCOPE. The reason is that the ancestor cache maintenance and the node migration mostly piggyback on update disseminations for sliding window optimization to reduce overhead.

E. Fault Tolerance of BCoM

This simulation examines BCoM's robustness against node failures by varying the node mean life time. Life time is the ratio of the average number of slots a node stays online at one time to the total number of slots in a simulation cycle. The smaller the life time is, the more frequently the nodes join and leave. The results of SCOPE are not presented because their discard rate is nearly 100% when the nodes are joining or leaving. The results of Fig.11, Fig.12, and Fig.13 show that BCoM keeps the tree depth, the discard rate and the latency in good status for different frequencies of node joining and leaving. The ancestor cache helps maintain the tree structure under node churns. And adaptive window size setting keeps the availability and latency performance stable.

V. Related Work

A. Consistency Maintenance in P2P systems

In structured P2P systems, strong consistency is provided by organizing replica nodes to an auxiliary structure on top of the overlay for update propagation, like the tree structure in SCOPE [24], two-tired structure in OceanStore [16], and a hybrid of tree and two-tired structure in [29]. The tree constructions in [24] [29] follow the node ID partitioning, instead, dDT inserts the new node to the smallest subtree to make it balanced under dynamic node joining and leaving. dDT achieves the same load balance and failure robustness as them with greatly enhanced availability for consistency maintenance.

In unstructured P2P systems, mainly two types of bounded consistency are provided: 1) probabilistic bounded consistency: rumor spreading [4] and replica chain [30] are used to ensure a certain probability of an update being received; 2) time-bounded consistency: TTL guided push and/or pull methods are used in [25] [26] to indicate the valid period for a replica copy, when the period expires the replica node

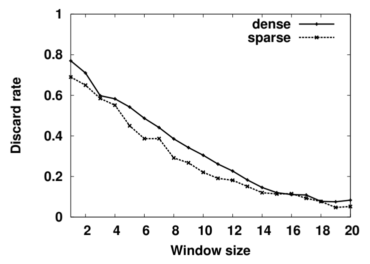

Fig. 4. The impact of window size on discard rate

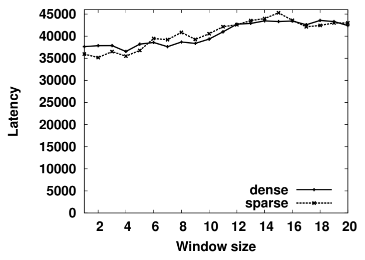

Fig. 5. The impact of window size on latency

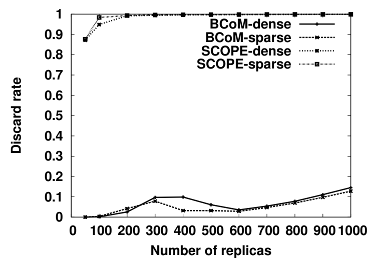

Fig. 6. The impact of replica number on discard rate

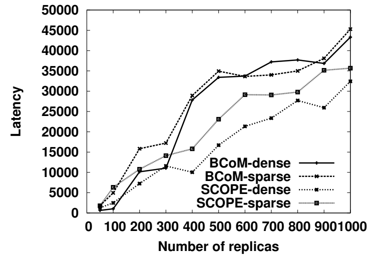

Fig. 7. The impact of replica number on latency

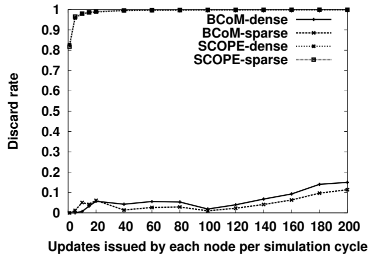

Fig. 8. The impact of update pattern on discard rate

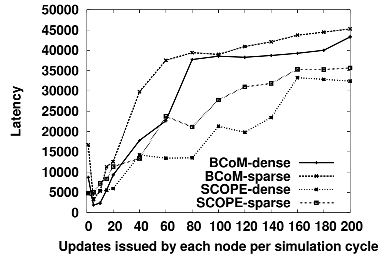

Fig. 9. The impact of update pattern on latency

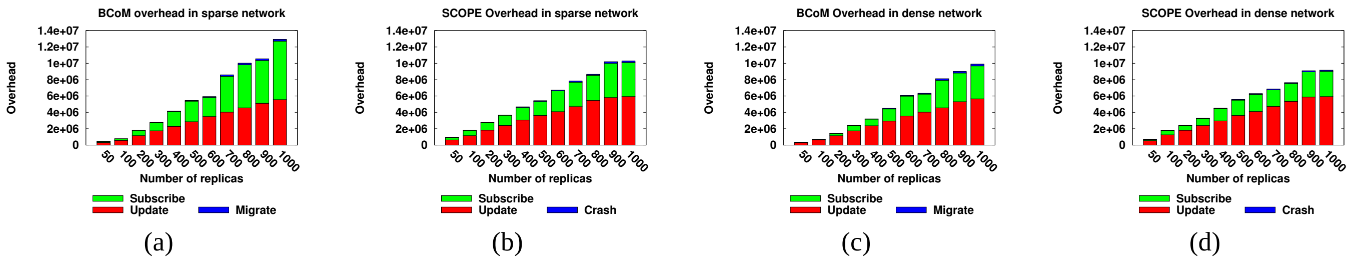

Fig. 10. Overhead comparison between BCoM and SCOPE

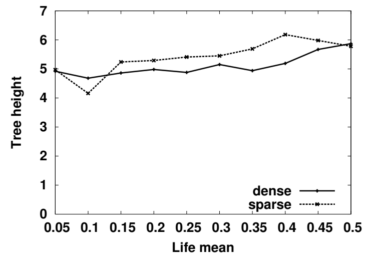

Fig. 11. The impact of churn rate on tree height

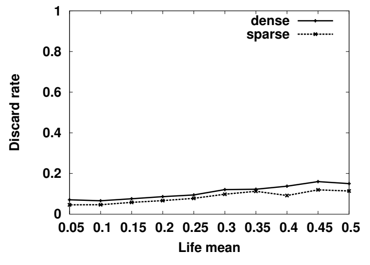

Fig. 12. The impact of churn rate on discard rate

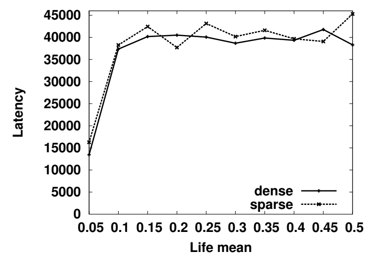

Fig. 13. The impact of churn rate on latency

needs to check validity with the source to serve the following read requests. Node-level consistency is not ensured by probabilistic bounded consistency, and the ambiguity introduced by translating the valid time duration to the number of inconsistency updates in time-bounded consistency, are avoided by our sliding window update protocol.

B. *Overlay Content Distribution*

Update delivery in P2P overlay has four requirements: (1) a bounded delay for update delivery, (2) robustness to frequent node churns and update workload changes, (3) awareness of heterogeneous peer capacities, and (4) scalability with a large number of peers. The LagOver [3] constructed an update delivery tree by jointly considering each user's capacity and latency requirements to address (1) and (3), both of which are also handled by tree node migration in BCoM. The major difference is that LagOver improves the performance to meet the individual replica node's requirement, while node migration improves performance system-wide. And LagOver requires information on each user's latency requirement and capacity, which are infeasible to be implemented in P2P systems. But node migration exploits local information and is performed on demand to support (2) without asking a replica node to specify requirements in advance.

The "side link" is used in content dissemination tree in [11] to address (2), where each node keeps multiple side links from other subtrees to minimize the impact of loss multiplication in a tree structure. The two end nodes of a side link do not share any ancestor except the root. Our ancestor cache achieves the same goal by only caching ancestors and contacting the ancestor one layer above the failed nodes. The ancestor cache in BCoM has extra benefit by avoiding communication overhead to maintain end nodes on the other subtree. All the ancestors' information can be piggyback on the update propagation. Besides, in BCoM a node sequentially contacts the cached ancestors to avoid conflict relocation decisions while in [11] a node uses multiple side links in parallel to retrieve the lost packets, serving different aims.

C. Tunable Consistency Models

Previous works have explored continuous models for consistency maintenance [17] [27], which have been extended by a composable consistency model [23] for P2P applications. Hybrid push and pull methods are also used to provide application tailored cache consistency [32] [25]. An analytical model for adaptive update window protocol is provided in [31], where the window specifies the number of uncommitted updates in each replica node's buffer and the information of each node's update rate and propagation latency are required to optimize the window size. While in BCoM updates are serialized to eliminate the update conflicts and potential cascading effects. The window specifies the number of committed updates each replica node can buffer, no information on each node is required for the window size setting in BCoM.

VI. Conclusion

This paper presents a balanced consistency maintenance (BCoM) framework for improving availability, performance and consistency strictness in structured P2P systems. A sliding window update protocol is applied with two enhancement schemes, and the window size setting is analyzed through a queueing model. This gives a good response to observations on dynamic network conditions, update workload and peers' resource limits, such that various application consistency requirements are smoothly served. The simulation results from P2PSim demonstrate that BCoM outperforms SCOPE by greatly improving discard rate from almost 100% to 5% with a small sacrifice of latency under bounded consistency constraint.

References

[1] P2PSim. http://pdos.csail.mit.edu/p2psim/.

[2] A. Bharambe, J. R. Douceur, J. R. L. T. M. J. P. S. S., and Zhuang, X. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *ACM SIGCOMM* (2008).

[3] A. Datta, I. S., and Franklin, M. Lagover: latency gradated overlays. In *IEEE ICDCS* (2007).

[4] A. Datta, M. H., and Aberer, K. Updates in highly unreliable, replicated peer-to-peer systems. In *IEEE ICDCS* (2003).

[5] Agarwal, S., and Lorch, J. R. Matchmaking for online games and other latency-sensitive p2p systems. In *ACM SIGCOMM* (2009).

[6] Bertsekas, D. P., and Gallager, R. G. *Data Networks*. Englewood Cliffs, NJ: Prentice-Hall, 1986.

[7] B.Y. Zhao, L. Huang, J. S. S. R. A. J., and Kubiatowicz, J. Tapestry: a resilient global-scale overlay for service deployement. *IEEE J-SAC 22*, 1 (2004), 41–53.

[8] D. B. Terry, M. M. Theimer, K. P. A. J. D. M. J. S., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *ACM SOSP* (1995).

[9] E.W. Zegura, K. C., and Bhattacharjee, S. How to model an internetwork. In *IEEE INFOCOM* (1996).

[10] F. Dabek, M. F. Kaashoek, D. K. R. M., and Stoica, I. Wide area cooperative storage with cfs. In *USENIX Security Symp.* (2000).

[11] F. Wang, J. L., and Xiong, Y. Stable peers: existence, importance, and application in peer-to-peer live video streaming. In *ACM MobiCom* (2004).

[12] G. Oster, P. Urso, P. M., and Imine, A. Data consistency for p2p collaborative editing. In *CSCW* (2006).

[13] G. Urdaneta, G. P., and Steen, M. V. A decentralized wiki engine for collaborative wikipedia hosting. In *WEBIST* (2007).

[14] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. In *ACM PODC* (2002).

[15] Herlihy, M. P., and Wing, J. M. Linearlizability: A correctness condition for concurrent objects. *ACM Trans Programm. Lang. Syst. 12*, 3 (1990), 463–492.

[16] J. Kubiatowicz, D. Bindel, Y. C. S. C. P. E., and Geels, D. Oceanstore: an architecture for global-scale persistent storage. In *ACM ASPLOS-IX* (2000).

[17] Krishnakumar, N., and Bernstein, A. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM TODC 19*, 4 (1994).

[18] M. Castro, P. Druschel, A. K., and Rowstron, A. Scribe: A large scale and decentralized application level multicast infrastructure. *IEEE J-SAC 20*, 8 (2002), 1489–1499.

[19] M. Waldman, A. D. R., and Cranor, L. F. Publius: A robust, tamper-evident, censorship-resistant web-publishing systems. In *USENIX Security Symp.* (2000).

[20] Ramasubramanian, V., and Sirer, E. G. Beehive: exploiting power law query distribution for o(1) lookup performance in peer-to-peer overlays. In *NSDI* (2004).

[21] Rowstron, A., and Druschel, P. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. In *ACM SOSP* (2001).

[22] S. Iyer, A. R., and Druschel, P. Squirrel: a decentralized peer-to-peer web cache. In *ACM PODC* (2002).

[23] Susarla, S., and Carter, J. Flexible consistency for wide area peer replication. In *IEEE ICDCS* (2005).

[24] X. Chen, S. Ren, H. W., and Zhang, X. Scope: scalable consistency maintenance in structured p2p systems. In *IEEE INFOCOM* (2005).

[25] X. Liu, J. Lan, P. S., and Ramaritham, K. Consistency maintenance in dynamic peer-to-peer overlay networks. *Computer Networks 50*, 6 (2006), 859–876.

[26] X. Tang, J. X., and Lee, W. C. Analysis of ttl-based consistency in unstructured peer-to-peer networks. *IEEE TPDS 19*, 12 (2008), 1683–1694.

[27] Yu, H., and Vahdat, A. Design and evaluation of a continuous consistency model for replicated services. In *OSDI* (2000).

[28] Yu, H., and Vahdat, A. The costs and limits of availability for replicated services. *ACM TOCS 24*, 1 (2006), 70–113.

[29] Z. Li, G. X., and Li, Z. Efficient and scalable consistency maintenance for heterogeneous peer-to-peer systems. *IEEE TPDS 19*, 12 (2008), 1695–1708.

[30] Z. Wang, S. K. Das, M. K., and Shen, H. An efficient update propagation algorithm for p2p systems. *Computer Communications 30*, 5 (2007), 1106–1115.

[31] Zhang, C., and Zhang, Z. Trading replication consistency for performance and availability: an adaptive approach. In *IEEE ICDCS* (2003).

[32] Zhao, M., and Figueiredo, R. J. Application-tailored cache consistency for wide-area file systems. In *IEEE ICDCS* (2006).