Appears in *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems* (ICDCS '03).

The Hash History Approach for Reconciling Mutual Inconsistency

Brent ByungHoon Kang, Robert Wilensky and John Kubiatowicz
CS Division, University of California at Berkley
{hoon, wilensky, kubitron}@cs.berkeley.edu

Abstract

We introduce the hash history *mechanism for capturing dependencies among distributed replicas. Hash histories, consisting of a directed graph of version hashes, are independent of the number of active nodes but dependent on the rate and number of modifications. We present the basic hash history scheme and discuss mechanisms for trimming the history over time. We simulate the efficacy of hash histories on several large CVS traces. Our results highlight a useful property of the hash history: the ability to recognize when two different non-commutative operations produce the same output, thereby reducing false conflicts and increasing the rate of convergence. We call these events* coincidental equalities *and demonstrate that their recognition can greatly reduce the time to global convergence.*

1. Introduction

Optimistic replication is widely used to achieve increased availability and performance. Replica sites can create and disseminate updates without expensive global coordination such as locking or serialization. Updates are typically propagated epidemically, with replica sites converging to a consistent state by reconciling updates with one another. During reconciliation, if a site's latest version is a revision of another site's (*i.e.,* this version *dominates* the other version), then this fact must be efficiently recognized so that the more recent version can be replicated. If neither version is based on the other (i.e., they conflict), then one site can merge deltas based on a common ancestral version and send the merged version to the other site. Thus, the reconciliation process between replicas needs a mechanism to determine which version is a revision of another, or if two versions have arisen via separate modifications. In addition, sites need to maintain logs of deltas labeled with unique ids to figure out the set of deltas that needs to be exchanged during reconciliation.

Version vectors [6, 20] provide a mechanism for reconciling replicas by detecting conflicts [6, 18, 17, 14] and for determining the exact set of deltas to be exchanged for reconciliation [15, 22]. In version vector based approaches, each replica site maintains a vector of entries to track updates generated by other sites. Then, dominance relations between distributed replicas can be performed without a globally synchronized clock.

Unfortunately, version vector approaches do not scale well as the number of replica sites increases. The management of version vectors becomes complicated since entries for newly added sites have to be propagated to other replica sites [2, 15, 18, 16]. This problem is especially pronounced for peer-to-peer sharing of mutable data where it is reasonable to assume a given object has tens of authors and thousands of readers. In peer-to-peer systems, updates from any writer can be delivered through epidemic disseminations [1] such as anti-entropy in which each site periodically reconciles with a randomly chosen site. A reader needs to determine the dominance relation among updates from multiple writers. With dynamic version vector mechanisms, a writer membership change has to be propagated to all the readers. Depending on when the membership change information arrives, a reader may receive two updates with vectors of different sizes.

As an alternative, this paper proposes a *hash history* scheme for reconciling replicas. In our scheme, each site keeps a record of the hash of each version that the site has created or received from other sites. During reconciliation, sites exchange their lists of hashes, from which each can determine the relationship between their latest versions. If neither version dominates the other, the most recent common ancestral version can be found and used as a useful hint to extract a set of deltas to be exchanged in a subsequent diffing/merging process.

Unlike version vectors, the size of a hash history is not proportional to the number of replica sites. Instead, the history grows in proportion to number of update instances. However, the size of history can be bounded by flushing out obsolete hashes. A version hash becomes obsolete if every site has committed that version. The simplistic aging method based on loosely synchronized clocks can be used to determine if a given hash history is old enough to be flushed. In addition, a single complete hash history can be shared among replica sites, whereas, using version vectors each replica sites would have to maintain its own record.

The hash history approach is also economical in the storage overhead required for labeling log entries to extract sets of deltas that need to be exchanged during reconciliation. A unique version id is required for labeling every delta in the logs. Version ids in version vector based systems typically incorporate a local site name, whose size is not bounded, whereas using hash histories, it is possible to assign unique identifiers of fixed size without global coordination.

We simulated anti-entropy reconciliation using a hash history based approach with the trace data that was collected from CVS logs from `sourceforge.net`. The results show that the size of hash histories can be held to acceptable level—about 122 entries—with a 32 days aging policy and have no false conflicts due to aging. More importantly, the results highlight the fact that hash histories are able to detect the equality of versions that the version vector reports as a conflict—reducing the number of detected conflicts. Our simulations demonstrate that these *coincidental equalities* are remarkably prevalent, as shown by the vast difference in convergence rate between version vectors and hash histories.

The remainder of the paper is as follows: First, Section 2 introduces background concepts. Then, Section 3 introduces the hash history approach and discusses its advantages. Section 4 evaluates hash histories against user traces from `sourceforge.net`. Finally, Section 5 discusses related work and Section 6 concludes.

2. Background

Figure 1 shows a *version history graph* that tracks the dependencies between versions. This graph shows five different sites at which changes can originate. In this graph, each version is labeled with a subscripted pair indicating an epoch number and the site at which the version was created. Arrows indicate derivation of new versions from old ones. Any given version *dominates* all of its ancestors.

Consider an example of pair-wise reconciliation as illustrated by this figure. Here, site A creates $V_{0,A}$ and sends it out to site B and C. Then, site B transforms $V_{0,A}$ into $V_{2,B}$ by applying operation d_2; concurrently site C makes $V_{3,C}$ by applying d_3 to $V_{0,A}$ and site A makes $V_{1,A}$ by applying d_1 to $V_{0,A}$. Later, site A creates $V_{4,A}$ by merging $V_{1,A}$ and $V_{2,B}$ (operation m_4) during pair-wise reconciliation with site B. Here reconciliation is required since $V_{1,A}$ and $V_{2,B}$ do not dominate one another (*i.e.,* they *conflict* with each other). Similarly, site C creates $V_{5,C}$ by merging $V_{4,A}$ and $V_{3,C}$ (operation m_5). Later, when $V_{2,B}$ at site B reconciles with $V_{5,C}$ at site C, both sites conclude that $V_{5,C}$ *dominates* $V_{2,B}$, so that site B can accept $V_{5,C}$ directly as a newer version, without merging.

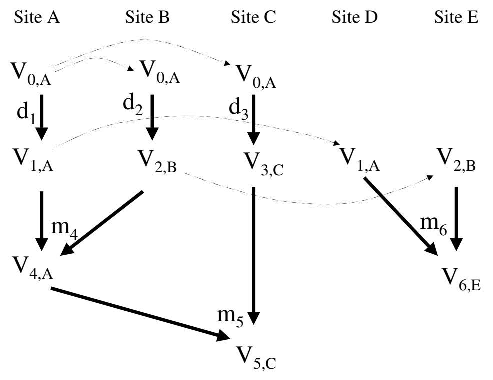

Figure 1. Version History: Replica sites collect versions from other sites, modify these versions, and merge versions together. One version *dominates* another version if it is derived from this version (*e.g.:* $V_{5,C}$ dominates $V_{1,A}$). When two versions are equal but neither dominates the other, we call this a *coincidental equality*.

The process of reconciling diverged replicas can be expressed as the process of exchanging deltas. Since each version is expressed by a series of operation deltas applied to the known previous state, one can efficiently select a correct set of deltas by maintaining some form of version history along with deltas. Hence, as in the above example, site B can receive operation deltas (d_1,m_4,d_3 and m_5) from site C instead of full version $V_{5,C}$. To do so, one needs to label each delta with a unique version id.

Occasionally, two independent chains of operations produce identical version data. We call such events *coincidental equalities*. As we will show in Section 4, recognizing coincidental equalities can greatly reduce the degree of conflict in the system by introducing aliasing into the version graph. When properly handled, such aliasing will increase equality and dominance rate during anti-entropy reconciliation because the equality information is conveyed to the descendants. In general, if V_1 and V_2 are considered equal, then all the versions that are based on V_2 will dominate V_1. If V_1 and V_2 are considered as in conflict, then all the versions that are based on V_2, will be in conflict with V_1.

It is important to note, however, that the version history graph does not recognize coincidental equality. In Figure 1, for instance, site E creates $V_{6,E}$ by merging $V_{1,A}$ and $V_{2,B}$ (operation m_6). Since site E and site A cannot determine whether $V_{6,E}$ and $V_{4,A}$ are the same or different until they meet each other during anti-entropy reconciliation, the IDs have to be assigned differently. Then, all the descendants of $V_{6,E}$ (e.g., $V_{7,D}$) are in conflict with $V_{4,A}$, although, in fact, all the descendants of $V_{6,E}$ could dominate $V_{4,A}$. This inability to exploit coincidental equality is a consequence of tracking ancestry independent of content.

2.1. Causal History

The *causal history of an event* [20] is defined as the set of events that causally precede a given event. The version history of Figure 1 can be considered one restriction of this definition: a directed graph of predecessors for each version. Version history can be implemented by maintaining the set of causal predecessors and the list of (parent, child) relations between versions in the set. In general, parent-child relations among versions are not required to determine the dominance relation or conflict between versions. However, the parent-child relations can be used to figure out the exact set of deltas to transform one version into another.

Let C(v) be the version history of a version v; if v_1, v_2 are unique and not equal then,

(i) v_1 dominates v_2 iff v_2 belongs to C(v_1)
(ii) v_1 and v_2 are in conflict iff v_1 does not belong to C(v_2) and v_2 does not belong to C(v_1)

For example, in Figure 1, the versions $V_{0,A}$, $V_{1,A}$, and $V_{2,B}$ causally precede $V_{4,A}$; hence, the version history of $V_{4,A}$ is the set of causal predecessors: $\{V_{0,A}, V_{1,A}, V_{2,B}, V_{4,A}\}$ and the set of parent-child relations: $\{(V_{0,A}, V_{1,A}), (V_{0,A}, V_{2,B}), (V_{1,A}, V_{4,A}), (V_{2,B}, V_{4,A})\}$. Thus, one can determine that $V_{4,A}$ dominates $V_{1,A}$, $V_{2,B}$ and $V_{3,C}$ by using only the set of causal predecessors. A hash-table based technique can make this determination efficiently.

2.2. Version Vectors

Causal histories are impractical because their size is of the order of total number of versions in the system. Moreover, causal histories cannot uniquely name versions that are created and merged independently at each replica site in a partitioned network. *Version vectors* were designed to overcome these challenges [6, 20].

A version vector is a vector of counters, one for each replica site in the system. In the version vector method, each site maintains a version vector to describe the history of its own local replica. When generating a new local version or merging other versions with its local version, a replica site increments its own entry in its local version vector. Further, when two versions are merged, every entry in the merged version vector should have higher or equal value than the corresponding entries in the previous two version vectors.

The dominance relation is determined by comparing all the entries in the version vector. Let VV(v) be the version vector of a version v; then,

(i) Either v_1 *equals* v_2 iff all the entries VV(v_1) are the same as all the corresponding entries in VV(v_2).
(ii) Otherwise v_1 *dominates* v_2 iff all entries in VV(v_2) are not greater than corresponding entries in VV(v_1).
(iii) Otherwise, v_1 and v_2 are in *conflict*.

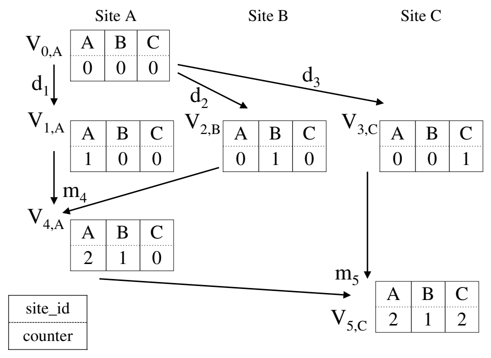

Figure 2. Reconciliation using version vectors

Figure 2 illustrates this process. The version vector for $V_{1,A}$ is [A:1,B:0,C:0] after site A generates a new write from $V_{0,A}$, whose vector is [A:0,B:0,C:0]. When site A reconciles with site B, the version vector for the merged result (i.e., $V_{4,A}$) might be [A:1,B:1,C:0]; however, since one must conservatively assume that each site may apply writes in different orders, we increment the entry of A's merged version from [A:1,B:1,C:0] to [A:2,B:1,C:0], since A determined the ordering of the merge.

2.3. Limitations to Version Vectors

However, using version vectors for optimistic replication has the following limitations [15, 18, 6, 2, 3]:

(1) Replica site addition and deletion require changes in version vector entries of all replica sites
(2) The size of version vectors grows linearly with the number of replica sites.
(3) Each site name has to be uniquely assigned.
(4) The version vector scheme cannot accommodate coincidental equality, i.e., cannot readily exploit instances in which different schedules of semi-commutative operations independently produce the same result.
(5) A unique version ID is still required for labeling each entry in the logs to extract deltas during reconciliation.

In the case of coincidental equality (4), the content of the versions is the same, but the version vectors would be interpreted as requiring reconciliation. Using version vectors (or a causal history approach based on timestamps—see section 3.1), one could reduce the false conflict rate by retroactively assigning the same id when two versions are found to have the same content during reconciliation. However, until equal versions (say V_1 and V_2) meet each other

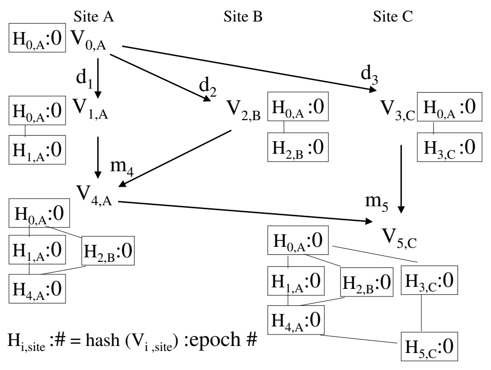

Figure 3. Reconciliation using hash history

for reconciliation, one cannot discern that descendants of V_1 and V_2 are in fact from the same root.

To label deltas (5), one could use version vectors; however, each delta can be labeled more economically with a [siteid,timestamp] pair, as is done in Bayou. Since each entry of the version vector tracks the most recent updates generated by the corresponding site, the timestamp of the latest write compactly represents all the preceding writes (i.e., the coverage property). Thus, given a version vector from another site, we can collect all the deltas whose version-id (i.e., [siteid,timestamp]) is not covered by that version[15]. The storage consumption for labeling log entries using the version vector based method is in the order of (size of version id $\times$ number of log entries), which is approximately the same as the storage requirement for causal histories.

3. The Hash History Approach

We now describe a scheme that overcomes the above limitations. Basically, the idea is to take a causal history approach, which readily addresses problems of the dynamic membership change (1) and the growth of vector size (2), and use a hash of a version as a unique ID, thus addressing the unique site name (3), the coincidental equality (4) and delta labeling requirement of (5). Also, since (5) shows that version vectors have a storage requirement of the same order as causal histories do, this scheme does not impose an additional cost penalty. We will use HH as an abbreviation for Hash History and VV for Version Vector.

3.1. Using Hashes as Version IDs

In returning to the version history, we must identify some mechanism for identifying replica versions. A timestamp that is locally assigned within a site is *probably* unique in practice since it is extremely rare that different users (or sites) make an update at the same time. However, such rare events do happen in practice, and their consequence is prohibitive: An update may be completely lost by different versions having the same id. Indeed, such an occurrence is present in the CVS logs of sourceforge.net, whose granularity is in seconds. Hence, we reject simple timestamps as too risky.

If a local timestamp is prefixed by a unique site name, then the version ID that is composed of [unique site ID, local timestamp] is guaranteed to be unique. However producing a unique site name in a network-partitioned environment requires a recursive naming method. For example, in Bayou[15], a new site name is prefixed by the unique id of the introducing site. If the sites are introduced through linear chaining (e.g., A introduces B, B introduces C, C introduces D, and so on), the total size of all site names could grow quadratically in terms of number of sites.

Instead, we can name versions by applying a cryptographic hash function (*e.g.*, SHA-1 or MD5) over serialized bytes of data content. By using the hash of a version as a unique ID, we automatically recognize coincidental equalities since the hash would be the same if the same results were produced from two different schedules of semi-commutative operations. However, by itself, the hash of a version is not necessarily unique, since a version with the same content may appear previously in a version's history, and hence the latest version can be mistaken for an old one during reconciliation with other sites. Therefore, we add an epoch number to distinguish the hash of the latest version from that of old versions with the same content.

When a new version is created, each site checks whether the same hash can be found in the history; if so the epoch number of the current version hash is assigned by increasing the largest epoch number of the versions with the same hash. It is possible, of course, that two sites generate the same content independently with the same epoch-number. We simply stipulate that these are the same versions, even though they are not truly causally related.

3.2. Hash History Based Reconciliation

We use the term "hash history"(HH) to refer to schemes in which version histories comprising hash-epoch pairs are used to encode causal dependencies among versions in a distributed system. Note that the hash history also contains the set of (parent, child) pairs of the hash-epoch pairs. Figure 3 shows an example of causal history graph of hash-epoch pairs. If $H_{4,A}$ is the same with $H_{0,A}$, then the epoch number for $H_{4,A}$ will be increased by 1, although the example shows epoch number 0 for $H_{4,A}$ since $H_{4,A}$ is not the same with $H_{0,A}$.

Hash history based reconciliation is able to capture coincidental equality automatically. Using version vectors or causal history based on a unique id (e.g., timestamp), one could reduce the false conflict rate by assigning the same id when V_1 and V_2 are found to have equal content during reconciliation. This remedy may work to a degree; however, until V_1 and V_2 meet each other for reconciliation, all the descendants of V_1 and V_2 would be unable to tell they are from the same root.

Each hash of a version in the hash history can be used as a label for the corresponding operation delta. Given a hash of a version from site A, site B can locate the matching hash in the site B's history, and then traverse the history graph toward the most recent copy while collecting all the deltas and all the siblings of the matched hash along the way.

Since one single hash can replace the unique version-id (e.g., [siteid,timestamp]), the storage consumption for tagging the log entries is in the order of (size of hash $\times$ number of log entries). The actual size of hash is fixed (e.g., 160 bits for SHA1) while the site-id could grow depending on the site creation pattern.

3.3. Truncating the Hash History

Classical techniques for truncating logs can be applied toward pruning hash histories. The *global-cutoff timestamp* (e.g.,[19]) and the *acknowledgment-timestamp* (e.g., [10]) can efficiently determine the committed versions; however, these methods fundamentally require one to track the committed state per each site, and hence would not scale to thousands of sites.

Instead, we use a simple aging method based on roughly synchronized timestamps. Unlike version vectors, the hash history for the shared data can be readily shared and referenced among many sites since it does not contain site-specific information, but rather the history of the shared object. Thus, one can maintain the truncated histories locally, archiving portions of the history at primary sites to handle the case in which a version that belongs to the pruned hash history would otherwise be mistakenly considered a new version. Note that the dominance check with pruned hash history is conservative in a sense that it would mistakenly consider dominance as a conflict, thereby triggering a merge process; hence, no updates would be lost.

4. Evaluation

To evaluate the efficacy of the hash history approach, we implemented an event-driven simulator. Our goal was to explore whether or not hash history schemes would converge faster and with a lower conflict rate than version vector schemes. We also sought to explore the sensitivity of hash histories to the rate at which we truncated them.

	Dri	Freenet	Pcgen
# events	10137	2281	404
# users	21	64	39
Duration	4/27/1994-5/3/2002	12/28/1999-4/25/2002	1/17/2002-4/12/2002
AVG interval	101.3 min	237.8 min	225.4 min
Median	0.016 min	34.6 min	2.16 min

Table 1. Trace data from sourceforge.net

4.1. Simulation Setup

The simulator performs anti-entropy reconciliation of information across a set of replica sites. It reads events in sorted order from a trace file and generates write events. The events are in the form of [time, user, filename]. If the user is new, we create a new site for the user. Each site has logs, hash histories and static version vectors. Periodically, the simulator performs anti-entropy by picking two sites at random. The first site initiates the reconciliation with the second site by getting a hash history and a version vector. The first site determines the equality, conflict and dominance. In case of conflict, the first site merges the conflicts, and then sends the merged version back to the second site along with the updated version vector and hash history.

The simulator repeats the anti-entropy process at every 60 seconds. For example, if the interval between events is 1200 seconds, 20 anti-entropy cycles is performed. The conflict moving rate is defined as the number of conflict determination results over moving window of 100 anti-entropy cycles.

The implementation of hash history, the simulator, and other simulation results that are not presented in this paper can be found at http://www.cs.berkeley.edu/∼hoon/hashhistory/.

4.2. Trace Data

In optimistic replication systems, small individual writes are aggregated until they are exported to another site. Therefore, we would prefer trace data that shows the inter-commit time rather than inter-write time. (Here, committing a write means that the write needs to be propagated to other replicas.) File system traces are not suitable for this purpose since they do not carry the information that the write is committed with the user's intention. Thus, we created test data based on CVS logs. CVS (Concurrent Versioning System) is a versioning software system that enables different users to share mutable data by checking-in and checking-out at a centralized server. CVS provides a serialized logs (update history) for each file in a shared project. We treat the project itself as under optimistic replication control and

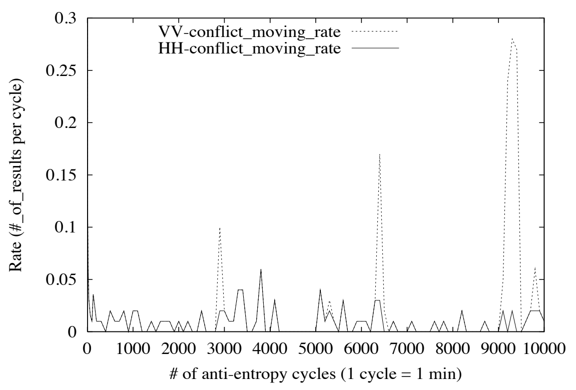

Figure 4. Conflict rate of VV and HH from pcgen shown between 0 to 10000 cycles

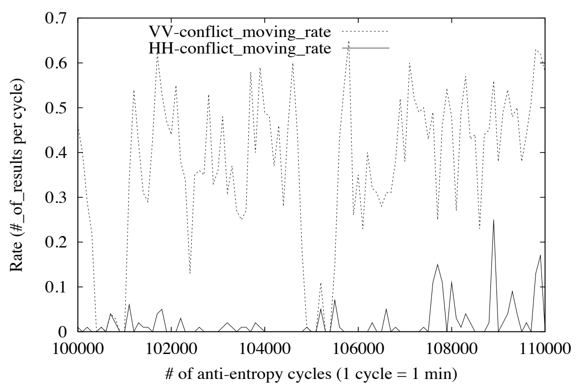

Figure 6. Conflict rate of VV and HH from dri shown between 100000 to 110000 cycles

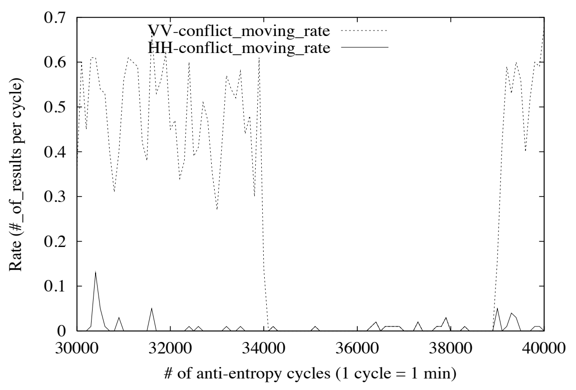

Figure 5. Conflict rate of VV and HH from freenet shown between 30000 to 40000 cycles

consider the individual files in the project as items of shared document content. We treat each user as one replica site.

We collected the CVS logs of three active projects from `sourceforge.net` that provides a CVS service to open source development communities. We first combined all the CVSlogs of the files in a project and then made the result into one serialized trace of events by sorting the events. Table 1 shows that the writes are bursty–the median is far smaller than the average of the inter-commit time.

4.3. Comparison with Version Vector Result

To check the correctness of our implementation, we ran the simulation forcing the hashes of merged writes to always be unique. In this case, the results of the dominance checks in the version vector scheme should be the same as those of the hash history implementation, and, indeed, they were.

The hash history scheme converges faster, and with a lower conflict rate, than the version vector scheme. Figure 4 shows that HH converged faster for the writes that was generated around at 9000 cycles in pcgen trace data. Interestingly, HH converged twice during the period between 9000 and 9500 cycles, while VV converged once around at 9500 cycle. This effect is also shown between 30000 and 34000 cycles with freenet trace data in Figure 5. HH converged around at 31000 cycles for the writes generated around at 30000 cycles; however, the VV could not converged completely and yet had more conflicts when the next writes were introduced. VV could not converge until the long non-bursty period between 34000 and 39000 cycles. Dri trace data in Figure 6 shows similar effect from 101000 to 105000 cycles.

One might wonder why there is so much difference between VV and HH. This is due to the fact that HH was able to capture the coincidental equalities and thereby treat two different sets of deltas that lead to the same content as the same delta. This has cumulative effects to the dominance relations among all the descendant versions. For example, let V_1 and V_2 are independently created with the same content but different version histories. If V_1 and V_2 are considered equal as in HH, then all the versions that are based on V_2 will dominate V_1. However, using VV, all the descendants of V_1 and V_2 would not be able to tell they are from the same root. In VV, each descendant of V_2 is in conflict with V_1. It is important to note that the simulation assumed the merged results of each descendant of V_2 and V_1 will be a new version with a different id but with the same content of the descendant of V_2 itself. In contrast, the merged version in HH will have the same id as the descendant of V_2. This is because the strict VV implementation in general conservatively assigns a new id for all the merged operations without looking at the content and its parents. And we simply

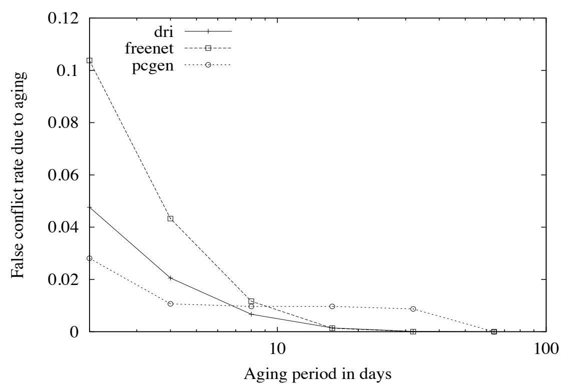

Figure 7. False conflict rate due to aging

Aging period (days)	HH size (number of entries)			
	Dri	Pcgen	Freenet	Average
32	146.3	159.1	61.5	122.3
64	413.9	443.9	147.5	335.1
128	551.5	591.7	612.8	585.3

Table 2. Average HH size with the aging period

assumed that the content of the merged result between a version v and its ancestors (including coincidentally equivalent ones) will be the same as that of the version v. That is the reason that the Figure 4 - 6 show more drastic difference as anti-entropy cycles increases.

4.4. Aging Policy

The results show that aging method is effective by holding the size of hash histories to an acceptable level—about 122 entries—with a 32 days aging policy and have no false conflicts due to aging. A false conflict could occur when the pruned part of the hash history is required for determining the version dominance. For example, V_1 from site A belongs to the hash history of the site B. After the site B pruned out the V_1 from its hash history because V_1 became too old according to the aging parameter (say 30 days), then the site B no longer be able to determine the dominance when V_1 is presented as a latest copy from A. Figure 7 shows that, using a pruning method based on aging, the false conflict rate due to pruning the hash history converges to 0 after 32 days. The average number of entries in a hash history with an aging policy of 32 days is measured as 122.3 (in # of entries) as shown in Table 2.

5. Related Work

In the context of mobile communication and reliable message delivery [4], it is well known that the dependency among events can be determined by keeping a history of all the causally preceding events. For example, Prakash et al. [16] proposed the use of dependency sequences that contain all the causal predecessors of an event as an alternative to the version vectors for mobile communications. A causal history based approach has also been proposed to address the problem of scalability of version vectors. However, using causal histories, a method is required to provide unique ids for replicas. Almeida et al. [2, 3] presented a unique id assignment technique for replicated versions, in which the bit vector names are expanded minimally enough to be distinguishable from other concurrent replica versions. When the diverged versions are merged later, the names are compacted into the name of their recent common ancestor.

In Coda [11, 13], the latest store id (LSID) is used to determine version dominance by checking whether an LSID appears in another replica's history, as per causal history approaches. Since it is impractical to maintain the entire update history of a replica, a truncated version is maintained, along with the length of the log history.

A number of recent peer-to-peer systems have used hashes based on SHA-1 or MD5 to identify documents or blocks of data. For instance, CFS[7], Past[9], Publius[21], FreeHaven[8] and FreeNet[5] identify read-only objects by a hash over the contents of those objects. OceanStore[12] goes further and uses hashes to identify read-only data blocks, then builds a tree of versions from these blocks. The resulting data structure contains both a tree of version hashes as well as pointers to all of the data from these versions.

Parker et al.[6] presented static version vectors as an efficient mechanism for detecting mutual inconsistency among mutable replicas during network partition. They also mentioned that extraneous conflicts may be signaled when two replicas are equal, but did not clearly note that the false conflict result can affect less number of dominance results among the descendant versions. Using static version vectors requires site membership to be previously determined. Static version vectors have been used to implement optimistic file replication in Locus and Ficus [17, 14].

Ratner et al.[18] noted the scalability problem of static version vectors and proposed a dynamic version vector mechanism. In this approach, entries of a version vector can be dynamically modified rather than statically pre-assigned. The method dynamically adds an active writer into a version vector and expunges an entry from a version vector when the corresponding writer becomes passive. This method can alleviate the scalability problem to a degree, at the cost of adding the complexity of tracking whether a site becomes passive or not.

Methods for dynamic replica creation and retirement using version vector were presented in Bayou [15]. In this approach, creation and retirement events are treated as writes, so that such events can have a causal accept order with other writes in the log. However, this method may require one to look through the write logs during dominance determination using the version vector.

6. Conclusion

In this paper, we presented the hash history approach for detecting mutual inconsistency and reconciling conflicts. Hash histories are causal histories that use the cryptographic hash of version content for naming versions. The hash history approach has four primary advantages over previous approaches: First, globally unique names can be determined without global coordination since the hash can be computed locally. Second, hash histories do not grow in size relative to the number of participating sites—an extremely important property in highly dynamic peer-to-peer environments. Third, hash histories recognize *coincidental equalities*, thus providing faster convergence with fewer conflicts than other approaches. Finally, the hash ID can have a relatively small fixed size, and hence the hash history approach has less or equal amount of the storage overhead for labeling log entries relative to the version vector based approach. We expect that hash history based reconciliation will be the technique of choice for highly-dynamic and global-scale Internet applications.

7. Acknowledgments

We'd like to thank Eric Brewer, Anthony Joseph, Mike Franklin, and John Chuang for their helpful discussions in developing hash history ideas. We also like to thank Jonathan Traupman and Byung-Gon Chun for their generous help in proof-reading this document. This research was supported by the Digital Libraries Initiative, under grant NSF CA98-17353, and by NSF Career Award ANI-9985250.

References

[1] A. Demers et al. Epidemic algorithms for replicated database maintenance. In *Proc. of ACM PODC Symp.*, 1987.

[2] P. S. Almeida, C. Baquero, and V. Fonte. Version stamps- decentralized version vectors. In *Proc. of IEEE ICDCS*, 2002.

[3] C. Baquero and F. Moura. Improving causality logging in mobile computing networks. ACM Mobile Computing and Communications Review, 2(4):62–66, 1998.

[4] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[5] I. Clark, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

[6] D. Stott Parker et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of ACM SOSP*, 2001.

[8] R. Dingledine, M. Freedman, and D. Molnar. The freehaven project: Distributed anonymous storage service. In *Proc. of the Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[9] P. Druschel and A. Rowstron. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of ACM SOSP*, 2001.

[10] R. A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems*, 5(4)(UCSC-CRL-92-31):379–405, 1992.

[11] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[12] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*, 2000.

[13] M. Satyanarayanan et al. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[14] P. Reiher et al. Resolving file conflicts in the ficus file system. In *USENIX Conference Proceedings*, Summer 1994.

[15] K. Petersen et al. Flexible update propagation for weakly consistent replication. In *Proc. of ACM SOSP*, 1997.

[16] R. Prakash and M. Singhal. Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems. *Wireless Networks*, 3(5):349–360, 1997.

[17] R. Guy et al. Implementation of the Ficus Replicated File System. In *USENIX Conference Proceedings*, Summer 1990.

[18] D. Ratner, P. Reiher, and G. J. Popek. Dynamic version vector maintenance. Technical Report CSD-970022, University of California, Los Angeles, June 1997.

[19] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering*, 13(1):39–47, 1987.

[20] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[21] M. Waldman, A. Rubin, and L. Cranor. Publius: A robust, tamper-evident, censorship-resistant, web publishing system. In *Proc. 9th USENIX Security Symposium*, 2000.

[22] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *Proc. of ACM SOSP*, 2001.