Summary Hash History for Optimistic Replication

Abstract

The unprecedented growth of the world's first non-profit, open-source encyclopedia has put considerable stress on the Wikipedia foundation, which is constantly looking for donations to support its rising infrastructure and hosting costs while maintaining adequate quality of service. That the public-owned content depends on a single organization's financial fate is a major concern to many. We propose using optimistic replication to ensure that the encyclopedia content is preserved at multiple sites managed by different organizations. Replicating the Wikipedia database requires not only an efficient update exchange protocol but also a mechanism to identify the origin of update pollution or "anonymous slander" as it is frequently referred to by Wikipedia users. In order to meet these challenges effectively, we introduce the Summary Hash History (SHH) approach. In this approach, each site maintains a tamper-evident update history to mitigate security challenges and to readily determine the exact set of updates to be transferred during peer-to-peer reconciliation between sites. We first implemented Basic-SHH which confirmed our intuition that SHH can be used for both the tamper-evident history and the efficient update exchange mechanism. However, our evaluations revealed that Basic-SHH is unable to guarantee convergence among replicas in scenarios involving concurrent updates. Thus, we developed a variant called Associative-SHH that overcomes Basic-SHH's limitations by not only providing eventual convergence but also enabling convergence of concurrent updates across partitioned networks.

1. Introduction

Wikipedia, the world's first non-profit, open-source encyclopedia, has witnessed unprecedented growth in its relatively short five year existence. As of September 2006, Wikipedia contained over 2,550,000 unique articles with more than 1 million in the English language alone [WK06]. Understandably, this rapid growth has put considerable stress on the Wikipedia foundation, which is constantly looking for donations to support its rising infrastructure and hosting costs while maintaining adequate quality of service of Wikipedia content. Users are concerned about whether the public owned content can be safely guarded if only one organization supports it.

We consider utilizing optimistic replication to collaboratively host the public content among large organizations such as public universities and libraries. Optimistic replication allows data to be replicated at various points (i.e., replicas) in the network [Ki92,Ra98,Sa05]. This would ensure that the encyclopedia content is accessible at multiple organizations, eliminating the current dependency on a single organization. Such a decentralized Wikipedia can also support its users with better service, given that optimistic replication is a proven technology to provide high data availability and improved performance more effectively than a centralized server [Ka88,Ki92,Mu95,Ra98,Sa00,Sa05,Ku00].

To permit efficient read and write operations, as well as to maximize content availability, optimistically replicated systems allow users to access any individual site. An update to one replica needs to be propagated to the other replicas using pair-wise exchanges, and concurrent updates need to be identified and resolved during this reconciliation process to ensure a consistent view across replicas. Notably, in order to bring consistency to the replicas, such update exchanges require an *efficient update propagation* mechanism that does not overload the network. Given that network bandwidth is still considered an expensive resource compared to disk and CPU resources, network efficiency has been one of the primary focuses of previous approaches to optimistic replication.

In a peer edited distributed application like Wikipedia, additional concerns arise: (i) a misbehaving (or perhaps misinformed) users can pollute the shared content by introducing false information – commonly referred to as "anonymous slander"[Se06]. (ii) Moreover, a malicious replica site can easily falsify the causal ordering between updates, propagate incorrect updates to replicas, and even halt the propagation of valid updates [Me87, Sc94, Sm94, Sp99], thereby preventing replicas from converging on consistent, correct information – update reordering/dropping attack.

In order to mitigate such security challenges, each site must maintain or have access to a *tamper-evident update history* [Ma02, Sp99]. Such a history contains a complete record of the updates that have been applied to the replicated content along with the causal relationships between such updates and is constructed such that any modification of the previous update history will be detected at other replica sites. This ensures that (i) the misbehaving users (e.g., source of article slander and pollution) can be made accountable. And the tamper-evident history can guarantee the undo of the article pollution into correct previous article. Also, (ii) the misbehaving sites (e.g., source of reordering/dropping attack) can be detected through post-inspection of previous update/merge history.

Our solution to such challenges is an interesting approach to optimistic replication based on each site maintaining a Summary Hash History (SHH) that addresses these requirements for (i) efficient update propagation

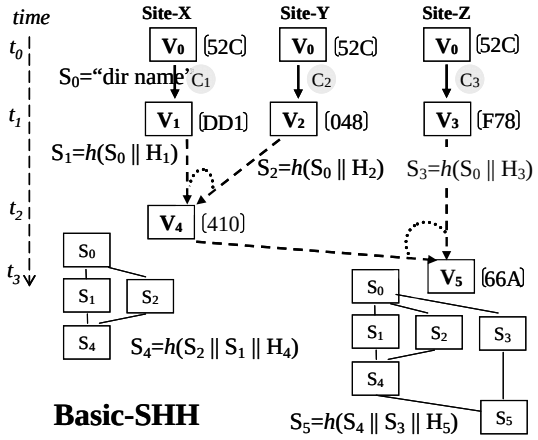

Figure 1: **B-SHH Example**. At time t1, Site X, Y and Z created new states S1, S2 and S3. At time t2, Site X pulled S2 (i.e., delta(S0,S2)) from Site Y and deterministically merges the two states: S1 and S2 , creating S4 state.

and (ii) tamper-evident update histories at the cost of maintaining a tamper-evident update history. With previous approach, version vector is used to provide efficient update propagation by figuring out exact set of deltas to be propagated, and a separate tamper-evident update history to ensure the correct undo and the accountability of the participants.

In our proposed SHH, version vector is not needed, freeing from the overhead of version vector maintenance. SHH uses a causal history approach [Sc94] as a decentralized ordering mechanism, where each site keeps a record of the updates that it has created and incorporated from other sites in the form of a secure version tree with SHs as version identifiers. Because it utilizes this unique summary hash identifier, SHH represents a tamper-evident update history that ensures the verifiability of updates and enables the system to protect against malicious compromise and faulty ordering through an undo/time-travel mechanism. Additionally, the SHH scheme supports efficient update propagation. During reconciliation, two sites exchange their SHHs, from which each can extract the minimal set of updates that need to be transmitted over the network to bring the sites into a mutually consistent state.

The effectiveness of SHH-based replication was tested by implementing S-Sync, a directory synchronization tool to share and synchronize files/folders between multiple distributed sites. With this experience we then built a replicated Wikipedia application (RepliWiki), using S-Sync as a pluggable framework. Each RepliWiki site periodically publishes article updates, which are aggregated into a file in the shared directory. These files are propagated to other RepliWiki sites through S-Sync protocol. RepliWiki is currently deployed on PlanetLab nodes and exists to demonstrate the usefulness of the SHH technique in a real world application.

However, we found two practical issues in building scalable, replicated systems using SHHs. First, the size of an SHH can grow unbounded, which can overload the network during reconciliation. Second, our initial implementation of S-Sync exhibited convergence problems when reconciling concurrent updates.

To address this first issue, we explored protocol variations based on SHH, in addition to using a well-known decentralized pruning technique. Instead of sending the entire SHH or only the latest SH, we found sending up to 'k' entries from the SHH can be beneficial and efficient, since it may cost the same amount to send 1 SH entry as to send 100 SH entries.

The second, perhaps more serious, problem arises because our initial implementation, called B-SHH, assigned a new summary hash identifier to the result of the merge-of SHHs, when two sites reconcile. Our analyses actually show that this can derail the whole reconciliation process by creating new versions in the SHH that are continually propagated to other replicas. Consequently, eventual convergence is not guaranteed in B-SHH. Our experiments not only show the detrimental affects of such entries on the version tree, they also demonstrate that B-SHH produces false conflicts when three or more concurrent updates are merged. For instance, reconciliation between randomly chosen sites leads to an abundance of *vacuous* (i.e. non-data-transferring) reconciliations, in which only summary hashes, not data content, are transferred during the reconciliation process.

These observations guided the design of our follow up SHH construction mechanism referred to as Associative-SHH (or A-SHH). In A-SHH, a merge identifier is a set of summary hashes of all previous revisions on which the merge is based, instead of a newly generated hash as in B-SHH. Further analysis has shown that A-SHH not only converges faster than B-SHH but also provides convergence of concurrent updates even across partitioned networks as long as each partition received the same set of updates before the network partitioned. Thus, A-SHH is the ideal choice for large distributed applications such as RepliWiki.

The rest of our paper is organized as follows. Section 2 introduces S-Sync and B-SHH. Then Section 3 presents the Associative-SHH including the problems in B-SHH. The implementation of S-Sync and RepliWiki are discussed in Section 4 followed by the evaluation of SHH design and results of our PlanetLab experiment in Section 5. Section 6 discusses the related work and we conclude in Section 7.

2. S-Sync with Basic-SHH

We first show how B-SHH version trees are constructed and used in S-Sync, a directory synchronization tool to share and synchronize files and folders among distributed

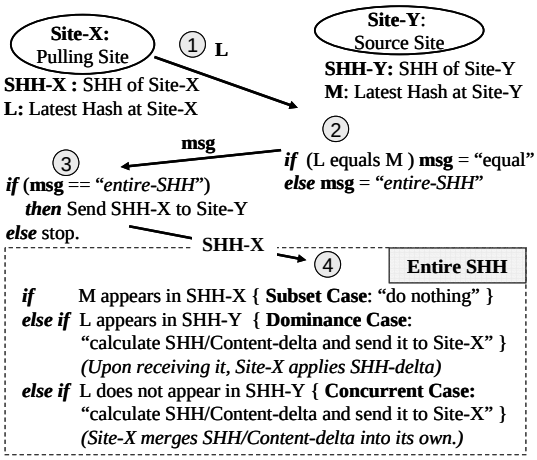

Figure 2: **S-Sync Protocol Details**. Figure shows the variations in S-Sync protocol: Latest Hash Equal, Subset case, Dominance case, and Concurrent case.

sites. Our implementation of S-Sync furnishes the underlying update transfer framework used in RepliWiki. Each RepliWiki site periodically publishes updates to its locally stored articles, which are aggregated into files in the shared directory. These files are exchanged with other RepliWiki sites through S-Sync.

S-Sync creates a shared space that houses files and directories for synchronization with other sites. S-Sync keeps information about versions and updated content in the form of local files which we call "change-sets". These change-sets are exchanged during periodic reconciliation between randomly selected sites.

2.1 Definition of Summary Hash and Basic-SHH

An SHH is a causal history using summary hashes as version identifiers. A version indicates a state that a replica site can create by applying the update(s), the subsequent changes due to content upload, and the various merge operations resulting in a new merged version.

S-Sync uses summary hashes as pointers into the change-sets to provide auditability and traceability in case of malicious user attack and undo/time-travel in case of accidental loss of data. In such cases, S-Sync can traverse the SHH version history tree to locate the changes that need be undone and provide users with the changes that precede the accidental deletion or the undesirable update.

Basic-SHH: Let the summary hash S_i is an identifier to represent a version V_i, where a version indicates a state of the synchronization unit. (e.g. snapshot of the shared directory). The identifier S_i, we call, Summary Hash (SH), is generated as below:

Let $H_i = \boldsymbol{h}(V_i)$, where $\boldsymbol{h}$ is a collision resistant hash function, and S_p is a V_i's predecessor's identifier.

- $S_i = \boldsymbol{h}(S_p||H_i)$ when V_i has a single predecessor.
- $S_i = \boldsymbol{h}(S_p^*||H_i)$ when V_i has multiple predecessors (i.e., V_i is created by merging multiple concurrent versions). S_p^* is the concatenation of multiple S_ps, where S_ps are sorted by lexicographical value of S_p.)

For instance, as shown in Figure 1, V_1's predecessor is V_0, thus S_i is $\boldsymbol{h}(S_0||H_1)$. Also, V_4 is a merge of V_1 and V_2, thus $S_4 = \boldsymbol{h}(S_2||S_1||H_2)$. Please note that S_2 = (048) comes before S_1 = (DD1) in lexicographical order (048 < DD1).

Tamper-Evident Verification: The inclusion of predecessor hashes in the summary hash is similar to that of Merkle's tamper-evident hash tree [Me87] or hash chaining structure [Ba92, Ha91]. Therefore, by using this summary hash as a version identifier, one can readily prevent various ordering attacks [Sp99]. Moreover, since the summary hash is collision resistant, it is computationally infeasible to find two different summary hash histories given the latest version's summary hash. This is important because there is a unique summary hash history associated with a given summary hash; a version's summary hash is a compact and secure summarization of all the causally preceding writes.

Furthermore, SHH achieves its tamper evident property by signing the latest summary hash, as one can easily verify the previous update history by traversing the signatures and matching them with the site that signed it. For example, to verify the summary hash S_5 for version V_5 in Figure 1, one needs to locate summary hashes for both S_3, for version V_3, and S_4, for version V_4, and check if the hash over $(S_3||S_4||H_5)$ matches S_5. If it does, then one can recursively verify S_3 and S_4 until reaching either a previously verified summary hash or S_0, the initial root.

Our current implementation provides an "S-Sync State Reconstruction" interface with which the user can specify a previous state that needs to be reconstructed. RepliWiki utilizes S-Sync's interfaces to trace and undo slanderous edits to articles.

2.2 Efficient Update Propagation using SHH

Efficiency in update propagation is at the heart of optimistic replication [Sa05]. In S-Sync, we craft summary hashes in a manner that is useful both for verifying the version-ordering and for figuring out the exact set of updates to be exchanged. The SHH data structure combines the modification (updates) and synchronization (reconciliation) histories in a single data structure, which is significantly simpler than those used with traditional version vectors and also provides a tamper-evident update tree.

2.2.1 Anti-Entropy Reconciliation with SHH

Optimistic replication protocols are flexible with respect to network topology because techniques such as epidemic algorithms propagate updates in a reliable fashion even when the communication between replicas is unstable or unreliable due to network partitions [De87,Sa05]. To be network efficient, S-Sync propagates updates between replicas via pair-wise reconciliations [De87,Te95].

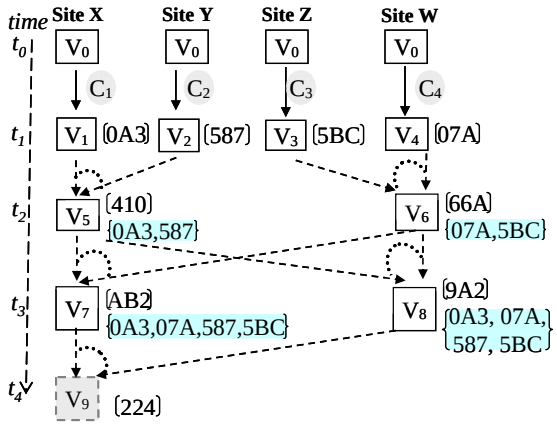

Figure 3: B-SHH creates new identifier for each merges in this case S7, S8 even if the merge will produce the same deterministic result. Thus, B-SHH may not converge in some unfortunate merge paths as above. However, the A-SHH, shown in {}, will be able to determine the equality in any arbitrary sequence of merges.

S-Sync employs a "pull" mechanism for reconciliation. The site that initiates reconciliation is called the *initiator* or the "pulling" site. The *source* is the site that responds to the initiator's request for new updates by checking if the two sites have any differences in their SHHs and then determining the summary-hashes that need to be transferred back to the pulling site.

2.2.2 S-Sync Protocol Explained

The S-Sync protocol, as in Figure 2, facilitates update exchange among replica sites using the following two variations. The first, called Entire SHH exchange sends the entire SHH during pair-wise reconciliation. The second variation sends the latest SH first. If the source requires additional SH's to compute the dominance, the entire SHH can be transferred. For instance, in the first step of the S-Sync protocol, the initiator or pulling Site X sends the latest hash (S_X) to the source or reconciling Site Y. If S_X is not equal to S_Y (Site Y's latest hash) then Site Y asks for the entire SHH from Site X. With the entire SHH at Site Y, the version dominance can now be determined by checking whether Site X's latest version (S_X) appears in its SHH. In a typical reconciliation, where Site X with latest hash S_X pulls from Site Y with latest hash S_Y, there can be one of the four possible cases detected at the source, as also shown in Figure 2:

Latest-Hash-Equal Case: The reconciling sites may have the exact same updates either because both sites have received the latest updates or because there have been no recent updates in the system. If the latest hashes (S_X and S_Y) of both sites are equal, this implies that the sites have the same content. If this is the case, nothing needs to be transferred back to the pulling site.

Subset Case: The pulling site may dominate the source, in which case the source site's version-tree is found to be a subset of the initiator's.

Dominance Case: The initiator's latest hash may already exist in the source's SHH, in which case the source site dominates. Having established dominance, the source-site (Site Y) must next determine the updates to be transferred back. To do so, the source-site calculates the difference between the two sites, referred to as the deltas.

The updates to Site Y's SHH are captured in, what we call an "SHH Delta". Since the SHs are simply the identifier for a version the changes in the versions are bunched together in a "Content Delta". Site Y transfers these deltas back, which are then applied using either B-SHH or A-SHH, to bring the initiator's state to the source's current state.

Concurrent Case: A concurrent case is where the reconciling sites may have common summary hashes in their respective version trees but have different latest hashes. For instance, Site X's SHH and Site Y's SHH are concurrent when these two SHH trees have some SHs in common but the latest hashes (S_X and S_Y) are not the same. In S-Sync protocol, Site Y calculates the SHH and Content Deltas, to be transferred back to Site X, where these deltas are merged.

2.2.3 Delta Calculation in SHH

The SHH Delta is obtained by traversing the SHH tree from top to bottom in a topologically sorted manner, this delta excludes the nodes which already appear in the other SHH tree. The following is a simple technique for computing the SHH Delta.

SHH Delta = *Edges containing topologically sorted nodes in SHH_Y - topologically sorted nodes in SHH_X, if appears in SHH_Y* -------- [I]

The edges in the SHH tree contain the actual data content. Thus, traversing the version tree for the edges provides the required updates in the system and corresponding version changes.

Content Delta = *Edges for the Nodes in SHH_Y with exactly one parent - Edges on Nodes in SHH_X with exactly one parent if appears in SHH_Y* -------- [II]

Upon receiving the deltas, the initiator performs one of the following two steps: (1) apply the delta if the source site dominates or (2) merge the delta if the sites are concurrent.

3. Associative-SHH

We wrote a SHH-Visualizer utility to visualize the complete SHH version tree, from which we found that S-Sync with B-SHH has a detrimental impact on the reconciliation protocol and cannot guarantee eventual convergence when reconciling concurrent updates. We show the visualization results for vacuous reconciliations in the following sections.

3.1 Vacuous Reconciliation of Basic-SHH

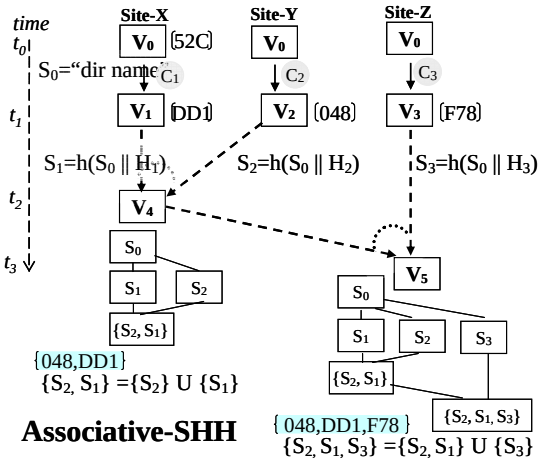

Figure 4: **Associative-SHH example**. The merged state is represented as a set of predecessors' identifiers. At time t3 when Site X merges S1 and S2, the merged state is represented as a set {S1, S2}.

Our experience with B-SHH shows that it creates a large number of merged summary hashes, in the SHH tree, due to the out-of-order (or random) reconciliation among replica sites. We attribute this behavior to the flaw in the construction mechanism of B-SHH which creates a new summary hash for every merge. During the reconciliation process the sites end up transferring these intermediate summary hashes but no real content is transferred. It is therefore fitting to label such a reconciliation as *vacuous or non-data transferring*.

The obvious fix for the above problem is to avoid vacuous reconciliations, which also happens to be the primary motivation for developing A-SHH. Experiments and further analysis show that the A-SHH construction mechanism consumes an order of magnitude less bandwidth than the B-SHH by avoiding these vacuous reconciliations. The results are discussed in the evaluation section.

Figure 3 shows vacuous reconciliation and how these intermediate version identifiers create a vacuous dominance situation. For instance, when Site Y calculates the SHH Delta and Content Delta, as described earlier, it determines that there is no content that needs to be sent back to the pulling Site X. The SHH Delta in this case is $\{(S_2,S_8), (S_6,S_8), (S_7,S_9)\}$. This reconciliation synchronizes both sites in terms of their SHH trees without sending the actual content deltas. The other possible reconciliation scenario is vacuous concurrence, where the sites have the same content but a different latest hash, which leads to the flagging of false conflicts when there are none. A-SHH on the other hand averts the need for any non-data transferring reconciliation due to its set-based design, thus providing faster convergence while consuming far less bandwidth than B-SHH.

3.2 Improvement in A-SHH over B-SHH

A-SHH's summary hash construction follows one of the following distinct approaches, as illustrated in Figure 4. If the version change is due to;

- Content Update, the construction follows the B-SHH's construction,
- Content Merge, it uses a set based concatenation.

In the case of Content Merge, the summary hash (S_i) is a union of S_i's parents in a lexicographically sorted order as we see in Figure 4. We refer to the latest hash in the A-SHH construction mechanism as the "latest-hash-set", shown as $\{S_1, S_2, S_3\}$. As given in Figure 4, the updates C_1, C_2, C_3 show the updates on the base version, V_0. Consequently, C_1 brings the base version V_0 into V_1 ($V_0 \rightarrow V_1$). The edges form due to this content update and thus are always representative of a version update and actual data changes in the system. The global SHH keeps information about these version transitions and the update information. Notably, all updates lead to states with only one parent, e.g., S_1, and S_2 and states with more than one parent indicate a merge operation, e.g., states $\{S_2, S_1\}$.

3.3 Eventual Convergence Guarantee in A-SHH

The fundamental property of optimistic replication design is to achieve consistency across replicas: that is, all sites move towards eventual consistency [Sa05]. Replicas held by two replica sites may vary in their content because of the order in which they receive and process updates. As the replicas try to achieve consistency by exchanging latest updates, it is necessary to identify updates and their order of arrival in order to avoid/detect replica inconsistency. Having a technique that can resolve inconsistencies due to merges based on faulty/incorrect ordering is also desirable.

Merge Properties in SHH: To correctly identify all inconsistencies that occur due to an update or an out-of-order merge, it is imperative to first identify the merge operation and subsequently develop techniques to facilitate identification of the correct update-order. Please recall that any version tracking mechanism in optimistic replication should be able to assign the same version identifier to the final merged content if the merges are deterministic [Sa05]. We extend this property and introduce two additional merge properties: commutative and associative merges. The individual merge in SHH can thus have one or more of the following merge properties: Deterministic, Commutative, and/or Associative. Our approach to correct update-ordering and version-tracking for eventual convergence, in view of concurrent updates, follows these merge properties. We first define these properties and then illustrate how SHH incorporates them in its construction mechanisms.

•D-Merge: Merge, m(x,y), is said to be deterministic if it produces the same merged result from inputs x and y, irrespective of the site that performs the merge operation.

•C-Merge: Merge, m(x,y), is commutative if m(v1,v2)

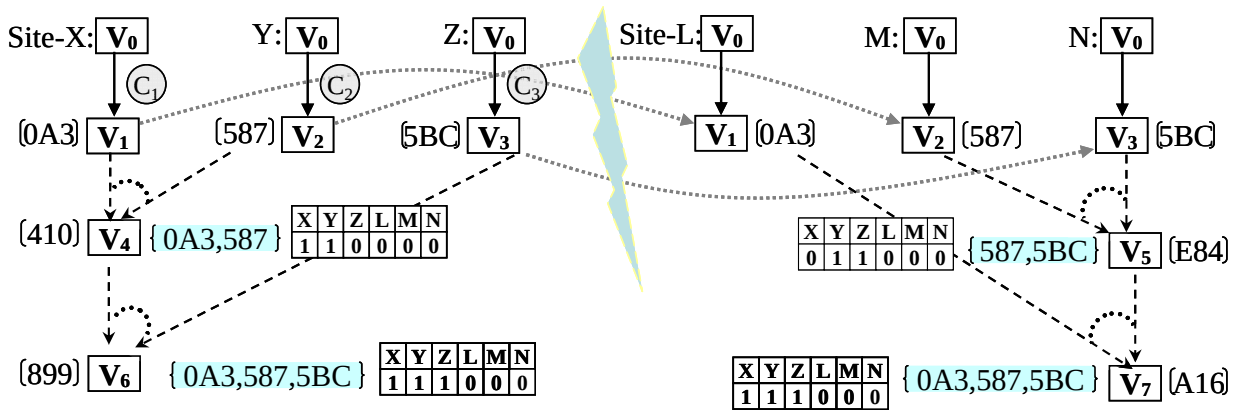

Figure 5: **Example of convergence across partitions**. Convergence across Partitioned Network Compares B-SHH, A-SHH, and Bayou-VV. The Sites L, M and N receive the updates (C1, C2 C3) by reconciling with sites X, Y and Z respectively, before the partition. Since the updates were made of different files, the merges produced by them are commutative and associative.

produces the same merged output as m(v2,v1). E.g., A+B = B+A.

•A-Merge: We call a merge operation associative if m(v1,m(v2, v3)) produces the same merged result as m(m(v1, v2), v3) . E.g., A+(B+C) = (A+B)+C.

B-SHH assigns different version identifiers, 899 and A16 in Figure 5, on sites X and N respectively, even though the data content (i.e., X_6) is the same on both the sites. The assignment of different version identifiers for the same merge operation is made because B-SHH does not handle the associative properties. This also results in flagging the merge as conflicting which is obviously a false-conflict. However, the B-SHH construction correctly handles the deterministic and commutative merge properties as it creates a new summary hash for all newly introduced updates, be they Content Updates or Content Merges. However, A-SHH, in Figure 5, assigns the same version identifiers {0A3, 587, 5BC} on both the sites by correctly capturing D-Merge, C-Merge and A-Merge. The A-SHH construction mechanism does this by taking the union of lexicographically sorted summary hashes of the parent's when all the merges are DCA. A-SHH is more efficient than B-SHH in managing the summary hash and merging updates with other sites.

The merged summary hashes capture the meta-information of the merge operation, including the order of merge and if it meets any of the D, C, or A merge properties. However, should the merge be non-DCA, both the B-SHH and A-SHH flag the merged summary hash and report it as a potential conflict. The conflict resolution in both SHH construction approaches assumes knowledge of some kind of application-specific semantics.

Decentralized Convergence in SHH: It is assumed that identical versions with the same content will be produced at different sites, possibly as a result of merges between reconciling sites.Indeed, such an occurrence can be frequent when the same deterministic merge procedure is used to resolve the same set of conflicting updates. SHH will assign the same version identifier if the identical content is independently produced from the identical histories. Interestingly, this property of SHH allows distributed replicas to converge even across partitioned networks. For instance, in Figure 5, a comparison of B-SHH, A-SHH, and Bayou–VV [Te95] show how each of these techniques will identify and assign version identifiers.

The merge operation on different replica sites can assign the version identifier independently without any communication; it is possible for a site in each partitioned network to assign the same version identifier. However, this is difficult to capture in a decentralized setting. Therefore, unless two sites communicate with each other, the sites cannot recognize that each site has independently produced an identical version. If different content version identifiers are not assigned to resolve this problem then SHH will report concurrent update, thus introducing possible conflict in the system. We refer to such a scenario as *false conflict* cases. Unfortunately, such false conflicts have a vast cumulative effect on any future descendant versions. For example, if V_1 and V_2 are considered in false conflict, then all the versions that are based on V_2 will be in conflict with V_1. This false conflict will cumulatively create further false conflicts among descendant versions. In contrast, if V_1 and V_2 are not in conflict, then all the versions that are based on V_2 will dominate V_1. This cumulative false conflict not only incurs the unnecessary overhead of running the Conflict Resolver, but can also create an undesirable merged result among descendant versions.

Figure 5 shows three concurrent updates in the system. Since these updates are on different files, the merges produced by them are by definition deterministic, commutative and associative, as discussed earlier. The final output produced by merges (M_1, M_3) and (M_2, M_4) on Site X and Site N should therefore be the same in terms

of their data content. We also show the affects of associative merge property of A-SHH in terms of its convergence across partitioned networks in Figure 5. It is important to note that the A-SHH was designed to converge faster without creating any false conflicts across partitioned networks. However, B-SHH promises an identical version identifier only in the case where either the partition or the converging site has received the updates previously, and only when the sites attempt convergence of two deterministic concurrent updates.

A-SHH, on the other hand handles any number of deterministic concurrent updates. For instance, update represented as A, B, and C will be merged as: A+(B+C) = (A+C)+B = (A+B)+C. This guarantees the identical final version identifier regardless of the order of the reconciliation and network partitions. This is accomplished through "set" based concatenation of its lexicographically sorted parents. In this way, by correctly resolving the different partitions, A-SHH guarantees that no false conflict will occur in the system. B-SHH, on the other hand, mistakenly treats these two versions 899 and A16 as concurrent versions on Sites X and N respectively (see Figure 5) and therefore will require a vacuous reconciliation before full convergence can take place.

4. Overcoming SHH Overheads

Any attempt to collate causal histories into a single data structure significantly affects its size [Sa05]. In the same respect, the size of the SHH grows in proportion to the number of update instances in circulation. Another important challenge for our S-Sync protocol is to conserve the bandwidth consumption, either by reducing the size of the SHHs being sent over the network or by restricting the number of round trips. To address these issues, in addition to using a well-known decentralized pruning technique we also explored a protocol enhancement to the S-sync protocol.

4.1 Acknowledgement List for Log Pruning

As it is based on a causal history, the size of the SHH can be considered unbounded. To be network efficient, SHH need to be pruned periodically. We take a common approach used in other optimistic replication techniques [Go92, Sa02]. We leverage SHH's reconciliation process for both tracking the update and transferring its acknowledgement receipts. An acknowledgement receipt establishes, for a given version, which replica sites have received that particular version.

To provide an audit trail for possible malicious updates and to provide undo guarantees against any unintended deletes, it is necessary to assure that at least one site (called the archiving site) in the system maintains the complete history for any future verification. The archiver-site is assumed to have a fairly large storage to keep the update history along with the acknowledgement receipts from other sites.

A site can safely delete the update histories and their change-sets up to state S_X, if the archiving site has received all update up to S_X. If the archiving site is not available for an elongated period, each site can make its own pruning decisions based on the acknowledgement receipts that it has collected. We assume that each site will use the global membership information that it has. (i.e., participantsList.txt file in S-Sync implementation)

The update receipts are propagated as part of the shared directory in S-Sync. A special shared folder called "AckList" collects the summary hashes that were signed by the receiving site. The anti-entropy reconciliation synchronizes the AckList folders across the sites, exploiting the acknowledgement-receipts collected at other sites. Sites can independently arrive at the pruning decision as below:

(a) Prune SHHs up to version V_i if the archiving site has received this state and delete the corresponding change-sets after the SHH is pruned.

(b) Prune SHHs up to version V_i, provided that all sites have received that version based on the current acknowledgement list, if the archiving site is not reachable.

4.2 Optimization in S-Sync Protocol: *k*-SHH

Our initial attempts at optimizing the reconciliation protocols were based on the decision to transfer either the entire SHH at once or the latest SH first and the entire SHH in the next step. The S-Sync protocol targets the bandwidth consumption issues by reducing the number of round trips by sending the entire SHH in the first step. However, sending only the latest hash in the first step and then sending the entire SHH, if needed, in the second step consolidates on bandwidth. The intuition for choosing the "latest hash first" variation is to avoid sending the entire SHH when sites are already or almost in sync which, in turn, reduces the traffic over the network.

We know that transferring 20 byte SH will have the same overhead and bandwidth expense as sending a fixed additional number of SHs. Thus, instead of sending the entire SHH or only the latest SH, we found sending up to 'k' entries in SHH to be most efficient, since it may cost the same amount of network delay to send 1 SH entry and 100 SH entries.

In case where the pulling site dominates the source or has a concurrent update, the pulling site will end up sending the entire SHH, after the initial exchange of latest SHs exchange. If the pulling sites send 100 SH entries, the source may be able to find the common ancestor and send back the needed delta to the pulling site. In this case, the entire SHH transfer is not needed. However, in cases where the common ancestor cannot be found using 100 SH entries, the entire SHH transfer is required in order to find the common ancestor and the delta.

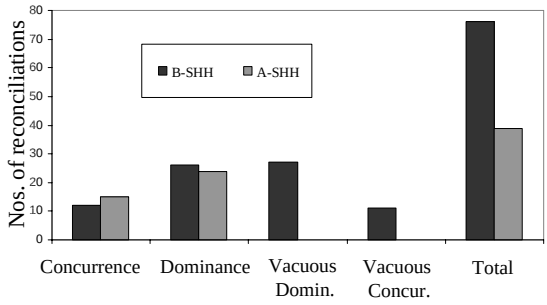

Figure 6: **Effective Reconciliation Steps in S-Sync**. Our experiments show that the B-SHH construction mechanism has to perform 76 effective reconciliations compared to just 39 for A-SHH when converging 4 concurrent updates among 16 replica sites.

5. Evaluation

In this section, we prove that the SHH data structure is capable of providing an efficient and reliable optimistic replication mechanism for distributed applications. Thus, our focus is primarily on testing the fundamental properties of A-SHH. Through our experiments and simulations, we demonstrate that A-SHH:

1. Prevents unbounded replica divergence,
2. Shows an order of magnitude improvement in convergence speed by limiting the number of vacuous/non-data transfer reconciliations,
3. Tolerates divergence among disconnected nodes, yet provides convergence across partitioned networks,
4. Performs comparable to the traditional version vector approach in dealing with multiple concurrent updates.

SHH Implementations: The SHH version tree and other data-structures, used in the development of S-Sync Protocol, are implemented in Java. We, extensively, use Java's Hashtable functionality to maintain the version tree's internal reference and collections. The "dominance information" and "successor-predecessor (parent-child)" relationships are maintained as a key-value pair in a separate hash table. Such a data facilitates traversal during dominance calculation based solely on the latest hash. Further, a pointer is maintained for each Content Delta and its corresponding summary hashes.

Please note that the Content Delta keeps track of changes made to the S-Sync's shared-folder, where any updates to a file or addition/deletion of files are recorded. Whereas the SHH Delta captures the changes in the SHH tree, affected by the updates made to the shared-folder. Therefore, a delta or the change-information is an efficient mechanism to avoid transferring a lot of content across the network [Tr99]. Since, SHH stores the delta information in memory, the SHH Delta and Content Delta are calculated by recursively traversing the Hashtable in a *topologically sorted* [Co01] order as described in Section 2.2.3.

5.1 Evaluation Methodology

We developed the SHH-Simulator to aid in the evaluation and simulation of the SHH protocols. The SHH-Simulator has three components: "Replica Site Selector," "Update Injector," and "Shared Object." Using the Replica Site Selector, a replica site is made to reconcile with another randomly selected site. After reconciliation, the Update Injector is then used to choose and implant new updates at a randomly selected replica site, thus simulating a real world arbitrary order of update among replicas. These randomly generated updates can be clean or concurrent depending on the state of each replica site. We, interchangeably, refer to concurrent update as non-conflicting updates, as these updates are made to different files on different replica sites.

Further, a Shared Object is the unit of replication, for instance the "shared directory" in the case of S-Sync. At predetermined time intervals the reconciling site randomly selects a replica site and exchanges the version information based on the reconciliation scheme.

To aid in consistency across experiment runs, the random order for each site is first logged and then enforced by the SHH-Simulator during all subsequent simulation runs. This is done to aid in validating the results against simulations carried out on a local desktop machine. We found that the results follow the same pattern apart from obvious differences in the reconciliation times due to PlanetLab nodes having varying wide area latencies.

5.2 Update Propagation & Divergence Control

We first show through simulation that while the number of states generated by the B-SHH techniques are unbounded as the number of concurrent updates increases, the performance of SHH and Bayou-VV remain relatively constant. We then demonstrate the efficient update propagation and divergence control properties of A-SHH and compare the variations in SHH protocol construction.

We studied several parameters to understand the S-Sync reconciliation protocols, namely the: (i) reconciliation scenarios (e.g., Dominance case, Latest Hash Equal case, Subset case and Concurrent case), (ii) size of SHH data structure, the number of SHH states being maintained and stored as key-value pairs, and (iii) speed of update convergence among replica sites.

First, to determine the version dominance between the point of update injection and replica convergence, S-Sync protocol transfers either the latest summary hash or the entire hash history. As described earlier, efficiency in update propagation is considered an important aspect of optimistic replication. Therefore, to be network efficient we compare the number of bytes transferred using each of the replication schemes.

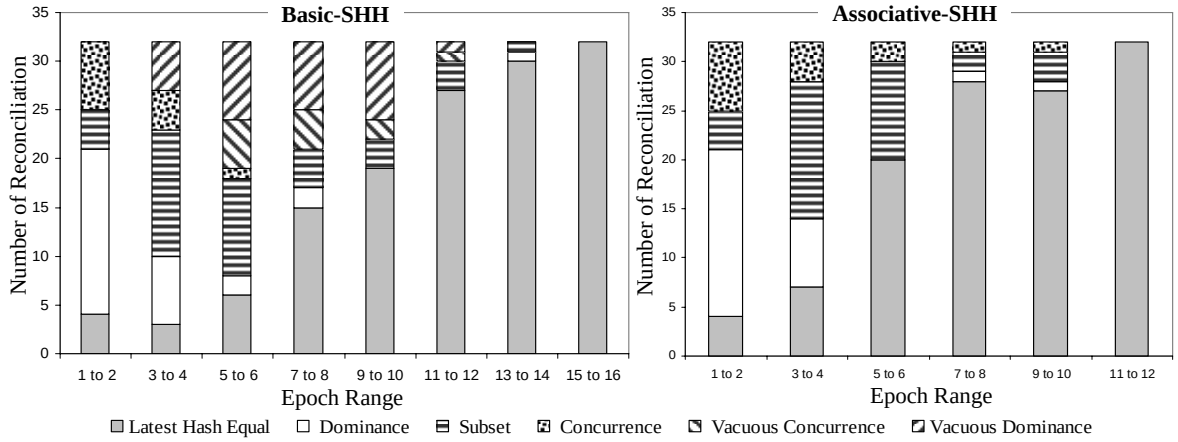

Figure 7: **Reconciliation Scenario in Basic and Associative SHH**. The experiment starts at a quiescent point when all 16 sites are at the same state. At this point, four concurrent updates are injected at epoch time 0. Both the B-SHH and A-SHH reconciliation schemes encounter the same number of dominance and concurrence cases when the sites start pulling the data (i.e., actual exchanges) in the first epoch range (1 to 2) but as the time progresses, B-SHH encounters a lot of vacuous reconciliations and thus takes more time to converge than A-SHH.

The term "reconciliation scheme" is used to refer to the combination of SHH construction and the reconciliation protocol, e.g., B-SHH with entire SHH and A-SHH with entire SHH, as shown in Figure 6. We measure "effective reconciliations", that is, the reconciliation scenarios where sites update their SHH version tree. For experimental purposes we ignore the reconciliation schemes of: latest-hash-equal-case and subset-case, as these schemes only exchange the top hash and no summary hash. As it is, these vacuous hashes do not affect any change in SHH tree.

The results below were obtained using the SHH-Simulator. The simulations were carried out with 16 replica sites and 4 concurrent updates injected on randomly selected sites at the beginning of the simulation. These non-conflicting updates are injected in the system using an Update Injector program. Further, the replica sites are configured to start the random anti-entropy process every 30 seconds. Finally, we experimented with the Basic and Associative SHH implementations, sending entire SHH.

5.2.1 Reconciliation Scenarios

In our study of SHH's properties, we experimented with several reconciliation schemes that exhibited varying degrees of success with respect to scalability and performance. We identified six reconciliation scenarios, in Section 3.1. As discussed, the vacuous cases indicate non-data transferring reconciliations where only the latest summary hashes are exchanged to compute a dominance relationship. The variations in reconciliation schemes stem from the evaluation requirement for the most efficient SHH exchange.

Figure 6 shows B-SHH has to perform 37 vacuous reconciliations and a total of 76 effective reconciliations before it converges, as compared to the 39 effective reconciliation steps required for A-SHH to converge. Obviously, more reconciliation steps translate into higher bandwidth consumption; consequently B-SHH with Entire SHH exchange shows an order of magnitude increase in bandwidth usage.

After experimenting with the summary hash history exchange across 16 sites on PlanetLab, we found that the traditional measure for exchange does not effectively prove the overheads of reconciliation schemes. Thus, to show the reconciliation scenario breakup we use an epoch range. An epoch signifies a cycle where all replica sites in the system complete reconciliation at least once among all their peers. The epoch '1' indicates that all replicas have reconciled at least once. However, this does not mean that the sites were able to resolve all dominance relationship, which might take more steps to accomplish as updates might not have been propagated across the system during the first anti-entropy sweep.

As shown in Figure 7, the experiment starts when all replicas have the same initial state. We then introduced 4 concurrent updates at randomly selected sites from a pool of 16 sites. At the start of the experiment (i.e. epoch range 1 to 2), we observed more dominance and concurrence cases while the sites were busy collecting the actual changes (i.e. content deltas). However, during the epoch range starting from 3 to 12, we notice changes resulting from the two different SHH constructions and attribute them to the manner in which A-SHH and B-SHH constructions resolve version conflicts. We found that the B-SHH with Entire SHH exchange reconciliation-scheme performs a significantly large number of vacuous reconciliations to synchronize the SHH trees, by continuously exchanging SHH Deltas implying that only the summary hashes are exchanged. Note that the SHH Deltas are relatively small in size as compared to content deltas, which

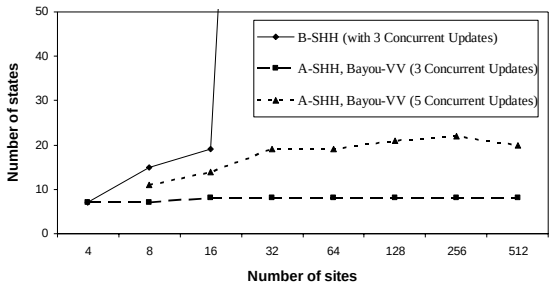

Figure 8: **Comparison of number of SHH States in B-SHH and A-SHH**. This graph compares the number of SHH states generated in B-SHH and A-SHH, in a completely random anti-entropy environment. With an increase in the number of sites, B-SHH tends to reach an infinite upper bound even with just 3 concurrent updates in the system. The A-SHH construction mechanism slowly inches towards its defined upper bound, in this case, 8 states for 3 updates and 32 states for 5 updates.

of course depends on the number and size of updates published at the replica sites.

5.2.2 Number of States and Convergence Issues

In comparing the number of SHH states; (i.e. the number of SHH states in the Hashtable), for the same experiment, we found that the number of SHH states in B-SHH were twice as many compared to A-SHH (18 vs. 9). This difference can be attributed to an important design decision in A-SHH construction, where every new successor state is a set based concatenation of all the causally preceding parents. This ensures that even out-of-order updates and random reconciliations produce the same version identifiers and no intermediate and vacuous states. Thus, A-SHH converges faster than B-SHH and furthermore, guarantees the convergence even across partitioned network. This is contingent on the fact that the updates were transferred before the network partition occurred.

We compare B-SHH and A-SHH in terms of the number of intermediate states generated in a completely random anti-entropy environment. With the increase in number of replicas, B-SHH tends to create an infinite number of intermediate states even with a very small number of updates. For instance, experimenting with only three concurrent updates, B-SHH takes an exponential leap in Figure 8. We attribute this to the fact that every merge in B-SHH creates a new state, and continual reconciliation and subsequent merges create a very high number of such states. A-SHH, on the other hand with three concurrent updates and 16 replicas, reaches a stable upper bound with number of intermediate state remain constant thereafter. This, we believe, is promising because we now know we can guarantee convergence. To further verify, we introduce five concurrent updates, the number of states in A-SHH mechanism slowly inches upwards and subsequently stabilizes when the numbers of simulated sites are increased to 512, as shown in Figure 8.

The convergence properties of SHH is tested and the replica convergence rate is plotted, in Figure 9. Here, we also draw a comparison of SHH with traditional version-vector approaches. The effective reconciliations per site is compared with the number of sites converged. The coordinate (6, 14) represents 14 converged sites with at most six effective Reconciliations. Importantly, some of the sites could have converged in less than six reconciliation steps. For instance, in the case of the A-SHH protocol all 16 sites converge after 7 effective reconciliations.

5.2.3 Reconciliation Protocols and SHs Exchanged

The A-SHH-Latest Hash Protocol reconciliation scheme, in Figure 10, achieves better performance in terms of data-size exchanged. Note that when the A-SHH Entire SHH protocol does not perform any vacuous reconciliation, its performance is closely comparable to B-SHH Entire SHH and B-SHH Latest Hash protocols. This is because each of these reconciliation scheme share the entire SHH version tree per reconciliation.

To determine the version dominance from the point of update injection and replica convergence, in Figure 10 we show the comparison of all four reconciliation schemes in terms of size of data transferred in the first step and the compound steps for the Entire SHH or Latest Hash Protocol respectively. The amount of bytes includes pulling site's latest hash, both the sites IDs and site's entire SHH & RMI callback handle, depending on the protocol variation.

To further improve on the Latest Hash Protocol which sends the entire SHH in the second step, we conjectured that if the sites are not totally out of sync (i.e. the difference in terms of states among their SHH trees is less than a certain threshold value which we refer to as k), sending 'k' SHH states in the second step would be enough to determine the version dominance, thereby avoiding the need for sending the entire SHH on the other side in the second step.

To determine the optimal k size, we attempted sending the varying number of SHH states over PlanetLab with varying latency ranging from 0.2 ms to 92 ms and found that it takes almost the same amount of time to send up to 100 SHH states. Thus, the protocol tweak from Latest Hash then Entire SHH to Optimized Latest Hash Protocol (latest hash then k number of SHs then Entire SHH) appears adequate assuming the sites are not out of sync by more than k SHH states.

5.2.4 Tamper-evident Update History

A tamper-evident update history has attributes that are exploited for update verification and version-roll-back (time travel and undo) capability. The SHH verification time involves a Depth First Search on the SHH tree from the latest hash to the initial state. The time-travel/undo

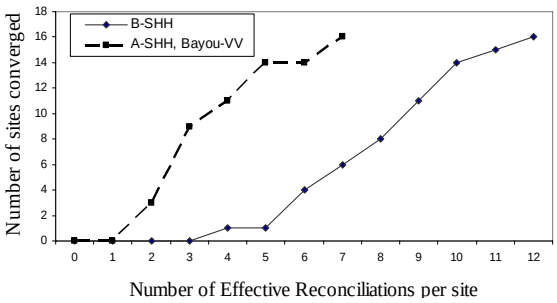

Figure 9: Convergence Comparison: A-SHH, like Bayou-VV, converge faster requiring just seven reconciliations compared to the twelve required by B-SHH.

time involves locating the change-sets by performing a topological sort between the initial and the chosen state. DFS and topological sort times for the Hashtable containing 1000 entries have been found to be less than 100ms.

5.3 RepliWiki Trace Replay on PlanetLab

SHH is used as the underlying mechanism to disseminate and reconcile article updates published to the RepliWiki web application on various PlanetLab nodes. To observe the divergence, efficient update propagation, and convergence benefits in a deployable environment, we conducted a number of experiments that simulated article contributions for a 24 hour period between 4 nodes on PlanetLab.

We obtained the data necessary for this experiment by downloading the entire database dump from Wikipedia's English Language Archive Site [WF06] and extracting a full day's trace that occurred between April 1, 2005 00:00:00 and April 1, 2005 23:59:59. We determined that there were 5614 unique updates which occurred on April 1st, 2005 so we randomly assigned 1516, 1387, 1464, and 1247 to each of the four Planetab nodes. To do this, we created a RepliShuffle program which expects a file containing Planetlab hostnames and distributes them among the Trace table for exporting to each of the Planetlab. We then created the RepliTraceLoader program which replays the trace by simulating users creating updates on each of the four PlanetLab nodes at the time at which the update actually occurred. By default, the RepliTraceLoader program takes a full 24 hours to run; however, we implemented a multiplier feature so that the experiment could be sped up or slowed down while maintaining the time ratio between article updates so as not to skew the experiment.

Our reason for extracting Wikipedia's database into our proprietary format was primarily to eliminate primary and foreign key references which are difficult to manage in a distributed environment. By concentrating only on the relevant aspects to our prototype application (i.e., Title, Author, Hostname, Timestamp, and Description) and eliminating all key references, data can thus be easily exported and imported from the database between nodes.

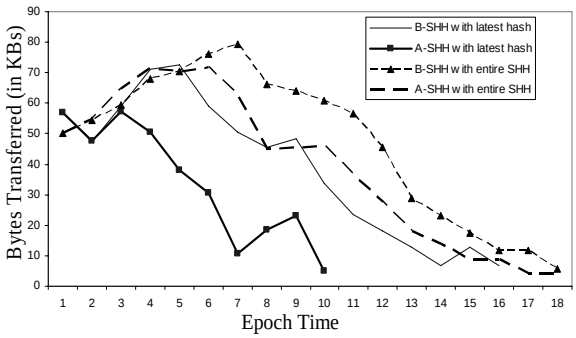

Figure 10: **Comparison of different reconciliation schemes in S-Sync protocol.** The comparison of all four reconciliation schemes in terms of amount of bytes transferred before convergence. The "A-SHH with latest hash first" reconciliation scheme has proven, by far, the most efficient scheme per epoch time.

As discussed, we use S-Sync to disseminate and reconcile Wikipedia article updates between the 16 PlanetLab nodes. Here, the S-Sync protocol is used only to publish the articles to other sites as it is the job of our RepliDriver program to export updates relative to each domain while importing updates exported from other domains and made available via S-Sync. The RepliDriver program is run as a cron job which exports updates in XML files relative to each domain. This cron job also publishes these updates in S-Sync for dissemination, and imports updates exported from other domains that are also disseminated via S-Sync. This cron job was run on each domain periodically throughout the 24 hour experiment. Notably, in RepliWiki, every modification to an article is treated as a new update that is appended to an existing set of modifications, thus the merges of concurrent updates are always associative and commutative. In Figure 11, we show the change in the summary hash history size when the RepliTrace program is run across four nodes from the 24 hour trace. The four sites W, X, Y and Z individually injected concurrent updates into the system every 5 minutes. It shows that the size of summary hash history at each node increases as the sites are merging concurrent updates. Once the concurrent updates are collected at every site, the size stays constant until next updates are introduced in the system. The anti-entropy is carried out every 30 seconds. Every node publishes one update every 5 minutes (i.e., every 10th anti-entropy cycle).

The results in Figure 12 validated our intuition to rethink the design decisions of B-SHH's construction and develop a method that detects and then avoid intermediate states. Hence, by incorporating the a mechanism that keep track of all previous summary hashes, by lexicographically sorting them, we were able to see two order of magnitude difference in the number of states. In this experiment, we deployed S-Sync's shared-folders on 32 PlanetLab nodes and randomly injected concurrent up-

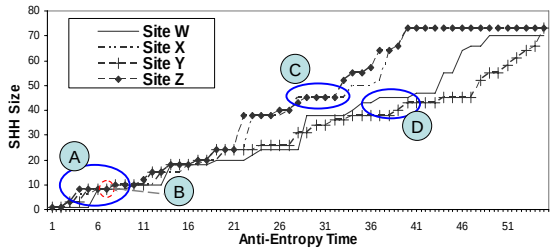

Figure 11: RepliWiki trace replay run 4 planet lab nodes, showing that the size of summary hash history at each node increases as the sites are merging concurrent updates. Once the concurrent updates are collected at every site, the size stays constant until next updates are introduced in the system. Here, A: First concurrent updates from the four sites collected, B: All four sites converge the first concurrent updates, C: Concurrent updates injected, and D: SHH size remains constant till new concurrent updates are injected.

dates. Interestingly, both B-SHH and A-SHH's performance is comparable with just 7 states to reconcile 32 nodes. However, publishing 4 and 5 concurrent updates across the system shows a remarkable increase in the intermediate states. The nodes running the A-SHH variant converged in about 24 minutes, with 30 seconds anti-entropy cycle. We stopped the non-converging B-SHH after the experiment runtime of 24 minutes. Nonetheless, we suspect that convergence would have taken substantially longer time, perhaps in the order of hours.

The size of A-SHH can be different from site to site even if they have the same latest-hash set. For example, a site with the latest hash set, {a,b,c}, can have {a},{b},{c},{a,b},{b,c}, and {a,c} in the history, while another node with the same latest hash set can have only {a}, {b}, {a,b} and {c} in the history. Likewise, Figure 11 shows Site X and Site Z have more merge variants in their A-SHH histories, while Site W and Site Y are enjoying a rather efficient merging path. Note that, A-SHH only collects update or revision histories and *not* merge histories

6. Related Work

Version Vectors are commonly used for tracking update dependencies in optimistic replication [Te95, Sa02, Sa05, Al02, Ki92,Wa83, Gu91,Sa2]. A version vector is a set of counters - one for each replica site in the system [Pa83]. Since version vectors require one entry for each replica site, the size of the version vector grows as the number of replica sites increases. This has the potential to not only limit scalability in terms of the number of nodes that the system can accommodate, but also entails some additional overhead in updating version vectors in the event of site additions/deletions. In order to provide the entry-wise comparison in version vectors, the entries for newly added sites have to be propagated to other replica sites and the deleted sites have to be removed from

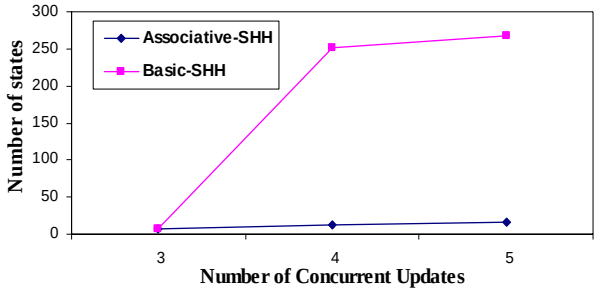

Figure 12: **Comparison of number of SHH States in B-SHH and A-SHH.** This graph compares the number of SHH states generated in B-SHH and A-SHH using random pair-wise update exchanges (i.e., anti-entropy) among 32 Planet Lab nodes. When the number of concurrent updates is 3, the B-SHH was lucky to converge with 7 SH entries. However, when the concurrent updates are 4 and 5, the B-SHH did not converge.

the version vectors [Al02,Pr97,Ra97]. Bayou [Te95,Pe97] provided an elegant solution to this problem by incorporating group membership change information directly in its version vector-based replication protocol. However, an SHH need not be affected at all by site additions or deletions.

Version vectors are vulnerable to various attacks on decentralized ordering because each entry in the version vector summarizes the causal relations to a number, without preserving enough information for others to prove the correctness of the causal ordering histories. Thus, it is easy for a malicious node to propagate incorrect information to all replicas by falsifying the decentralized ordering state. Another drawback to version vectors concerns accidental loss of data. For example, when the historical ordering of versions is altered, any attempts to undo corrupted updates may "restore" versions based on corrupt data or cause an excessive number of valid updates to be discarded [Ma02, Re95, Sm94, Sp99].

Spreitzer *et al.* [Sp99] designed a countermeasure to deal with server corruption in **Bayou**. Like S-Sync, the proposed solution offers both efficient update propagation and tamper-evident update histories. The local update is signed by the replica site that initiates the update to ensure that its contents cannot be altered by corrupt servers. The server that accepts the update then cryptographically chains it to the previous update accepted by the same server to prevent other servers from reordering and dropping updates. These chains are interleaved into each replica's update log and update origins can be traced by traversing the chains. Bayou requires per-site version vectors to figure out missing updates and cryptographic chaining to provide tamper-evident access to its update logs, as well as a dependency-check mechanism for conflict detection. S-Sync uses a per-site SHH for all of these functions, freeing itself from the version vector maintenance, from potentially complex log merging, and from specialized conflict management. Also, while the

Bayou paper presented only a paper design of the suggested countermeasures, S-Sync using SHHs has been implemented and has been running on Planet-Lab with a real RepliWiki application.

Both **Bayou** [Te95, Pe97] and the current implementation of S-Sync are operation-transfer systems that maintain and transmit records of update operations. The current design of S-Sync keeps one SHH per site with pointers into a locally stored change-set. S-Sync could be used as a state-transfer mechanism, as in WinFS [Ma05d], by keeping an SHH for each file.

PRACTI, unlike some other systems, including Bayou and S-Sync, sends invalidation messages between replicas when objects are updated. The updated contents are then fetched independently, for instance, when attempting to read an invalid object or by using a hoarding program. Since PRACTI uses version vectors similar to Bayou, SHH can provide security benefits to PRACTI. Likewise, S-Sync could employ PRACTI's efficient invalidation mechanism in its reconciliation schemes.

In the **causal history** approach [Sc94], each site maintains the preceding causal history (i.e., all the versions that the site has received or created). During reconciliation, sites exchange their latest version and its causal history, from which each site can check whether or not one version appears in the other's causal history (e.g., Version X is a revision of version Y if Y appears in X's causal history). Since the causal history does not require one entry for each replica site as in version vector, causal history approaches in general readily support dynamic membership changes. Unfortunately, since the size of the causal history grows in proportion to update instances in the system, the network bandwidth can be saturated by the continual exchange of unbounded causal histories.

Although the concept of causal history is not new [Sc94], this paper presents a novel, practical construction that meets applications' security needs and relates our experiences evaluating a real implementation. Our previous work on **Hash History** [Ka03] produced a causal history implementation which used the cryptographic hash of version content as a version identifier to determine dominance for state-transfer reconciliation. Unlike S-Sync in this paper, this previous design required a hash history for every shared object. This mechanism also used an epoch number (a counter maintained at each site for each version) to distinguish the current version from the previous versions with the same content. Unfortunately, this epoch number may or may not provide ordering correctness for some applications. For some applications, if each site independently produces the same content from different previous versions, the resulting version should not be marked identical. We note that, with the hash-epoch scheme, this case can happen. In addition, the epoch numbering may introduce complicated log management overhead, which can make it difficult to prune old histories to control unbounded storage. Finally, and perhaps most importantly, the hash-epoch scheme does not maintain tamper-evident history linking.

Pastwatch [Yi06] is a version control system that allows pair-wise reconciliation without a central server. Pastwatch's revtrees are similar to B-SHH's version tree structure since the version identifier includes the parent versions' identifier. It assigns two different identifiers to the new merged versions that are independently created from the same base versions, even when the resulting merges are the same (e.g., (AB)C = A(BC)). This may be a desirable characteristic for a version control system, but, as we discovered with B-SHH, may not be suitable for optimistic replication with random anti-entropy. S-Sync with A-SHH guarantees eventual convergence by using continuous random anti-entropy and assigning the same version (state) identifier to independent associative merges with the same predecessor versions.

BitKeeper [Bi06] also provides decentralized version exchanges among peers. Its versioning structure is similar to B-SHH and hence suffers from the same problems. Their web site recommends hierarchical reconciliation schedule to avoid divergence

LOCKSS [Ma05p] replicates published articles in order to preserve their content, such as if the publisher goes out of business. Unlike SHH, it does not use encryption, and it is not concerned with the process by which articles are updated or distributed. LOCKSS focuses on determining whether an article's contents have been corrupted. To check this, replicas periodically run a sampled poll to compare their stored contents and vote on the correct contents if differences are observed. S-Sync and LOCKSS differ in their application semantics but share some common motivations. We believe that these two schemes could work together to provide long-term preservation with update accountability.

7. Conclusion

A Summary Hash History (SHH) is a novel, practical method for maintaining a secure version tree in systems that employ optimistic replication. The use of collision-resistant summary hashes as version identifiers provides source traceability of updates and an undo/time-travel capability in a tamper-evident manner. S-Sync uses the SHH data structure to extract the set of incremental updates to be transferred between sites during pair-wise reconciliation without using version vectors, freeing itself from the version vector maintenance issues. By using summary hashes as pointers into change-sets, S-Sync does not need to keep operation logs, thereby avoiding log merging procedures.

In retrospect, the effort to build SHH-Visualizer was quite worthwhile. SHH-Visualizer discovered that S-Sync with B-SHH may not be suitable for optimistic replication with random anti-entropy and motivated our next

design: A-SHH. S-Sync with A-SHH not only guarantees eventual convergence but also enables the convergence of concurrent updates (sometimes across partitioned networks) by assigning the same state identifier to independent associative merges with the same predecessor states.

Finally, http://isr.uncc.edu/SHH provides S-Sync, SHH-Visualizer, and links to the RepliWiki nodes deployed on PlanetLab to demonstrate the usefulness of the SHH technique in a real world application.

References

[Al02] Almeida P. Baquero C., and Fonte V., Version Stamps-decentralized version vectors. In Proc. of ICDCS, 2002

[Ba92] Bayer D., Haber S., and Stornetta W. Improving the efficiency and reliability of digital time-stamping, Methods in Communication, Security, and Computer Science, 1992

[Be06]Belaramani N., Dahlin M., Gao L., Nayate A., Venkataramani A., Yalagandula P., and Zheng J., PRACTI Replication, In Proc. of NSDI, 2006

[Bi06] www.bitkeeper.com/UG/Advanced.Branching.How.html

[Co01]Cormen T., Leiserson, C., Rivest R, Stein C. Introduction to Algorithms, Second Edition, McGraw-Hill Book Company, http://mitpress.mit.edu/algorithms/, 2001

[De87] Demers A., Greene D., Hauser C., Irish W., Larson J., Shenker S., Sturgis H., Swinehart D., and Terry D., Epidemic Algorithms for Replicated Database Maintenance, PODC, 1987

[Go92] Golding R., Weak-consistency group communication and membership. PhD thesis, Tech. Report, UCSC-CRL-92-52, 1992

[Gu91] Guy R., Ficus: A Very Large Scale Reliable Distributed File System. PhD thesis, UCLA, 1991

[Ha91] Haber S. and Stornetta W., How to time-stamp a digital document, Journal of Cryptology, 1991

[Ka88] Kawell, L., JR., Beckhart, S.,Halvorsen, T.,Ozzie, R., and Greif, I. Replicated document management in a group communication system., In CSCW. Chapel Hill, NC, 1988

[Ka03] Kang B., Wilensky R., and Kubiatowicz J. The hash history approach for reconciling mutual inconsistency, In Proceedings of ICDCS, 2003

[Ki92] Kistler J. J. and Satyanarayanan M., Disconnected operation in the coda file system. ACM Trans. on Computing Systems, 1992

[Ku00] Kubiatowicz J., Bindel D., Chen Y., Eaton P., Geels D., Gummadi R. Rhea S., Weatherspoon H., Weimer W., Wells C., and Zhao B. OceanStore: An Architecture for Global-scale Persistent Storage, In Proc. of ASPLOS, 2000

[Ma02] Maniatis P. and Baker M., Secure History Preservation Through Timeline Entanglement. In Proc. of Usenix Security Symposium, 2002

[Ma05p] Maniatis P.,Roussopoulos M., Giuli T., Rosenthal D., Baker M., The LOCKSS - Peer-to-peer digital preservation system. ACM Trans. on Computing Systems, 2005

[Ma05d] Malkhi D. and Terry D., Concise Version Vectors in WinFS. In Proc. of DISC, 2005

[Me87] Merkle R., A digital signature based on a conventional encryption function. In C. Pomerance, editor, Crypto'87, 1987

[Mu95] Mummert, L. B.,Ebling,M. R., and Satyanarayanan,M. Exploiting weak connectivity for mobile file access, In 15th SOSP, Copper Mountain, CO. 143–155., 1995

[Pa83] Parker Jr.D., Popek G., Rudisin G., Stoughton A., Walker B., Walton E., Chow J., Edwards D., Kiser S., and Kline C., Detection of Mutual Inconsistency in Distributed Systems, Trans. on Software Engineering, 1983

[Pe97] Petersen K., Spreitzer M., Terry D., Theimer M., and Demers D., Flexible update propagation for weakly consistent replication. In Proc. of SOSP, 1997

[Pr97] Prakash R. and Singhal M., Dependency sequences and hierarchical clocks: efficient alternatives to vector clocks for mobile computing systems, Wireless Networks, 1997

[Ra97] Ratner D., Reiher P., and Popek J. Dynamic version vector maintenance, Tech. Report CSD-970022, UCLA, 1997

[Ra98] Ratner, D. H. Roam: A scalable replication system for mobile and distributed computing, Ph.D. thesis, Tech. Report. No. UCLA-CSD-970044, UCLA, Los Angeles, CA, 1998

[Re95] Reiter M. and Gong L. Securing causal relationships in distributed systems, The Computer Journal, 1995

[Sa00] Saito Y. and Levy M, Optimistic Replication for Internet Data Services. In Proc. of DISC, 2000

[Sa02] Saito Y., Unilateral version vector pruning using loosely synchronized clocks, Technical Report HPL-2002, 2002

[Sa05]Saito Y. and Shapiro M., Optimistic Replication, ACM Computing Surveys, vol. 37, March 2005

[Sc94] Schwarz R. and Mattern F. Detecting causal relationships in distributed computations: In search of the holy grail, Distributed Computing, 1994

[Se06]Seigenthaler J., Seigenthaler and Wikipedia-Lessons and Questions: A Case Study on the Veracity of the "Wiki" concept www.journalism.org/resources/research/reports/Wikipedia/defaultwiki

[Sm94] Smith S. and Tygar J. Security and privacy for partial order time, In ISCA International Conference on Parallel and Distributed Computing Systems, 1994

[Sp99] Spreitzer M., Theimer M., Petersen K., Demers A., and Terry D. Dealing with server corruption in weakly consistent replicated data systems,Mobile Computing & Networking, 1997

[Te95] Terry D., Theimer M., Petersen K., Demers A., Spreitzer M., and Hauser C., Managing update conflicts in Bayou, a weakly connected replicated storage system. SOSP, 1995

[Tr99]Tridgell, A. Efficient Algorithms for Sorting and Synchronization, PhD thesis, Australian National University, 1999.

[Wa83] Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. The Locus distributed operating system. SOSP, 1983

[WF06]"Database Dump Progress." *Wikipedia: The Free Encyclopedia*. Wikipedia Foundation, 2006

[WK06] Millionth English article posted on Wikipedia, IDG News, www.networkworld.com/news/2006/030206-wikipedia-million.html

[Yi06] Yip A., Chen B., and Morris R., Pastwatch: A Distributed Version Control System, In Proc. of NSDI, 2006