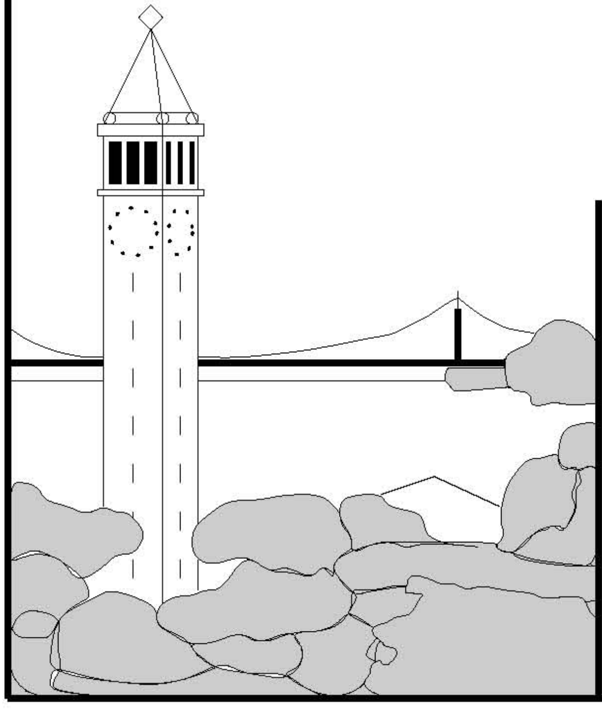
S2D2: A Framework for Scalable and Secure Optimistic Replication

Brent ByungHoon Kang
University of California, Berkeley



Report No. UCB/CSD-04-1351

October 2004
Computer Science Division
University of California
Berkeley, California 94720

S2D2: A Framework for Scalable and Secure Optimistic Replication

by

Brent ByungHoon Kang

B.S. (Seoul National University) 1993
M.S. (University of Maryland at College Park) 1995

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Fall 2004

The dissertation of Brent ByungHoon Kang is approved:

Chair	Date
	Date
	Date
	Date

University of California at Berkeley

Fall 2004

S2D2: A Framework for Scalable and Secure Optimistic Replication

Abstract

S2D2: A Framework for Scalable and Secure Optimistic Replication

by

Brent ByungHoon Kang

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Robert Wilensky, Chair

High data availability and scalability beyond a single shared server require data to be replicated and exchanged in a decentralized way. Optimistic replication is the primary technique to achieve these goals; however, current approaches to optimistic replication have fundamental limitations. Version vector based approaches entail complicated management in site addition/deletion and have limited scalability in terms of the number of replica sites. Moreover, version vectors entail significant overhead in maintaining revision histories that are essentially required to deter various attacks on decentralized ordering correctness. Because of the cooperative nature of decentralized dependency tracking mechanisms, a malicious site can easily falsify ordering information, which may cause the shared state to diverge without being detected.

This thesis presents S2D2, a framework that provides optimistic replication based on a novel decentralized ordering mechanism, called Summary Hash History (SHH). Being based on a causal history approach with secure summary hashes as version identifiers, SHH supports simple management of site membership changes, scales regardless of the number of sites, and guarantees the correctness of decentralized ordering in a scalable way. SHH uses "two-step reconciliation" to overcome the inherent limitation of the causal history approach, and thus, consumes orders of magnitude lower bandwidth than reconciliation based on version vectors. Interestingly, SHH provides faster convergence than version vector based approaches by recognizing "coincidental equalities," cases when identical versions are produced independently at different sites. This is of significant value in that SHH can enable distributed replica to converge even in the network partitions or disconnections that mobile computing and wide-area distributed computing have to cope with fundamentally.

S2D2 employs an elegant "hash typing" mechanism to enforce correctness in the error-prone usage of hashes and uses an "adaptor architecture" to support the application-specific consistency requirements. Prototype implementations of adaptors, for a peer-to-peer CVS (a concurrent version control system) and a replicated-shared folder, demonstrate that the S2D2 framework is highly suitable for supporting secure optimistic replication among global-scale and pervasive applications.

Professor Robert Wilensky
Dissertation Committee Chair

To my beloved wife Katherine JungYun, my lovely son Joshua HanKyul,

and my parents.

Contents

List of Figures

List of Tables

Acknowledgements

Finally, I want to thank the many people who have provided invaluable help to me during my Ph.D. program at Berkeley. Appreciating other's help always gives me a heart-warming and enjoyable moment.

First, I'd like to thank my advisor and mentor, Professor Wilensky. I highly appreciate his generous support from the moment I started my Ph.D. program at Berkeley until this dissertation. He believed in me when I was a mere student who was not known. He taught me how to think through a problem as a researcher and showed me how to present ideas effectively. His brilliant intellect always put me back on track when I was stuck with a problem that seemingly did not have a solution or when I was jumping from one idea to another without purpose. He had an amazing ability for extracting clear concepts from my vague descriptions of unformulated ideas. I thank Professor Wilensky for also being a mentor to me. He gave me lots of encouragement, as well as taught me his optimism in professional relationship with others. I truly enjoyed our many discussion meetings.

I was fortunate to have Professor Kubiatowicz as my co-advisor. His keen and visionary insight in system's design has helped me to focus. When we first designed hash history to address scalability issue, he strongly suggested that I research security issues more, because he felt that hash history design might have an interesting security story. He was very right. After a year of effort, the Summary Hash History came out and became the central foundation of this dissertation. I also appreciate that he provided me with many good opportunities to meet other systems researchers.

Professor Brewer generously provided me with his guidance throughout my Ph.D.

study. His impressive insight in systems research was instrumental in forming Summary Hash History idea. When I first presented a fledgling Hash History idea at a work-in-progress session at the 18th SOSP conference, he was there in the audience and later gave me an insightful feedback and encouragement, which I highly appreciate.

Professor Chuang helped me as an outside department committee member and shared with me his invaluable insight and time. He was always kind and encouraging with his big smile, which I'd remember for a long time. I remember that he first suggested that I use CVS log as a multi-writer trace for simulation.

Jon Traupman, my lunch buddy and office mate, always encouraged me and helped me by proof-reading various research documents.

Byung-Gon Chun helped me to run SHH based reconciliation simulation, and I especially enjoyed our discussions regarding various systems research issues.

Alex Sim, who is a researcher at Lawrence Berkeley National Lab, helped me from the very early years of my Ph.D. program as my mentor. He always gave me an objective perspective on my various research ideas.

I'd like to thank folks in the Digital Library Project, including Tom, Taku, Jeff, Phil, Ginger, Joyce, Jon and Loretta. They have been good friends and provided me with helpful feedbacks on my early research ideas. Especially, Jeff and Phil were eager to help me in setting up machines for simulation.

Of course, I cannot forget folks in the OceanStore Project, including Patrick, Dennis, Ben, Hakim, Steve, Emil, Sean, Rachel, Gon and Kris, they all shared their insights and provided helpful feedbacks during numerous practice talks. They are all good colleagues.

I'd like to thank the administrative staffs at EECS Dept. at Berkeley. Especially I'd like to thank Mary Byrnes who helped me with scholarship support and encouraged me with her cheerful and optimistic attitude. Peggy Lau and La Shana Porlaris were always available when I needed their help. Winnie Wang, Jane Doyle, and Laneida Lakagawa have always been helpful with numerous administrative matters. I highly appreciate their kind willingness to help me.

I'd like to thank our funding agency, the National Science Foundation, which supported this research through the Berkeley Digital Library project and the Career Award program for Prof. Kubiatowicz.

Finally, I'd like to thank my parents, my lovely wife and my son, who gave me unwavering support throughout my Ph.D. program. Joshua, my 4 year old, always enjoyed playing with the giant abacus on the 4th floor of Soda Hall. Many times, he came by to deliver my dinner meal and encouraged me by saying "Daddy, write a good paper, Bye !!!"

Chapter 1

Introduction

Optimistic replication is a fundamental technique that is used to achieve high data availability and performance in distributed systems. This chapter provides an overview of optimistic replication and characterizes its limitations with respect to the scalability and security requirements of current dynamic, peer-to-peer and global-scale data sharing applications. It then introduces our proposed solution, the S2D2 framework, based on a novel summary hash history approach.

1.1 Optimistic Replication

Sharing mutable data and exchanging information updates are fundamental aspects of human collaboration, such as document co-authoring, collaborative information gathering, and team development of source code.

Central data servers have been widely used, for example, to support web-based file sharing, shared directories on file servers, and version control systems. However, relying on a central shared server has a number of shortcomings.

First, the central server can be a single point of failure. If the central server is not available, data sharing is not possible. If a network-partition happens between the shared server and client-sites, the client sites cannot exchange updates, although they are reachable among themselves.

Second, dependency on a single central server cannot provide the autonomy that may be required for collaborative tasks across administrative domains. For example, in collaboration between two organizations, each organization may want to store up-to-date copies on its own server for ownership and security reasons rather than depend on someone else's data server.

Instead of depending on a centralized data server for every read and write access, data can be replicated at various points in the network to provide high data availability and good performance to users. Users can then access a locally available replica server. However, concurrent modifications can potentially happen with replica servers, so the servers need to ensure a consistent view of data modifications. There are two well-known approaches to maintaining consistent data modification.

First, a pessimistic approach would require a globally exclusive lock to be obtained before every update operation, preventing the other users from writing until the lock can be returned. This approach prevents concurrent modifications from occurring. However, this strategy entails complicated lock management and severely limits data availability in the case that the replica site with an exclusive lock is not reachable for an extended period.

An alternative approach is optimistic replication (also known as weakly-consistent replication). Optimistic replication allows the replica sites to be available for users to modify data without locking control. The optimistic replication approach assumes that concurrent modifications are rare and that they can be dealt with effectively after the fact.

Under this approach, replica sites need to track dependency orderings among distributed updates to detect concurrent modifications. In the event of concurrent modifications, the optimistic replication approach needs to provide reconciliation procedures to resolve conflicting updates. To guarantee eventual consistency, updates are typically propagated through epidemic disseminations [3] such as anti-entropy [1], 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 sites need to reconcile the conflicting updates to restore a consistent view of the data modifications. 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.

1.2 Limitations of Previous Approaches to Optimistic Replication

Version vectors [12, 43] have been used in reconciling replicas for detecting conflicts [12, 39, 38, 34] and for determining the exact set of deltas (incremental updates) to be exchanged for reconciliation [35, 54]. In version vector based approaches, each replica site

[1]Anti-entropy is a reconciliation mechanism in which each site periodically reconciles with a randomly chosen site.

maintains a vector of entries to track updates generated by other sites. Dominance relations between distributed replicas are determined by comparing the vector entries.

Unfortunately, optimistic replication approaches based on version vectors have fundamental limitations with respect to scalability and security.

First, version vectors entail complicated management in site addition/deletion and have limited scalability in terms of the number of replica sites. Since version vectors require one entry for each replica site, the size of version vector grows as the number of replica sites increases. The management of version vectors becomes complicated as entries for newly added sites are propagated to other replica sites [4, 35, 39, 37]. This problem is especially pronounced in pervasive and ubiquitous computing environments where hundreds of diverse devices and computers need to exchange updates in a pervasive and secure way.

For example, in a global-scale peer-to-peer sharing of mutable data, it is reasonable to assume a given object has tens of writers (authors) and thousands of readers. However, with dynamic version vector mechanisms, a writer membership change must be propagated to not only to the writers but also the readers, because the readers must also be able to determine the dominance relations among updates from multiple writers. Depending on when the membership change information arrives, a reader may receive two updates with vectors of different sizes, which entails complicated examination of orderings among membership change histories and modification histories.

Second, version vectors entail burdensome overhead in maintaining revision histories that are essentially required to deter various attacks on decentralized ordering correctness. Because of the cooperative nature of optimistic dependency tracking, it is easy for a malicious node to propagate incorrect information to all replicas by falsifying the decentralized ordering state [47, 48, 40, 46, 28].

For example, a malicious site can save the signed entries of recent version vectors and use them as version vectors for phony updates (i.e., a “same version id” attack). This attack is particularly pernicious because the attacker can partition the shared object into two divergent versions and prevent some sites from ever receiving the original update. The victim may never know its original version has been attacked and may not be able to counteract. If there are no more new updates, the state will stay diverged forever. If there is a new update, it may innocently be based on the bogus version, and overwrite the original victim’s update. In this case, the victim’s update is lost forever without being detected at all.

Thus, it is desirable that a dependency tracking mechanism enforces the correctness of the protocol that a writer should follow in assigning correct ordering among updates. By ensuring decentralized ordering correctness, the optimistic replication system can guarantee that updates are not vulnerable to a decentralized ordering attack. With this guarantee, each site can focus on validating the quality of data contents in a cooperative way. Whether the content of an update is good or bad, the update will be delivered in correctly formed ordering. The version vector mechanism and its limitations are discussed further in Chapter 3.

1.3 A Proposed Solution: S2D2 based on Summary Hash History

To address such limitations, we have developed a novel decentralized ordering mechanism, called Summary Hash History (SHH), which is based on a causal history approach [43]. Using summary hashes as version identifiers, SHH provides simple management in site membership changes, scales regardless of the number of sites, and guarantees the correctness of decentralized ordering in a scalable way.

Per the causal history approach, the size of a SHH that needs to be exchanged grows in proportion to the number of update instances. To effectively manage this overhead, "two-step SHH reconciliation" is used. Only the latest summary hashes in SHHs are exchanged frequently; the data/SHHs can be lazily pulled from any local site, with the latest summary hash verifying the associated data and SHH. Simulations show that the two-step SHH reconciliation consumes orders of magnitude lower bandwidth than reconciliation based on version vectors. Interestingly, the simulation also revealed that SHH provides faster convergence by recognizing "coincidental equalities," cases when identical versions are produced independently at different sites. This is of significant value in that SHH enables distributed replicas to converge even in the partitioned networks that mobile computing and wide-area distributed computing need to cope with. SHH is described in detail in Chapter 5.

To make an impact on the practice of computing, SHH needs to be readily available to various applications that rely on optimistic replication. To this end, S2D2 framework is developed. S2D2 provides global-scale applications with support for optimistic data replications and robust update exchanges. The S2D2 framework employs an "adaptor architecture" to support the diverse, application-specific consistency requirements. This architecture can also support legacy applications. S2D2 provides SHH and its two-step reconciliation protocol for exchanging decentralized updates in a scalable and secure way.

To enforce correctness in the error-prone usage of hashes, the "hash typing" mechanism is presented. A hash is annotated with a hash-type as a data is annotated with a data-type. The framework also uses "hash typing" as a secure way to communicate an object's structural information between adaptors and S2D2 sub-components such as the two-step reconciliation process. To show how one can build an application specific adaptor, I built a few example adaptors for useful applications, including a peer-to-peer CVS and a replicated shared folder. Experience in building these prototypes demonstrated that S2D2 provides simple interfaces for diverse global-scale applications that rely on secure and scalable optimistic replication.

1.4 Contributions

A central contribution of this thesis is the design and implementation of Summary Hash History, a novel decentralized ordering mechanism that enables each participating site in optimistic replication to verify in a scalable way the correctness of decentralized ordering. This thesis presents SHH's two-step reconciliation that practically manages the inherent overhead of causal-history based approaches. It also details how SHH can be used to provide convergence even during network partitions and disconnections.

A second central contribution is the S2D2 framework. S2D2 is based on SHH to provide a promising basis for building a global-scale federated infrastructure to support collaborative information management, pervasive computing, and peer-to-peer systems. It shows that the S2D2 framework can enable traditional applications as well as new non-traditional applications, such as “self-administering data”, which can change the way people interact with shared data.

1.5 Thesis Map

The remainder of this dissertation is as follows: Chapter 2 describes the desirable properties of an ideal optimistic replication systems and its evaluation method based on simulation. Chapter 3 introduces background concepts of the causal history and the version vector as base mechanisms for optimistic replication. Then it characterizes the challenges and limitations in previous approaches. Chapter 4 describes the Hash History (HH) approach, a predecessor of the SHH approach, that was designed to overcome the scalability limitation of the version vector approach, and then Chapter 5 proposes the Summary Hash History (SHH) approach that provides the ordering correctness guarantee that both the version vector and HH approaches lack. Chapter 5 describes various SHH based network protocols such as the two-step SHH reconciliation. Chapter 7 presents the S2D2 framework that provides global-scale applications with support for secure and scalable optimistic replications based on SHH, and describes various applications that can be built on top of the S2D2 framework. Chapter 8 discusses a new data management model, called “self-administering data”, that can be enabled by the S2D2 framework. Finally, Chapter 9 concludes.

Part I

Motivation

Chapter 2

Scalable and Secure Optimistic Replication

This chapter presents a list of desirable properties of an ideal optimistic replication system in terms of scalability and security. It describes some example applications that need a scalable reconciliation mechanism for exchanging distributed updates and some applications that require a secure mechanism to deter various attacks on the ordering among distributed updates. It then describes how one can evaluate an ideal optimistic replication system and its reconciliation protocols using a trace driven simulation.

2.1 Applications of Scalable and Secure Optimistic Replication

Here we describe some example applications with a need for a scalable and secure optimistic replication.

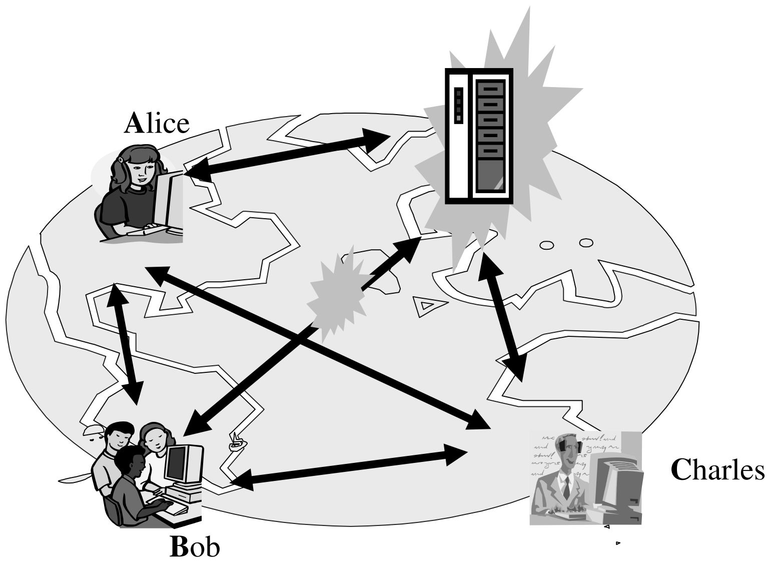


Figure 2.1: Optimistic Replication Masking a Single Point of Server Failure: In this example, three computers (Alice's, Bob's and Charles's) exchange updates directly without depending on a central data server. Thus, they can keep exchange updates even when the central data server is not available or connection to the server is disconnected.

2.1.1 Optimistic Replication for Pervasive/Ubiquitous Data Management

Traditionally, optimistic replication is performed using a few servers that are mostly available. Membership changes are not common and the number of replica is typically in the orders of tens. Figure 2.1 shows an example of three participants using optimistic replication to achieve high data availability by masking the single point of the server failure with direct peer-to-peer data exchange.

However, in the era of ubiquitous and pervasive computing, an update needs to be replicated and propagated to many diverse devices and computers. A user may need to replicate a shared object in her office computer, home machine, PDA device, and colleague's computer. For example, an address object (i.e., address book entry) can be replicated in friends', co-workers' and families's address books.

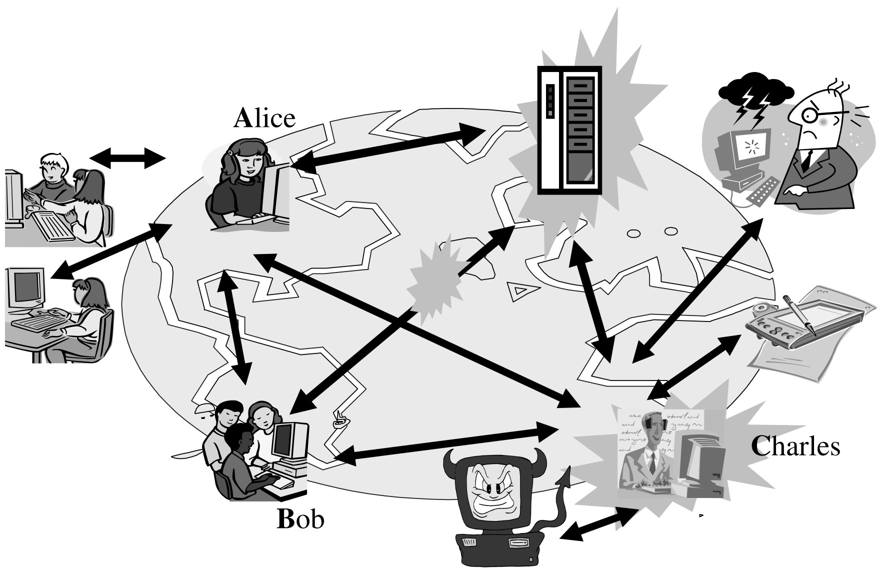


Figure 2.2: Secure Optimistic Replication for Pervasive/Ubiquitous Devices: In the era of ubiquitous and pervasive computing, the shared object often needs to be replicated in many diverse devices and computers. Moreover, if one device/machine is compromised or stolen, the other collaborating replica sites become vulnerable to various security attacks.

2.1.2 Optimistic Replication for Global Scale Data Replication

In practice, optimistic replication has also been used to replicate mutable objects among globally distributed servers for geographically distributed corporate offices and branches. Active Directory by Microsoft, for example, currently uses optimistic replication to exchange updates among thousands of Active Directory sites to support a global scale company. Such large scale data management may require a scalable reconciliation protocol that does not saturate the network to reconcile thousands of replicas. Given that the network bandwidth is still considered an expensive resource compared with other resource such as disk and CPU, it is likely that a global scale corporation is concerned about the volume of network bandwidth that is consumed for exchanging updates among thousands of corporate servers around the globe.

Another example is optimistically replicating Public Key Revocation Lists (PKRL). A PKRL contains public keys that have been revoked. A PKRL needs to be checked during every security procedure (e.g., verification and signing) based on the public key mechanism. Obviously, such a PKRL needs to be replicated into many servers around the globe and newly revoked keys must be exchanged in a secure and scalable manner. Otherwise, for example, an adversary could prevent the PKRL at some sites from receiving an announcement of the revoked key [1].

2.1.3 Optimistic Replication for Open/Closed Collaboration

The following example is a useful scientific data sharing application that enables collaboration among scientists. Scientific data such as climate data and protein sequences are extremely large and read-only data sets. After evaluating or processing the raw data, scientists can annotate the raw data using a data-specific visualization tool with annotation support. For example, a protein visualizer provides a data-specific annotation tool with which users can annotate the rendered protein image. Some approaches to annotation, such as that of Multivalent Documents [36], permit annotations to be kept separate from the documents they annotate. However, collaboratively sharing such annotations across administrative domains is difficult. Unlike the raw data, annotations are read-write in the sense that annotations can be annotated, thus require coherent ordering among decentralized annotations. Currently, as a commonly available data sharing tool, scientists can use e-mail attachments, which are cumbersome, error-prone and cannot support bulky data.

[1]Section 3.3 will identify this attack as a *same version ID* attack. A bogus update can be associated with the same ID of an original update, so that the site that receives bogus update will not be able to receive original update later.

Using a central data server is vulnerable to the server failure and cannot readily scale beyond a single administrative domain.

Here is another example that can benefit from optimistic replication to achieve high data availability with loose synchronization. Suppose a number of experts maintain their own FAQ document about a common topic (e.g., Linux administration) at their web sites. Each expert updates his or her own FAQ. Hence, there are many different versions of FAQs extant. It would be desirable to have a merged version so that users can access the most up-to-date and complete FAQ from any web site to which they can connect. Since a centralized document management server may require administrative commitments that none of the experts may be able to make, they need to share a mutable FAQ in a loosely synchronized fashion across multiple administrative domains. Optimistic replication that provides loose synchronization of FAQ sites using pair-wise reconciliations would be desirable in this application.

The open nature of this type of peer-to-peer mutable data sharing tends to have dynamic replica membership – the shared object needs to be replicated dynamically at various peer sites as needed. It is impossible to have fixed number of replicas; however, the most common tracking method (e.g., static version vectors) may require a static set of replica identifiers to track dependencies among updates. Thus, to take an effect of the membership change, the entire replication system may need to be put on hold to reconfigure the new membership change into the dependency ordering mechanism.

2.2 Reconciliation Process of Exchanging Updates in Optimistic Replication

In order to understand the potential difficulties in designing a scalable and secure optimistic replication system, here, we describe how a general reconciliation procedure may exchange independently accrued updates among replica sites. To exchange updates, each site first needs to exchange version ordering information instead of exchanging entire version content. (More detailed description can be found in Chapter 3.) Based on this ordering information, each site can determine which version *dominates* the other or if they are in conflict, with neither version dominating the other. We say X dominates Y iff X is a revision of Y. The dominating version's update needs to be sent over to the other pulling site. Figure 2.3 shows a pseudo code for basic pull reconciliation between Site A and Site B. Site A requests an ordering information from Site B to determine the dominance among updates and to pull the data from Site B according to the dominance determination.

Thus, a decentralized ordering mechanism that tracks the ordering information among distributed updates is the central foundation in designing a scalable and secure optimistic replications technology to support the various applications that we described above. For example, the ordering mechanism should scale as the number of replica sites increases; the size of the data structure for maintaining ordering mechanism should not become unbounded.

Decentralized ordering mechanism typically requires cooperative management among replicas. Each replica should modify the ordering information among updates according to the protocol. Unfortunately, because of this cooperative nature of decentralized depen-

```
BasicPullReconciliation (ObjectID objID, Site SiteB)  at SiteA {
//From SiteA with orderInfoA as ordering information of the latest version
 1.  Make connection to SiteB with objID,
 2.  orderInfoB ← RequestOrderInfo(SiteB, objID)
 3.  If (orderInfoA = = orderInfoB) { Case 0: Stop. }
 4.  Else if  (orderInfoA  dominates orderInfoB) { Case 1: Stop. //A dominates B}
 5.  Else if  (orderInfoB dominates orderInfoA) { Case 2:  // B dominates A
 6.            d1 ← RequestDelta(SiteB, objID, orderInfoA , orderInfoB)
 7.            ApplyDelta(objID, d1)  // apply the delta to the shared object
     }
     Else { Case 3: //Neither dominates the other
 8.            ca ← FindCommonAncestor(orderInfoA , orderInfoB )
 9.            d1 ← RequestDelta(SiteB, objID, ca, orderInfoB )
10.            d2 ← ComputeDelta(objID, ca, orderInfoA )
11.            ApplicationSpecificMergeProc (d1, d2)  //Call application specific merge procedure
     }
}

RumorAgentBasicPull (Owner owner)  at SiteB {
1.   while (true) { //wait for requests from other site
2.             case "RequestOrderInfo(objID)": send orderInfoB of objID
3.             case "RequestDelta(objID,fromID,toID)": send ComputeDelta(objID,fromID,toID)
     }
}
```

Figure 2.3: Pseudo Code for Basic Pull Reconciliation: Site A pulls the data from Site B. BasicPullReconciliation() routine at Site A first requests the ordering information, **orderInfoB**, of the latest version at Site B at line 2. Once it receives the ordering information, it determines the dominance between Site A's latest version and Site B's. In the event that Site A's latest version dominates Site B's (line 5-7), it requests delta that brings Site A's latest version into Site B's. In the event of conflict (line 8-10), a common ancestor, **ca**, can be found if the ordering information provides such capability. The common ancestor is used as a **fromID** in requesting delta. Meanwhile, at Site B, the RumorAgentBasicPull () procedure is running as a server to serve requests from the pulling site, Site A.

dency tracking mechanisms, a malicious site can easily falsify ordering information, which can propagate incorrect information to replicas [47, 48, 40, 46, 28]. For example, a malicious site may cause the shared version to diverge without being detected by improperly attaching the version ID of a original version to another bogus version. This attack can prevent some sites from receiving the original version because a victimized site (i.e., a site that has received bogus version with the same version ID) may incorrectly determine that it has already received the same data that is identified by the same version ID that is also associated with the bogus version.

Thus, it is desirable that a dependency tracking mechanism be designed to enforce the correctness of the protocol that the participating replica sites should follow in assigning the correct ordering among updates.

2.3 Desirable Properties for Optimistic Replication

We now summarize the desirable properties of a reconciliation mechanism for global scale optimistic replication with a decentralized ordering correctness guarantee.

- Scalable Network Bandwidth Consumption: Network bandwidth is still considered expensive resource compared with other resource such as disk and CPU. Thus, corporations need to worry about the sheer volume of network bandwidth that is consumed for optimistically exchanging updates among thousands of corporate servers around the globe.

- Support for Dynamic Membership Changes: In an open peer-to-peer data collaboration, sites are being added or deleted frequently. The reconciliation mechanism should not be impeded by these membership changes.

- Secure Enforcement of Decentralized Ordering Correctness: Without decentralized ordering correctness enforcement, many optimistic replication systems fails to propagate correct data. Due to the collaborative nature of optimistic replication mechanism, it is important to enforce this guarantee.

- Support for Diverse Applications: In the era of pervasive and ubiquitous computing, the shared data object often needs to be shared among many diverse devices and applications.

2.4 Methodology for Evaluating Optimistic Replication: Measures of Success

We use two methods for evaluating an optimistic replication design and its protocols that we will present later in this dissertation: Analysis and Simulation. We use analysis method to evaluate the security enforcement mechanism and use simulations to understand the performance, data availability and network bandwidth consumption. For example, we analyzed our design against the same version ID attack and we also ran simulation how the data availability is affected with this attack. (See Chapter 5.)

Now we describe the simulation methodology used to evaluate our optimistic replication design.

	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

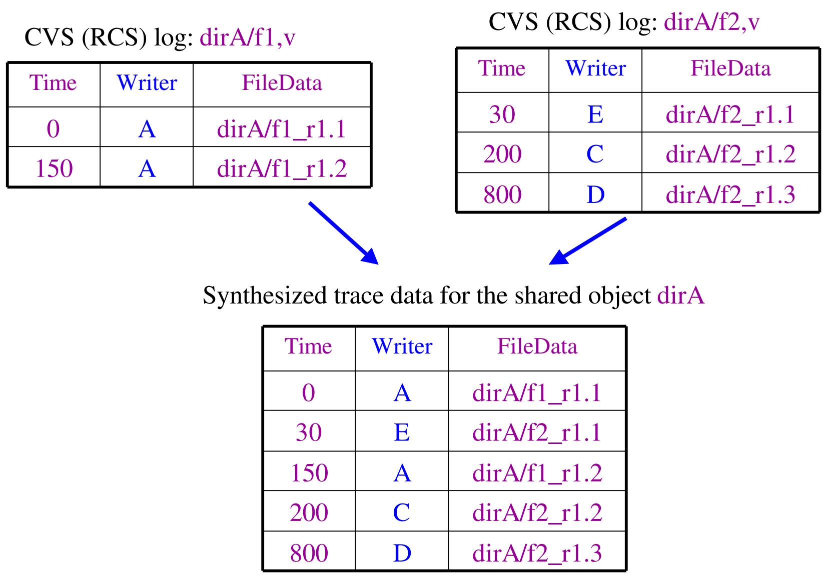
Table 2.1: Trace data from sourceforge.net

2.4.1 Synthetic Multi-User Trace Data

In order to evaluate an optimistic replication system through simulation, we need traces of realistic multi-writer-at-multi-site behavior. Unfortunately, a good multi-writer-at-multi-site trace does not exist.

First, the widely available file system traces are known to have extremely low write-sharing behavior among multi-writers at a single-site. Second, 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.

This leaves us with a need to synthesize realistic traces from a multi-writer-at-a-single-site data. We use CVS logs as a multi-writer-at-a-single-site trace to synthesize a multi-writer-at-multi-site trace. CVS (Concurrent Versioning System) is a versioning software system that enables different users to share mutable data by checking-in and checking-



CVS (RCS) log: dirA/f1,v

Time	Writer	FileData
0	A	dirA/f1_r1.1
150	A	dirA/f1_r1.2

CVS (RCS) log: dirA/f2,v

Time	Writer	FileData
30	E	dirA/f2_r1.1
200	C	dirA/f2_r1.2
800	D	dirA/f2_r1.3

Synthesized trace data for the shared object dirA

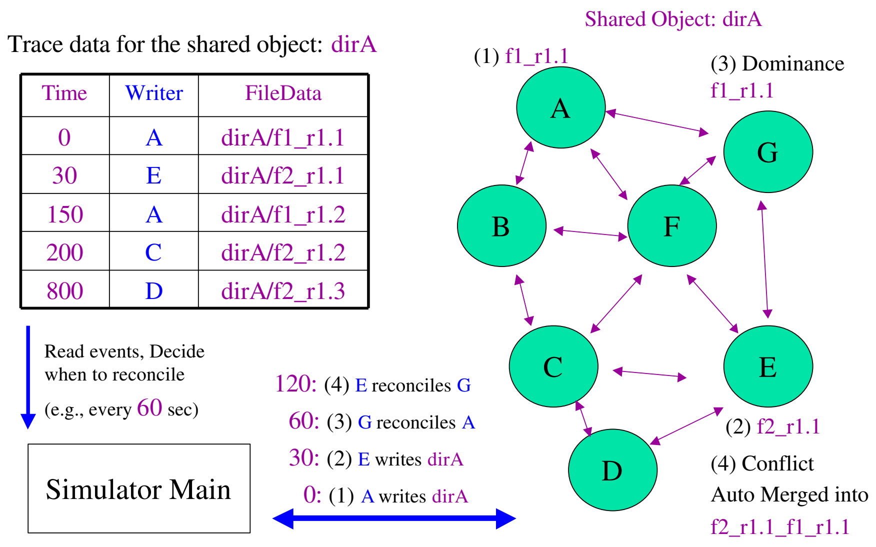
Time	Writer	FileData
0	A	dirA/f1_r1.1
30	E	dirA/f2_r1.1
150	A	dirA/f1_r1.2
200	C	dirA/f2_r1.2
800	D	dirA/f2_r1.3

Figure 2.4: Synthesizing Multi-User Trace from CVS Logs: Every file/subdirectory in the shared object dirA is considered as contents of dirA. By combining these individual file traces, we can create a conflicting (concurrent) update to dirA. The synthetic trace is created by merging each file's RCS log (e.g., f1,v and f2,v) into one synthesized trace file for the shared object dirA.

out at a centralized server (a single site). CVS provides a serialized log (update history) for each file in a shared project. We treat the project itself as under optimistic replication control and consider the individual files in the project as items of shared document content. We treat each writer as one replica site. Figure 2.4 illustrates this trace generation process.

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 CVS logs of the files in a project and then made the result into one serialized trace of events by sorting the events.

Table 2.1 shows that the write-events are bursty–the median is far smaller than the average of the inter-commit time.


Trace data for the shared object: dirA

Time	Writer	FileData
0	A	dirA/f1_r1.1
30	E	dirA/f2_r1.1
150	A	dirA/f1_r1.2
200	C	dirA/f2_r1.2
800	D	dirA/f2_r1.3

Figure 2.5: Trace Driven Simulation: Given the trace data for the shared object, dirA, the simulator reads the event from the trace and calculates when to reconcile based on the given parameters. In this example, anti-entropy is performed at every 60 sec. The simulator picks Site G and A at random at time 60. Site G finds that Site G's latest copy is dominated by Site A's. Site G pulls Site A's latest version, f1_r1.1. When Site E reconciles with Site G at time 120, Site E finds f1_r1.1. and f2_r1.1. are conflicting (concurrent) updates to dirA, so Site E merges these using a given merge procedure.

2.4.2 Trace Driven Simulation

The simulator reads events in sorted order from a trace file and simulate write event at sites. The events are in the form of [time, user, data]. The simulator performs various reconciliation protocols to be described in Chapter 4, 5 and 6. Based on the experiment goal, the simulator can be pre-configured with parameters such as how often to reconcile, whom to reconcile with and how to merge conflicts.

For each reconciliation, the simulator records the elapsed time to manage the data structure of the reconciliation protocol and the result of the reconciliation. In the network simulation case, the simulator measures total and bottleneck network bandwidth consumption. In the event of conflict, which is determined by the given protocol method, the simulator can apply a pre-configured deterministic merge procedure.

Figure 2.5 illustrates how the simulator simulates the events and reconciliations.

2.4.3 Metrics of Success

We will evaluate our design based on analysis and simulation. The analysis will show the robustness of the design against various types of decentralized ordering attacks and the simulation will measure the data-availability of the design compared to the centralized server, as well as the network bandwidth consumed for reconciling updates using our design compared to the widely-used previous mechanism.

2.5 Summary

In this chapter, we identified the needs for scalable and secure optimistic replication to effectively support global scale server replication, to enable open collaboration of shared data, and to manage pervasive data replication in a ubiquitous context. Then, we showed that the design of a scalable and secure decentralized ordering mechanism is the foundation for building a system capable of supporting such optimistic replication. Finally, we discussed why we need a synthesized multi-writer-at-multi-site trace, and how we will evaluate our design using a security analysis and the simulation based on this trace.

Chapter 3

Previous Approaches Limitations: Scalability, Security, and False Conflict

This chapter provides an overview of approaches to optimistic replication. First, it introduces the concept of the version history graph and describes how optimistic replication utilizes version histories for reconciling updates. Then, it introduces two well-known implementations of the version history graph: the version vector and the causal history. It also discusses the relations between the version vector and the causal history approaches with detailed examples.

Then, this chapter discusses the version vector approach's fundamental scalability and security limitations. First, it explains how version vector based approaches entail complicated management for site addition/deletion and have limited scalability in terms of the number of replica sites. Second, it characterizes the security limitations of version vectors by listing the various attacks on decentralized ordering correctness, to which version vectors are vulnerable. Finally, this chapter discusses the cumulative effects of false conflict among diverged versions. Using version vectors or timestamps as identifiers will create false conflicts that can cumulatively create further conflicts among descendent versions.

3.1 Previous Approaches to Optimistic Replication

3.1.1 Version History Graph

Each replica site creates and merges versions; both the versions and the relations among the versions constitute a history of versions. This history of versions can be depicted in the form of graph, where each node in the graph represents a version and an arrow between nodes indicates the derivation of a new version from previous ones. We call such an acyclic graph a *version history graph.*

We say version X *dominates* version Y iff version X is derived from version Y iff there is a directed path from Y to X. In other words, X is a child version of Y and Y is parent version of X. Such *dominance* relations are transitive. For example, if X dominates Y and Y dominates Z, then X dominates Z as well. Likewise, a given version, X, dominates its parents and, also, transitively its ancestors. Also, descendent versions of X (i.e., all the versions that are derived from X) dominate X.

We say two versions are in *conflict* when neither version dominates the other. We say the two conflicting versions have *diverged* from a common ancestor version. By exchanging updates, each site can converge the diverged versions or can bring an old version up-to-date with a newer version. Such a procedure is called a *reconciliation* process. Through a series of reconciliation processes, each site will eventually receive the latest version, which dominates all the previous versions (diverged or old) in the system. We call it a *convergence* when every site possesses the same latest version (i.e., the same version with the same version history).

Determining the dominance relation among version is important in this reconciliation process. With dominance information, each site can determine which version is derived from which version, and can acquire latter version to make the shared data up-to-date. In the conflict case (i.e., case with diverged versions), each site needs to exchange the conflicting versions to create a new merged version. The newly merged version dominates the previous conflicting versions.

A derivation such as creating a new version or merging versions can be expressed as an *operation delta*, a function that takes previous versions as inputs and produces a new version as output. Thus, each version at a site can be expressed by a series of operation deltas applied to the previous versions known to that site. Likewise, the process of reconciling replicas can be expressed as the process of "exchanging operation deltas". One can efficiently select a sequence of deltas for bringing another site up-to-date by maintaining some form of version history along with deltas, since the operation delta can be described as incremental updates to previous versions. The size of operation delta is typically smaller than the entire version; hence, it is more efficient to exchange operation deltas rather then exchanging entire versions.

Figure 3.1 shows an example of a version history graph. This graph shows the

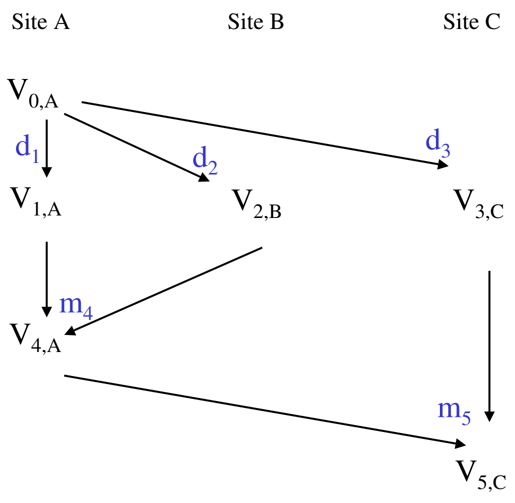


Figure 3.1: Version History Graph: Replica sites collect versions from other sites, modify these versions, and merge versions together. One version *dominates* another version if it is derived from this version (e.g., $V_{4,A}$ dominates $V_{1,A}$, $V_{2,B}$, and $V_{0,A}$). If neither version derived from the other, they are in *conflict* relation. (e.g., $V_{1,A}$ and $V_{2,B}$ are in conflict relation.) d_1,m_4,d_3 and m_5 are operation deltas that take previous versions as inputs and produce a next version as output. For example, operation delta d_2 takes $V_{0,A}$ and produce $V_{2,B}$ as output. Also, operation delta m_4 takes $V_{1,A}$ and $V_{2,B}$ as inputs and produce $V_{4,A}$ as output.

dominance relations among versions that are created and merged by three replica sites. In this graph, each version is labeled with a subscripted pair indicating a globally unique version name [1] and the site at which the version was created.

Figure 3.1 also illustrates a series of pair-wise reconciliations. Here, site A creates $V_{0,A}$ and sends it out to site B and C. Then, site B derives $V_{2,B}$ from $V_{0,A}$ by applying operation delta 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 delta m_4) during pair-wise reconciliation with site B. Here reconciliation is required since neither $V_{1,A}$ nor $V_{2,B}$ dominates the other (*i.e.,* they *conflict* with each other). Similarly, site C creates $V_{5,C}$ by merging $V_{4,A}$ and $V_{3,C}$ (operation delta m_5). Later, when $V_{2,B}$ at site B reconciles with $V_{5,C}$ at site C, both sites should conclude that $V_{5,C}$ *dominates* $V_{2,B}$, so that site B can accept $V_{5,C}$ directly as a newer version. More efficiently, site B can receive operation deltas (d_1,m_4,d_3 and m_5) from site C instead of entire version $V_{5,C}$.

3.1.2 Causal History

One way of utilizing the version history graph for reconciliation is the *causal history* approach. The causal history of a version is defined as the set of versions that are dominated by (i.e., causally precede [43]) the given version. In the language of causal history, we say X causally precedes Y iff Y dominates X. We also call the elements in the causal history the causal predecessors. In other words, the causal history of a version Y is the set of causal predecessors that are dominated by the version Y.

In causal history based approach, each site maintains the causal history of the

[1]For the illustration purpose, we assigned globally unique version names.

latest version that the site has received or created. During reconciliation, each site exchanges the latest version and its causal history, from which each site can check whether or not one version appears in the other's causal history. The dominance relation is determined as following.

Let C(v) be the causal 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 3.1, the versions $\mathrm{V}_{0,A}$, $\mathrm{V}_{1,A}$, and $\mathrm{V}_{2,B}$ causally precede $\mathrm{V}_{4,A}$; hence, the causal history of $\mathrm{V}_{4,A}$ at site A is the set of causal predecessors: $\{\mathrm{V}_{0,A}$, $\mathrm{V}_{1,A}$, $\mathrm{V}_{2,B}$, $\mathrm{V}_{4,A}\}$. Thus, one can determine that $\mathrm{V}_{4,A}$ dominates $\mathrm{V}_{1,A}$, $\mathrm{V}_{2,B}$ and $\mathrm{V}_{3,C}$, which appear in the causal history of $\mathrm{V}_{4,A}$.

Unique Version Identifier Assignment

Since it would be prohibitively inefficient to use each version's entire content as an element in causal history (sites needs to exchange histories), we need a unique identifier for each version so that causal history can be efficiently represented by the version identifier. However, assigning a unique version identifier requires careful planning and coordination among sites. In a simple distributed naming scheme, each site is pre-assigned with a unique identifier. Each site ensures the uniqueness of the local identifier that it assigns for its locally created and merged versions. By prefixing the local version identifier with the site's unique identifier, we can have a globally unique identifier for the version. For example, one can use the version's modification time (i.e., timestamp) as a local version identifier. In this case, the global identifier look like "siteA_9:40:10AM11Mar272004" for the version created by site A at 9:40:10AM11Mar272004.

Unique version identifier is also useful in figuring out operation deltas. During reconciliation, each site can exchange version identifiers instead of entire versions' contents, from which each site can extract the series of operation deltas that need to be exchanged later.

Note that causal history based reconciliation does not necessarily require each site to maintain information regarding parent-child relations among versions. Each site is required to maintain only the causal history of the latest version, because the dominance relation can be determined by simply checking whether one version appears in the other's causal history. However, the parent-child relations are needed to capture the sequence of operation deltas. Thus, each site needs to maintain the equivalent form of version history graph to determine the sequence of operation deltas.

In the causal history approach, the version history graph can be instantiated by maintaining the information regarding parent-child relations among versions in addition to the causal history of the latest version. For example, the version history graph at site A with $V_{4,A}$ as the latest version can be instantiated by the causal history of $V_{4,A}$ and the list of parent-child relations: $\{(V_{0,A}, V_{1,A}), (V_{0,A}, V_{2,B}), (V_{1,A}, V_{4,A}), (V_{2,B}, V_{4,A})\}$.

3.1.3 Version Vectors

Causal histories have generally been considered impractical because their size is of the order of total number of versions in the system. *Version vectors* were designed to overcome this drawback [12, 43].

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 replicas. 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 a value higher than or equal to 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) 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) 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 3.2 illustrates this process. The version vector for $V_{1,A}$ is [A:1,B:0,C:0], after site A generates $V_{1,A}$ by modifying $V_{0,A}$, 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]

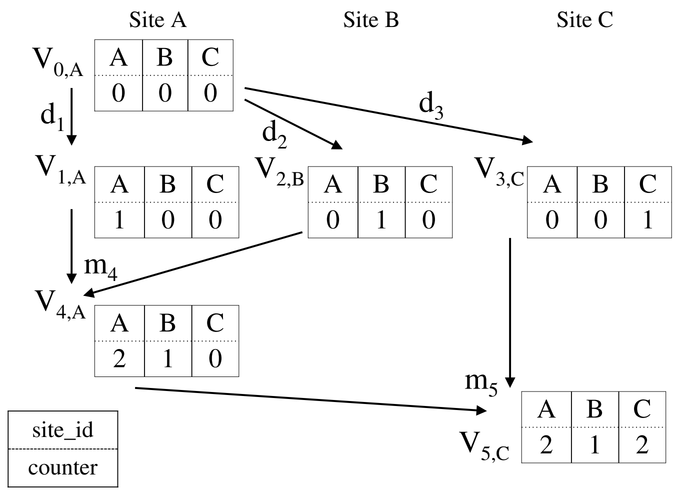


Figure 3.2: Reconciliation using version vectors

to [A:2,B:1,C:0], since A created the merged version.

Similar to the causal history case, the version vector itself cannot constitute the version history graph, although using version vectors one can determine dominance relations. Both the version vector and the causal history need to maintain the parent-child relations among versions to figure out the series of operation deltas to exchange during reconciliation process.

Unique Version Identifier Assignment

Note that some systems such as Bayou use version vectors as summarizations of updates that the site has received or created. The version vector is not used as a unique version identifier. In other words, if one site has the same version vector entries with other site, it only means the sites have seen the same set of operation deltas. The latest version content may be different if the operation deltas have been applied in different order. Ordering information needs to be exchanged in addition to version vectors. For example, Bayou exchanges version vectors and the ordering information using CSN (committed sequence number) that is assigned by a primary server. In contrast, with causal history (the one maintains parent-child relations), the parent-child relations (a sub-graph of version history graph) can convey ordering information among operation deltas.

As shown in the Bayou system, with a version vector mechanism, one can extract the deltas by scanning the complete log history where the log entry is identified with a version vector. It may need a complete scan if the log history is implemented in a linear list rather than a graph. In contrast, with a causal history maintaining parent-child relations, one needs to traverse each node in the version history graph because the log history is in fact implemented as a graph not as a linear list. It may depend on the implementation details if traversing the graph is more efficient and faster than complete scanning of linear list implementation of log history.

3.2 Scalability Limitations of Version Vectors

Version vectors requires one entry for each replica site, which entails the following scalability limitations, which have been recognized in the literature [35, 39, 12, 4, 5].

(1) Complicated management of site membership changes: Replica site addition and deletion require changes in version vector entries of all replica sites.

(2) Limited scalability in terms of number of replica sites: The size of version vectors grows linearly with the number of replica sites.

(3) Labeling each entry in the logs: A unique version ID is required for labeling each entry in the logs to extract deltas during reconciliation.

    To label deltas, 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 version vector of the latest write compactly represents all the preceding writes. Thus, given a version vector from another site, we can collect all the deltas whose [siteid,timestamp] is not dominated (i.e., covered [35]) by that version[35]. 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.

    In addition, we found that version vectors can incur vast amount of false conflicts.

(4) False conflict: The version vector scheme cannot accommodate coincidental equality, i.e., cannot readily exploit instances in which different sites produce the same result independently.

In the case of coincidental equality, 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 4.1.1), 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 for reconciliation, one cannot discern that descendants of V_1 and V_2 are in fact from the same root. More discussion can be found in section 3.4.

In sum, version vectors have some important limitations. The dynamic membership change, capturing coincidental equality and the delta labeling requirement are problematic even with a small number of replica sites. Moreover their space advantage over causal histories is not realized when support for incremental updates is taken into account.

3.3 Security Limitations of Version Vectors

If every node in the system is functioning correctly, then the basic version vector scheme is sufficient as a foundation for optimistic replication systems. By "function correctly", we mean that nodes that are not participating in the protocol for a given object do not interfere with the process and nodes that are participating in the protocol adhere to the rules.

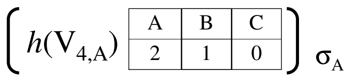
However, because of the cooperative nature of decentralized dependency tracking mechanisms, we cannot assume that all nodes are always operating correctly. A malicious site can easily falsify ordering information, which can propagate incorrect information to replicas [47, 48, 40, 46, 28].

Note that a malicious node may cause the shared state to diverge without being detected by improperly attaching the version ID of one version to another version. Thus, it is desirable that a dependency tracking mechanism enforces the correctness of the protocol that the writer has to follow in assigning correct ordering among updates. With such an ordering correctness guarantee, each site can focus on the data contents without worrying about false ordering of updates.

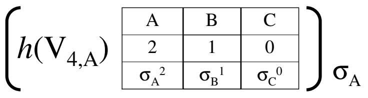
Enforcing ordering correctness with VVs entails having each site have access to a complete log history [47, 48, 40, 46, 28, 44]. To maintain such capability in a decentralized setting or network-partitioned environment, each site has to keep the complete log history, which can be quite a burden, especially for devices with limited storage.

3.3.1 Vulnerabilities and Modified Version Vector

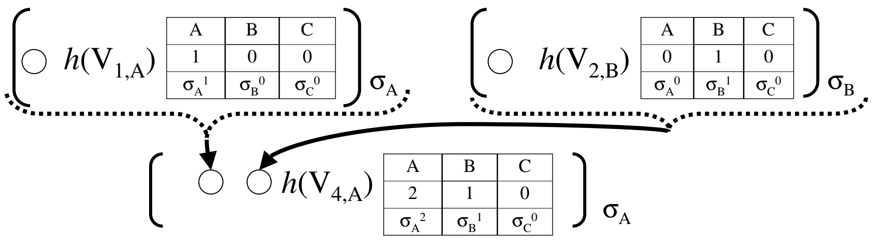
Unfortunately, the correctness of version vector mechanisms depends on trusting each site to maintain its own ordering state correctly. For example, when a site creates a new update, its new version vector should minimally dominate those of all previous updates that the site has seen so far. A malicious site can create a phony update with a grossly inflated version number if each vector entry is not signed by each entry owner. This phony update may improperly suppress future updates whose version vector entries are smaller than the inflated one [47, 48, 40, 46]. To address this “inflating version id” attack, signed version vector (SVV) methods [40, 46] have been proposed. Each entry is signed by the corresponding owner (writer) of the entry so that a malicious writer cannot inflate the entries of other writers.



(a) Unsigned-entry Version Vector (UVV)



(b) Signed-entry Version Vector (SVV)



(c) Linked-history Version Vector (LVV)

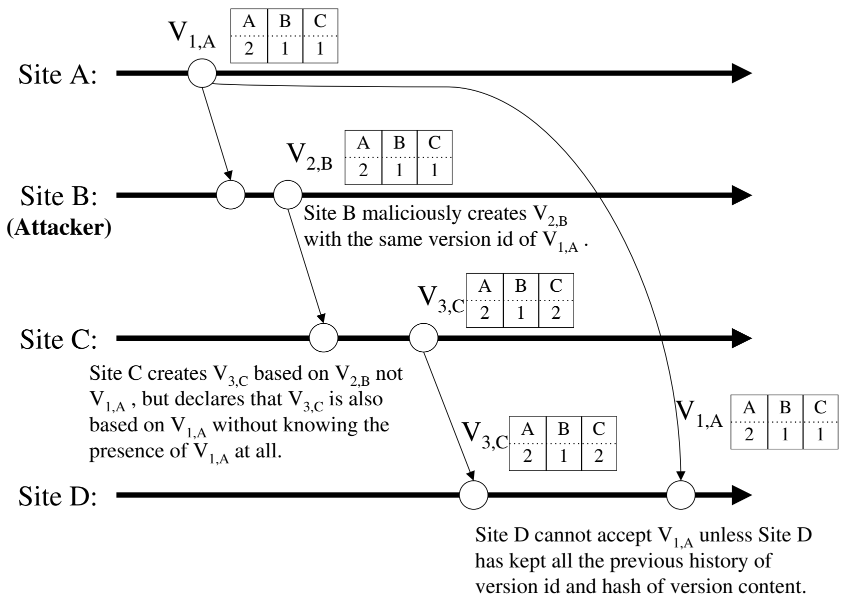
Figure 3.3: Securing Version Vectors Against Attacks: *Unsigned-entry Version Vectors* prevents substitution attacks. *Signed-entry Version Vectors* additionally prevent malicious writers from altering counters of other writers. Finally, *Linked-history Version Vectors* contain enough information to fix a unique history tree. (Note that the example uses the same versions (e.g., $V_{4,A}$) in the previous version history graph)

However, simply signing the entries still leaves one vulnerable to the following "same version id" attack: The malicious site can save the signed entries of recent version vectors and use them as version vectors for phony updates. Moreover, the pre-established version history graph can be changed if the histories are not tamper-evidently linked [47, 48, 40, 46, 28].

Here, we list a number of decentralized ordering attacks and counter mechanisms.

Substitution Attack

One way to sidetrack the optimistic replication process is to alter data without changing the version vectors at all. This is a simple way for an outside attacker to corrupt



Site A:
Site B:
(Attacker)
Site C:
Site D:
V1,A
A B C
2 1 1
V2,B
A B C
2 1 1
Site B maliciously creates V2,B with the same version id of V1,A .
V3,C
A B C
2 1 2
Site C creates V3,C based on V2,B not V1,A , but declares that V3,C is also based on V1,A without knowing the presence of V1,A at all.
V3,C
A B C
2 1 2
V1,A
A B C
2 1 1
Site D cannot accept V1,A unless Site D has kept all the previous history of version id and hash of version content.

Figure 3.4: The Same Version ID Attack: A malicious site creates a new version of a shared object with the same version vector as another site's update. This can partition the shared object into two divergent states and prevent some nodes from ever receiving the victimized update. The victim may never receive the bogus version (i.e., $V_{2,B}$) from the attacker; hence the victim (i.e., Site A) may never know its original version (i.e., $V_{1,A}$) has been attacked and may not be able to counter-act. If there will be no more new updates, the state will stay diverged forever. Even if there is new update, an innocent update (i.e., $V_{3,C}$) based on the bogus version can overwrite the original update in the other partition without knowing the existence of the update. In this case, the victim's update is lost forever without being detected at all.

the data utilized by participants in the protocol.

This attack is trivially addressed by signing the version vector and the data together as shown in Figure 3.3a.

Inflating Version ID Attack

A malicious site can create a phony update with largely inflated version vector, so that the phony update can improperly suppress all the future updates whose version ID is smaller than (i.e., is dominated by) the inflated one.

This attack is not possible if each entry of the version vector is authenticated by its entry owner as shown in Figure 3.3b.

Same Version ID Attack

A malicious site can create an update with the same version ID as another site's update, so that the phony update can be mistaken for the genuine one at some sites. If the phony update reaches a site before the genuine one, the site would not accept the genuine one because it already accepted the phony update as the same version.

This attack is particularly pernicious because the attacker can partition the shared object into two divergent states and prevent some nodes from ever receiving the victimized update. The victim may never receive the bogus version from the attacker; hence the victim may never know the victim's original version has been attacked and may not be able to counter-act by creating a newer version. If there are no newer updates, the state will stay diverged forever. If there is a new update out of bogus update, it will falsely overwrite the updates in the other partition. In this case, the attacker is successful in tricking other innocent writer into overwriting the original update with the same version ID without actually looking at the contents of the original update. An example is shown in Figure 3.4.

One might wonder why an attacker would bother with a same version id attack since the attacker's legitimate update can overwrite the victim's update. However, if the attacker's update overwrites the victim's with correctly increased ordering, then the victim will receive the attacker's update and be able to overwrite the attacker's update again. In contrast, if the attacker's update has the same version id, the victim's site thinks that the updates are the same and will not try to overwrite victim's update.

To address this attack, each participant must have access to a complete revision history. This is what the decentralized ordering correctness guarantee is based on. Any update including malicious ones will eventually appear to everyone including the victim.

So, the last line of defense in optimistic replication is an "undo" capability, since one cannot restrain the writer from producing "bad" contents. If we don't enforce ordering correctness, the system would not be able to counter-act when "bad" contents is introduced or the shared state is diverged.

Log Corruption Attack

A site may be able to interfere with the pre-established orderings in a complete revision history. This attack is problematic because, as just stated, the last line of defense in optimistic replication systems is to be able to undo modifications that have been later determined to be incorrect or otherwise generated by malicious nodes. If the historical ordering of versions is altered, then attempts to undo corrupted updates may "restore" version based on corrupted data or cause an excessive number of valid updates to be discarded.

In the version vector scheme, an attacker may *decrease* counters rather than increase them or perhaps alter the version vector of a piece of data *after* it has been generated. Both of these attacks can alter the apparent causal ordering history.

To address this attack, the signed previous history needs to be included in the next history as shown in Figure 3.3c. Thus, changing established revision history involves changing others' signatures. We assume that nodes propagate history to one another so that they will detect attempts to change history. Such attempts effectively become instances of the Same Version ID attack and can be dealt with accordingly.

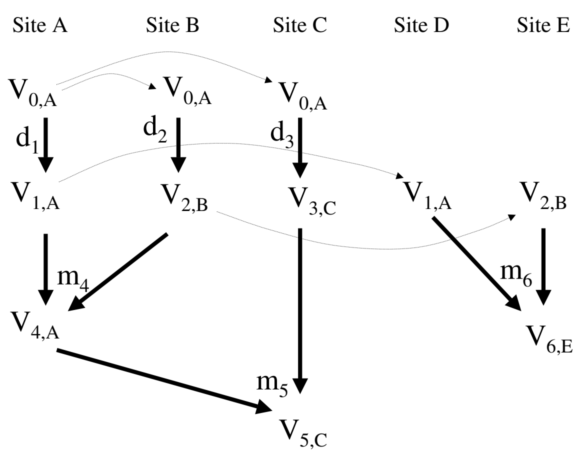


Figure 3.5: Version History Graph with False Conflicts Example: Suppose $V_{4,A}$ has the same content as $V_{6,E}$ (,which is common due to deterministic merge procedure), all descendant versions of $V_{4,A}$ should *dominates* $V_{6,E}$. However, if this dominance is not properly recognized, all descendant versions of $V_{4,A}$ is in *false conflict* with $V_{6,E}$ and $V_{6,E}$'s descendant versions. Thus, the false conflict can create vast number of false conflicts among descendants. Arrows (top-down) indicate derivation of new versions from previous ones (ancestors). Arrows (left-right) indicate copying versions from neighbor sites during reconciliations.

Such a "Linked-history Version Vector" (LVV) is considerably more complex than the original version vector scheme. Further, the final dominance checking mechanism no longer operates exclusively by comparing version vectors — it requires tracing through the version history. In fact, given the secure history, we no longer need the version vectors in LVV. The resulting scheme is the topic of the next section.

3.4 Cumulative Affects of False Conflict

Occasionally, two independent chains of operations produce identical version data. We call such events *coincidental equalities.* As we will show in Chapter 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 the 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 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 itself does not recognize coincidental equality. In Figure 3.5, 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, their IDs have to be assigned differently. (Indeed, if unique version identifier is assigned by prefixing each site's unique name, which is a common practice, the IDs will always be different.) Then, all the descendants of $V_{4,A}$, (e.g., $V_{5,C}$) are in conflict with $V_{6,E}$, although, in fact, all the descendants of $V_{4,A}$ could dominate $V_{6,E}$. This inability to exploit coincidental equality is a consequence of tracking ancestry independent of content. Thus, it is desirable to assign the version identifier not just to be globally unique but also to be able to capture coincidental equality to avoid creating vast amount of false conflicts among descendant versions. Given that most optimistic replication mechanism utilizes deterministic merge procedure, the chance of having coincidental equality in the system is quite high.

3.5 Previous Work

3.5.1 Previous work to address scalability limitation of version vectors

Causal History Log Based Approaches

In the context of mobile communication and reliable message delivery [9], 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. [37] proposed the use of dependency sequences that contain all the causal predecessors of an event as an alternative to the version vectors in the context of mobile communications. A causal history based approach has also been proposed to address the scalability problem of version vectors. However, in using causal histories, a method for providing unique ids for replicas is required. Almeida et al. [4, 5] 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 [25, 27], 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. (It was not clear whether the Coda's approach is particularly motivated by the limitation of version vector.) 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.

Dynamic Version Vectors

Parker et al.[12] 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 have a cumulative affect in creating further false conflicts among the descendant versions. Static version vectors have been used to implement optimistic file replication in Locus and Ficus [38, 34]. Using static version vectors requires site membership to be previously determined.

Ratner et al.[39] 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 added/deleted 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 (i.e., scale to the number of active writers), at the cost of adding the complexity that is entailed in tracking the site's active/passive status.

Methods for dynamic replica creation and retirement using version vector were presented in Bayou [35]. An existing site can introduce a new site by prefixing the name of the new site with the name of the introducing site. If all 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 the number of replica sites, although such a case would be rare in practice. With 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.

3.5.2 Previous work to address security limitation of version vectors

Spreitzer et al. [47, 48] proposed the countermeasures based on version vectors to deal with server corruption in weakly consistent replication. The mechanism requires the systems to build a tamper-evident audit trail that is composed of linked write-requests and write-replies along with digital signatures. They detailed how the proposed mechanism can deter the possible attacks against both data and meta-data (i.e., ordering state).

Section 5.7(Related Work: Securing Causal Relationship) presents other fundamental work such as secure time-stamping and Merkle Tree based authentication.

3.6 Summary

The version vector approach has fundamental limitations related to scalability and security. These problems are especially pronounced in pervasive and ubiquitous computing environments where hundreds of diverse devices and computers need to exchange updates in a pervasive and secure way.

First, version vectors are well known to require complicated management for site addition/deletion and 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 [4, 35, 39, 37]. Depending on when the membership change information arrives, a reader may receive two updates with vectors of different sizes.

Second, version vectors are vulnerable to various attacks and faults on decentralized ordering, because each entry of version vector summarizes the causal relations in a lossy way, not 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 [47, 48, 40, 46, 28].

By ensuring decentralized ordering correctness, the optimistic replication system can guarantee that all updates are not vulnerable to a decentralized ordering attack. With this guarantee, each site can focus on validating the quality of data contents in a cooperative way. Whether the content of an update is good or bad, the update will be delivered in correctly formed ordering.

Finally, it is important to assign the version identifier not just to be globally unique but also to be able to capture coincidental equality to avoid creating vast amount of false conflicts among descendant versions. Given that most optimistic replication mechanism utilizes deterministic merge procedure, the chance of having coincidental equality in the system is quite high. We will discuss how we achieve this in the next two chapters.

Part II

Our Approach

Chapter 4

Hash History Approach

This chapter now describes a scheme that overcomes scalability limitations of version vectors in Chapter 3. Basically, the idea is to take a causal history approach, which readily addresses problems of the dynamic membership change and the growth of vector size, and use a hash of a version as a unique ID, thus addressing the unique site naming, the coincidental equality capturing, and delta labeling requirement. Also, since delta labeling requirement 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 use HH as an abbreviation for Hash History and VV for Version Vector.

4.1 Data-Centric Hash History Design

Recall that causal history approach needs to identify version uniquely so that it can determine the version dominance.

4.1.1 Using Hashes as Version IDs

First, we consider three choices for a unique version id: (i) version vectors, (ii) unique site id + local timestamp pairs, and (iii) hash of version + epoch number (e.g., a count of the previous same versions) pairs. 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[35], 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

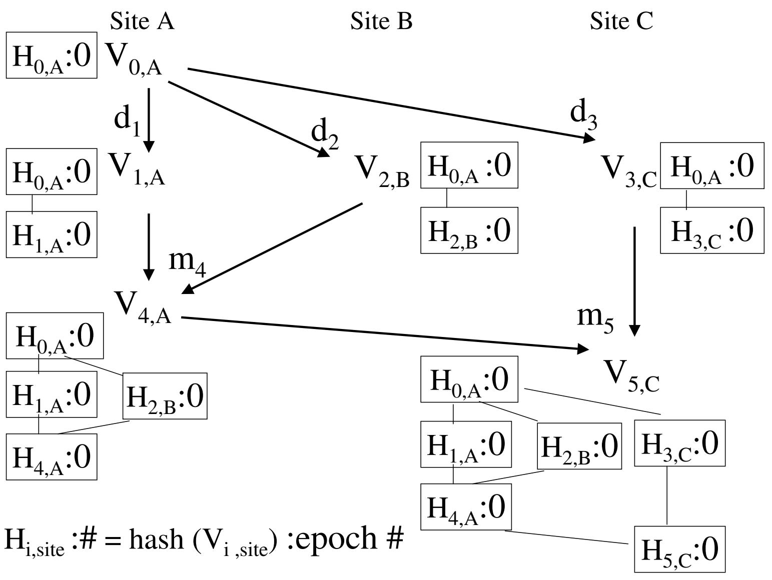


Figure 4.1: Example of Reconciliation using Hash History: During reconciliation between site B and C, each site exchanges only the latest hashes, $H_{2,B}$ and $H_{5,C}$. Given $H_{2,B}$, Site C can determine that the V_5 is a revision of V_2 of site B, because $H_{2,B}$ belongs to site C's history. Given $H_{5,C}$, however, since $H_{5,C}$ cannot be found in Site B's history, Site B *assumes* $H_{5,C}$ is a revision of $H_{2,B}$, and pulls the $H_{5,C}$'s histories from Site B.

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.

4.1.2 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 4.1 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 $H_{0,A}$.

4.1.3 Faster Convergence

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 × 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.

4.1.4 Truncating the Hash History

Classical techniques for truncating logs can be applied toward pruning hash histories. The *global-cutoff timestamp* (e.g.,[42]) and the *acknowledgment-timestamp* (e.g., [19]) 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.

4.2 Evaluation of Hash History Efficacy

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.

4.2.1 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, described in section 2.4.1, 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.

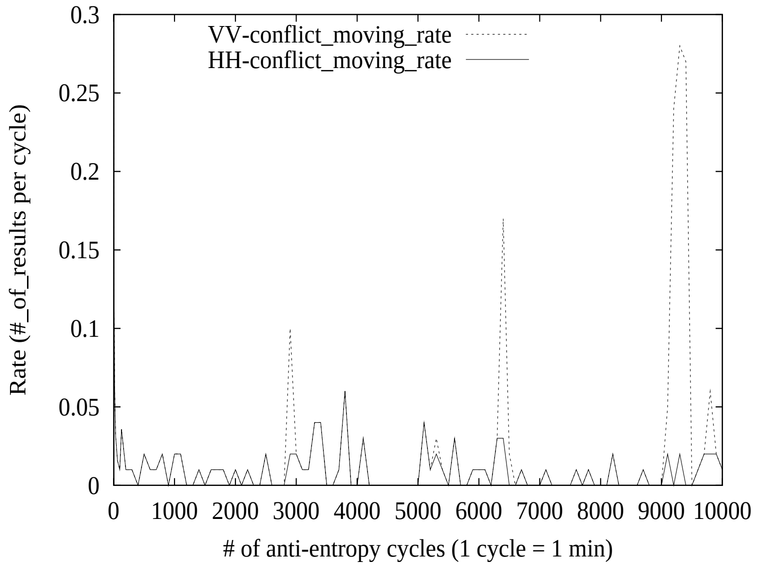


Figure 4.2: Conflict rate of VV and HH from pcgen shown between 0 to 10000 cycles

4.2.2 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 4.2 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 4.3. 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

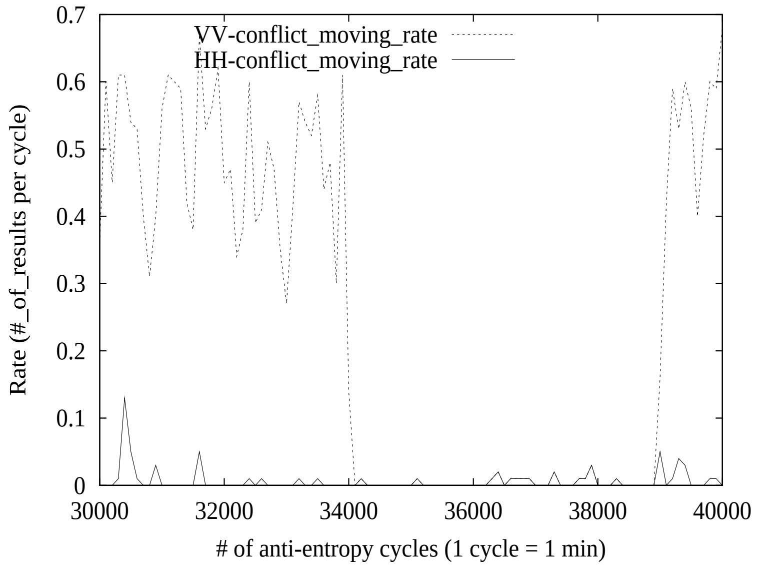


Figure 4.3: Conflict rate of VV and HH from freenet shown between 30000 to 40000 cycles

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 4.4 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

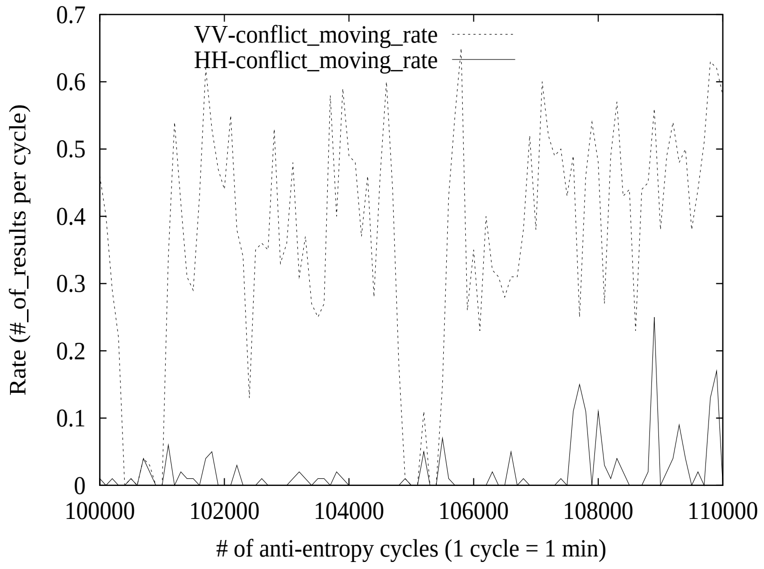


Figure 4.4: Conflict rate of VV and HH from dri shown between 100000 to 110000 cycles

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 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 4.2 - 4.4 show more drastic difference as anti-entropy cycles increases.

4.2.3 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

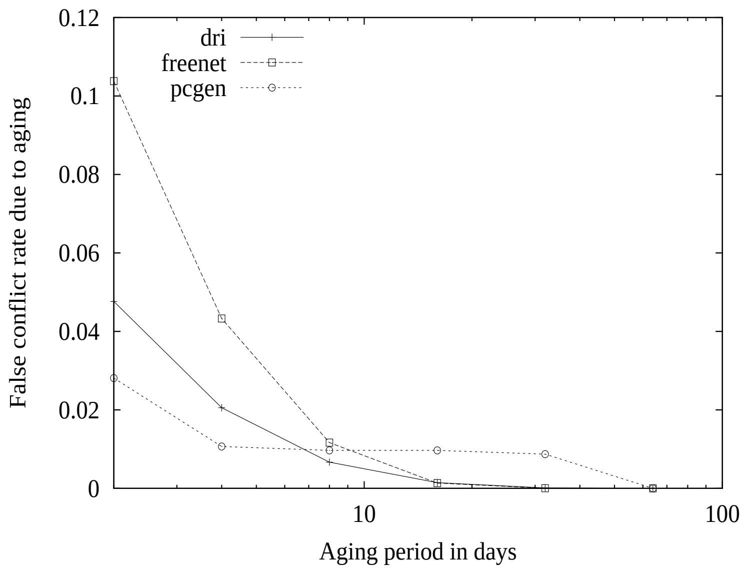


Figure 4.5: False conflict rate due to aging

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 4.5 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 4.1.

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 4.1: Average HH size with the aging period

4.3 Related Work

4.3.1 Using Hash as Identifier

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[13], Past[17], Publius[52], FreeHaven[16], and FreeNet[10] identify read-only objects using a hash over the contents of those objects. OceanStore[26] 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.

4.4 Summary

To address the scalability limitation of version vectors, this chapter 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.

Chapter 5

Summary Hash History Approach

The Hash History (HH) approach described in Chapter 4 provides scalability in terms of the number of participants and dynamic membership changes. However, HH does not provide a decentralized ordering correctness guarantee. This chapter describes a secure decentralized ordering mechanism, called Summary Hash History (SHH), which is an extension of HH. Instead of using the content hash as the version identifier, SHH uses a *summary hash* to provide the decentralized ordering correctness guarantee. This chapter describes SHH, and also shows that SHH has the following useful property: secure reconstruction of log history and convergence across partitioned networks. By using a summary hash as a version ID, aggressive log pruning is possible even with the requirement of maintaining a complete history to defend against various ordering attacks described in Chapter 3.

5.1 Secure Summary Hash History (SHH) Design

In the causal history approach, as described in Chapter 3.1, ordering dependencies among revisions is determined by checking whether a given version appears in another version's causal history [40, 43, 25]. A summary hash history (SHH) is a causal history, where the version identifier is the *summary hash.* The summary hash (S_i) of a version (V_i) is defined as the cryptographic collision-resistant hash over predecessors' summary hashes and the hash of the version content, i.e., $S_i = hash\ (S_{i_1} \parallel ... \parallel S_{i_n} \parallel H_i)$, where S_{i_j} is the summary hash of the j-th predecessor (in lexicographically sorted order) for the version V_i, $H_i = hash(V_i)$ and $\parallel$ is concatenation. For example, if the predecessors for the version V_i are S_l and S_r, then S_i is $hash\ (S_l \| S_r \| H_i)$. (Assume that S_l comes before S_r in lexicographically sorted order.)

The inclusion of predecessor hashes in the summary hash is similar to that of the Merkle's tamper-evident hash tree [30] or hash chaining [6, 21]. Likewise, the summary hash is collision resistant. A proof of this property can be found in Appendix A. Thus, by using this summary hash as a version id, one can readily address various ordering attacks as described in Chapter 3, including the "same version id" attack and the "log corruption" attack.

5.1.1 Verification of Summary Hash History (SHH)

Since the summary hash is collision resistant, it is computationally infeasible to find two different summary hash histories for a given summary hash. Roughly speaking, there is a unique summary hash history associated with a given summary hash. The summary

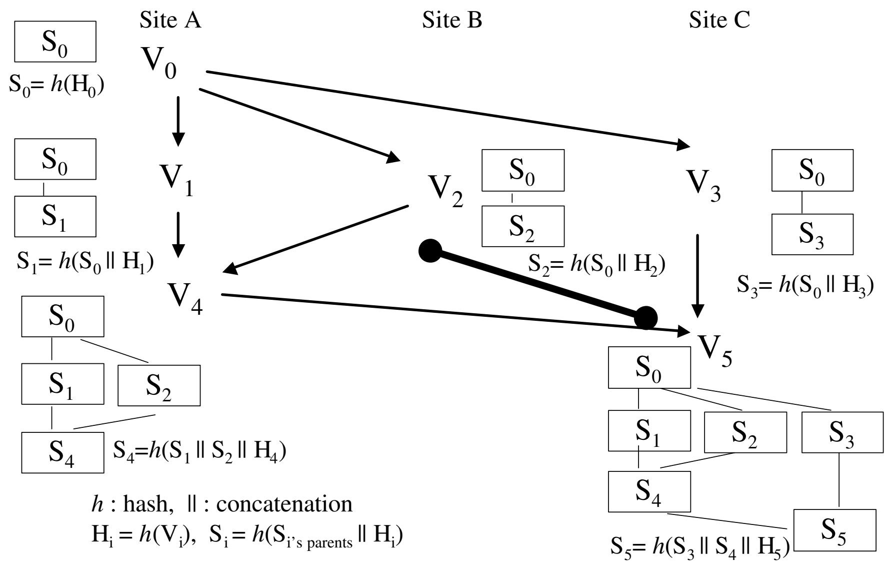

Site A
Site B
Site C
V_0
V_1
V_2
V_3
V_4
V_5
$S_0 = h(H_0)$
$S_1 = h(S_0 \| H_1)$
$S_2 = h(S_0 \| H_2)$
$S_3 = h(S_0 \| H_3)$
$S_4 = h(S_1 \| S_2 \| H_4)$
$S_5 = h(S_3 \| S_4 \| H_5)$
h : hash, $\|$: concatenation
$H_i = h(V_i)$, $S_i = h(S_{i\text{'s parents}} \| H_i)$

Figure 5.1: Example of Reconciliation using Summary Hash History: The summary hash (S_i) of a version (V_i) is the hash over the predecessor's summary hash(es) and the hash of the version, i.e., $S_i = h(S_p \| H_i)$, where p is i's predecessors, $H_i = h(V_i)$, h is a collision resistant hash function and $\|$ means concatenation. During reconciliation between site B and C, each site exchanges only the latest summary hashes, S_2 and S_5. Given S_2, Site C can determine that the V_5 is a revision of V_2 of site B, because S_2 belongs to site C's history. Given S_5, however, since S_5 cannot be found in Site B's history, Site B *assumes* S_5 is a revision of S_2 or in possible conflict, and later pulls the S_5's histories and data from Site B or other local sites that have already received S_5.

```
Boolean shhA.verifySHashDFS (SHash sh, Set verified ) {
1.  if (verified.hasItem (sh)) { return  true }
2.  String concat //to contain the concatenation of the parents' SHashes
3.  String contenthash ← shhA.getContentHash(sh)
4.  SHashArray parents ← LexicographicalSort (shhA.getParents(sh))
5.  if (parents ≠ null)
6.          for  i ← 0  to  parents.length-1 {
7.              if (!shhA.verifySHashDFS (parents[i], verified)) return false
8.              concat.add (parents[i]) //add every verified parent
            }
9.          concat.add (contenthash)
10.         if ( (sh.str = = getSHA1(concat)) and (verifySign(sh.str, sh.sig)) ) return true
11.         else return false
        }
12.     return false  //sh is not found in verified and has no parent
}

Boolean shhA.verifySHash (ObjectID objID, SHash  sh) { //sh.str is summary hash string
1.  if (!verifySign(sh.str, sh.sig)) return false            //sh.sig is signature of sh
2.  Set verified ← owner.getVerified(objID)  //owner keeps previously verified cache
3.  return shhA.verifySHashDFS (sh, verified) //verified has an initial root-hash by default
}
```

Figure 5.2: Pseudo Code for Verifying Summary Hash: The verifySHashDFS() routine recursively verifies the summary hash and data content until a previously verified summary hash or the initial root hash is reached.

hash is in fact a compact and secure summarization of all the causally preceding writes. By simply signing the latest summary hash, one can authenticate the associated previous history.

Thus, given a signed summary hash, first, we can check the authenticity of the signature of the summary hash, and then verify the summary hash's associated previous history. The verification can be done by traversing the directed acyclic version history graph with summary hash as version identifier. One can recursively verify the summary hash and data content until a previously verified summary hash or the initial root hash is reached. For example, to verify the summary hash S_5 for V_5 (i.e., H_5) in Figure 5.1, one needs to locate summary hashes for both S_3 for V_3 and S_4 for V_4, and check if the hash over $(S_3\|S_4\|H_5)$ matches S_5; if it matches, then one recursively verifies S_3 and S_4 until one reaches a previously verified summary hash or S_0 (the initial root hash). Figure 5.2 describes a pseudo code for verifying summary hash given initial root hash and SHH. Likewise, a set of deltas can be collected by traversing the directed acyclic graph [45, 25].

Figure 5.3 shows a pseudo code for collecting delta.

5.1.2 Secure Log Reconstruction for Light-weight History Access

Ensuring decentralized ordering correctness entails significant overhead, such as requiring each site to maintain a complete history. For example, to address same version id attacks, each site should have access to the complete history. In a decentralized setting, each site might maintain the complete history at a trusted location (e.g., secure local storage). This direct approach might impose a substantial burden, especially in supporting optimistic replication among sites with limited storage (e.g., mobile devices).

However, with summary hash, a site can aggressively prune its log history. Since a self-verifying summary hash can compactly represent its associated ancestral versions, one can prune out the old history aggressively and keep only the latest summary hash. Given a summary hash, one can locate a self-verifiable history of the pruned parts from neighbor sites. Since we can verify the validity of the given history, the history need not come from trusted sites. With this property, each site can then incrementally reconstruct the previous

```
Delta shhA.collectDeltaDFS(SHash curID, Delta delta, Set visited) {  //Depth First Search
1.  If (visited.contains(curID)  OR  delta.contains(curID))) {return delta }
2.  Put curID into visited.
3.  Put shhA.getLog(curID) into delta
    //Now recursively traverse into parents
4.  ObjectArray parents ← shhA.getParents(curID)
5.  if (parents ≠ null) {
6.          for i ← 0  to parents.length-1
7.              delta ← shhA.collectDeltaDFS(parents[i], delta, visited)
    }
8.  return delta
}

Delta owner.ComputeDelta (ObjectID objID, SHash fromID, SHash toID) {
1.  shhA ← owner.getSHH(objID)
2.  visited ← new Set()       visited.add(toID)
3.  Delta delta ← shhA.collectDeltaDFS(fromID, new Delta(), visited)
4.  return delta
}

SHH owner.ComputeSHHDelta (ObjectID objID, SHash fromID, SHash toID) {
1.  Delta delta ← owner.ComputeDelta (objID, fromID, toID)
2.  return delta.extractSHH()   //extracting SHH from the delta by simply removing logs
}
```

Figure 5.3: Pseudo Code for Collecting Deltas: The deltas are collected by traversing the summary hash history graph using depth first recursive traversal. For each **parent**, it keeps traversing the **shhB** (line 6-7) until it finds a **toID**.

	VV	VV + Hash	Epoch# + Hash	Summary Hash
Version ID Space	O(#nodes)	O(#nodes + hashsize)	O(epoch_num_size + hashsize)	O(hashsize)
Log Space O(IDspace * #revs)	O(#nodes*# revs)	O(#nodes*# revs)	O(#revs)	O(#revs)
Dominance check	Comparing VV entries		Checking if IDs belong to other's log history	
Secure log reconstruction	Version ID cannot rebuild the pruned (lost) log securely.			Version ID can rebuild the pruned (lost) log securely
Aggressive log pruning by keeping the version ID only	Possible at the cost of full content transfer since IDs can determine the dominance relation. (Dynamic membership change maintenance can complicate log pruning)		IDs cannot determine the dominance relation except equality. (epoch# can complicate pruning)	Securely rebuilt log history can determine version dominance and allow incremental delivery

Figure 5.4: Comparison of other dependency tracking mechanisms with Summary Hash History regarding log management

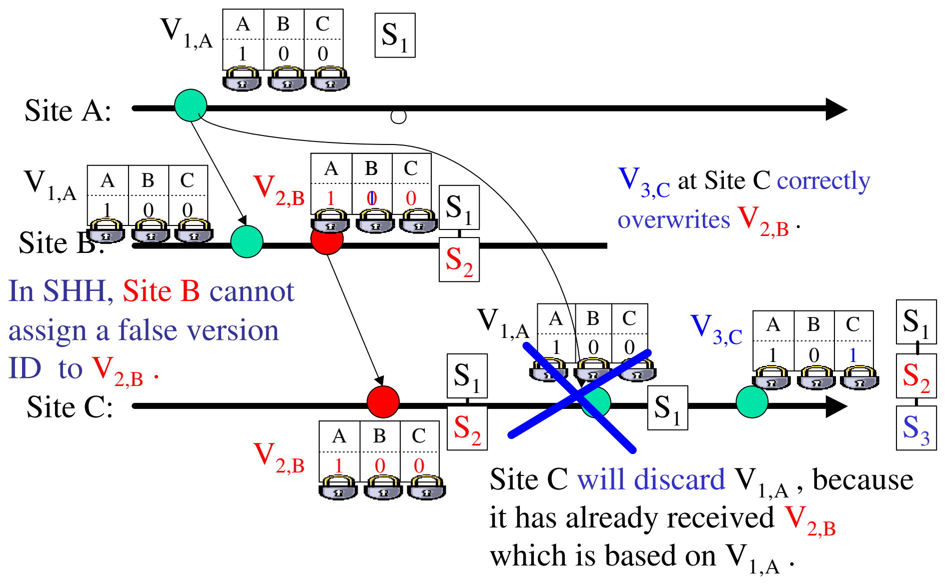
history enough to determine whether the summary hash appears in the other site's history. We call this SHH ability the *secure log reconstruction property.* This is quite beneficial property given that weakly consistent replication systems in general require each site to maintain the logs of data (or alternatively operations on the data) to exchange updates in a decentralized way. It is possible that modified VVs can also prune aggressively with the provision that it can retrieve the pruned parts of its history from neighbors on demand. However, as the reconstruction of history is not secure, a site with insufficient- or no-history needs to trust its neighbor to return a correct history. One might have to ask a number of sites to determine the correct history based on consensus, which would be quite expensive.

Figure 5.4 compares the log management mechanism of SHH with other various approaches.

5.2 Countermeasures with Summary Hash History

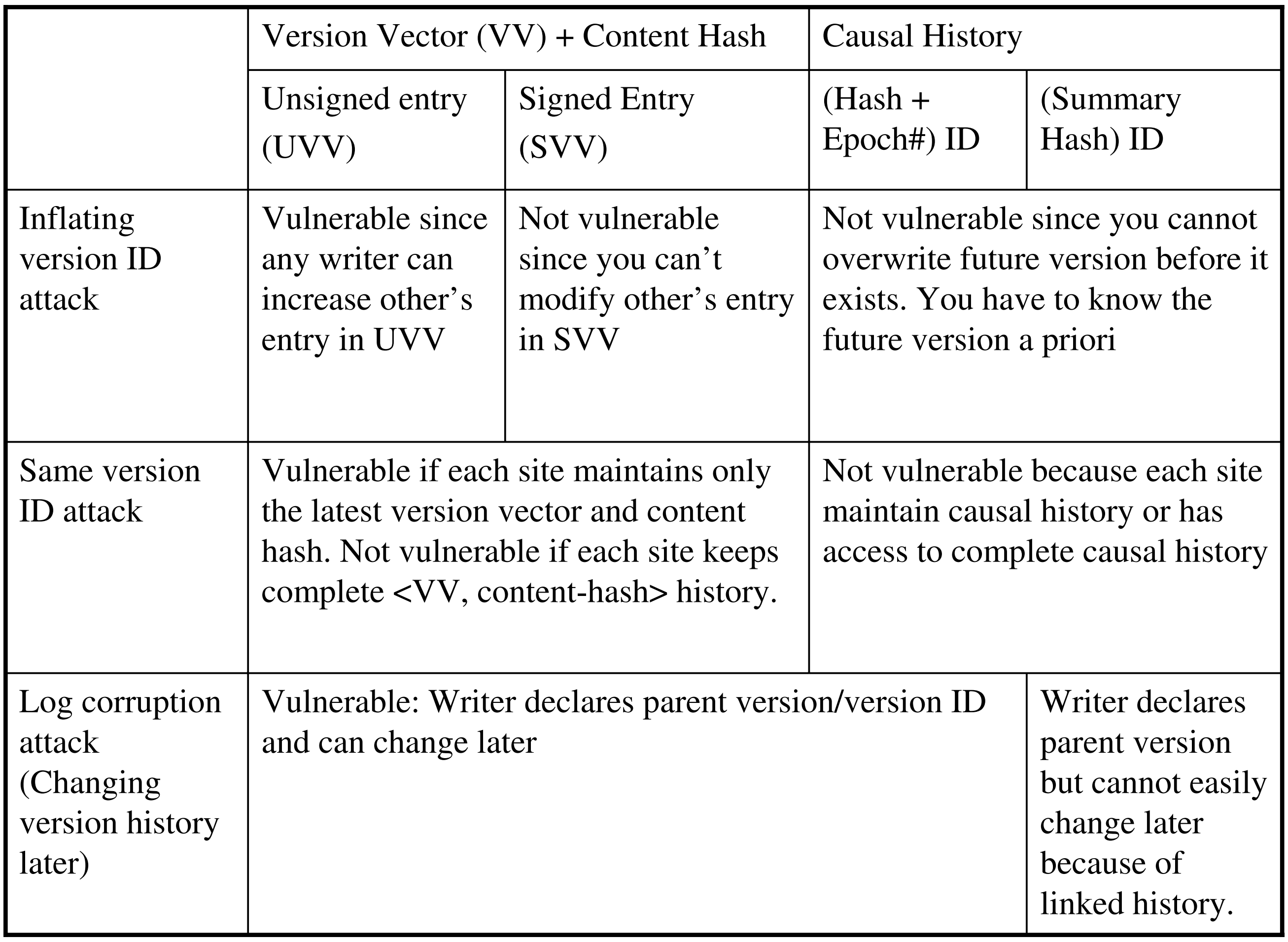
Unlike a version vector approach, a causal history approach in general is not intrinsically vulnerable to an "inflating version id" attack. The attacker would have to include the victim's future update into the causal history of the attacker's own update. Because it is generally difficult to predict a victim's future update, the causal history approach has a fundamental resiliency to this attack. Accordingly, so does the SHH.

Also, by using summary hash as version identifier, SHH is not vulnerable to the same version ID attack or the log-corruption attack. One cannot assign the same summary hashes to different versions, because the mapping between the version identifier and the content is determined by a hashing function, not by the declaration of a participating site.



Site A:
Site B:
Site C:
V1,A
V2,B
V3,C
A B C
1 0 0
1 0 1
S1
S2
S3
V3,C at Site C correctly overwrites V2,B .
In SHH, Site B cannot assign a false version ID to V2,B .
Site C will discard V1,A , because it has already received V2,B which is based on V1,A .

Figure 5.5: The Same Version ID Attack is Not Possible with SHH: A malicious site can create a new version of a shared object with the same version vector as another site's update. However, with SHH, this is not possible because summary hash includes the content hash of the version. The attacking bogus version will be detected as divergent version. Also, SHH can clearly verify that an update (i.e., $V_{3,C}$) is based on the bogus version (i.e., $V_{2,B}$) not on the original version (i.e., $V_{1,A}$).



	Version Vector (VV) + Content Hash		Causal History	
	Unsigned entry (UVV)	Signed Entry (SVV)	(Hash + Epoch#) ID	(Summary Hash) ID
Inflating version ID attack	Vulnerable since any writer can increase other's entry in UVV	Not vulnerable since you can't modify other's entry in SVV	Not vulnerable since you cannot overwrite future version before it exists. You have to know the future version a priori	
Same version ID attack	Vulnerable if each site maintains only the latest version vector and content hash. Not vulnerable if each site keeps complete <VV, content-hash> history.		Not vulnerable because each site maintain causal history or has access to complete causal history	
Log corruption attack (Changing version history later)	Vulnerable: Writer declares parent version/version ID and can change later			Writer declares parent version but cannot easily change later because of linked history.

Figure 5.6: Comparison of other dependency tracking mechanisms with Summary Hash History regarding various attacks

If the mapping is based on site's declaration, a site can easily change such information later, which basically makes the log-corruption attack possible. Obviously, with SHH, it is difficult to change past log history without being noticed, because changing history will result in different summary hash identifiers.

Figure 5.5 shows an example that SHH is not vulnerable to the same version ID attack. With SHH, a malicious site cannot create a new version of a shared object with the same identifier as another site's update. Because summary hash includes the content hash of the version, the summary hash will be different if the content hashes are different. SHH also preserves the history in tamper-evident way. It can verify which version overwrote which version. As in this example, SHH can show that an innocent version overwrites the bogus version not the original version. (Recall that, in version vectors, an innocent update (i.e., $V_{3,C}$) based on the bogus version can overwrite the original update in the other partition without knowing the existence of the update. In this case, the victim's update is lost forever without being detected at all.)

Figure 5.6 summarizes SHH's resistance to various attacks and faults, in comparison with other approaches.

5.3 Scalable Reconciliation with SHH

As a causal history approach, dominance relations (i.e., ordering dependency among revisions) can be determined by checking whether a given latest summary hash appears in another site's summary hash history [40, 43, 25].

To exchange updates through a reconciliation process, each site needs to determine the dominance relation among the latest versions. To do so, a complete SHH needs to be sent over to the other site so that one site can determine the dominance relation. Thus, one site site might receive the entire SHH from another site and determine the version dominance by checking if its latest version appears in the other.

More efficiently, the SHH can be sent over the network only when it is necessary. For example, during reconciliation, a signed summary hash, s_2, (for a version v_2,) is sent from the other site. Then, the receiving site can check whether the summary hash, s_2, appears in the receiving site's summary hash history. If so, the receiving site's latest version, say v_1 dominates v_2. Otherwise, the receiving site assumes v_2 to be a previously unseen version (i.e., either v_2 dominates v_1 or they are in conflict), and later downloads the SHH associated with s_2.

Chapter 6 discusses the SHH reconciliation protocols in detail.

5.4 Deterministic, Commutative, and Associative Merge with SHH

In some applications with optimistic replication, the merge procedures are deterministic, commutative and associative (DCA). A merge operation is deterministic iff a merge operation, $m(x, y)$, deterministically produce the same result from the same input input x and y. For example, if human user merges diverged versions, the merge operation by human may not be deterministic, since the merged result may be different even with the same input. An automated merge operation that is deterministic will produce the same result if the input is the same. We call a merge operation, $m(x, y)$, is commutative iff $m(v_1, v_2)$

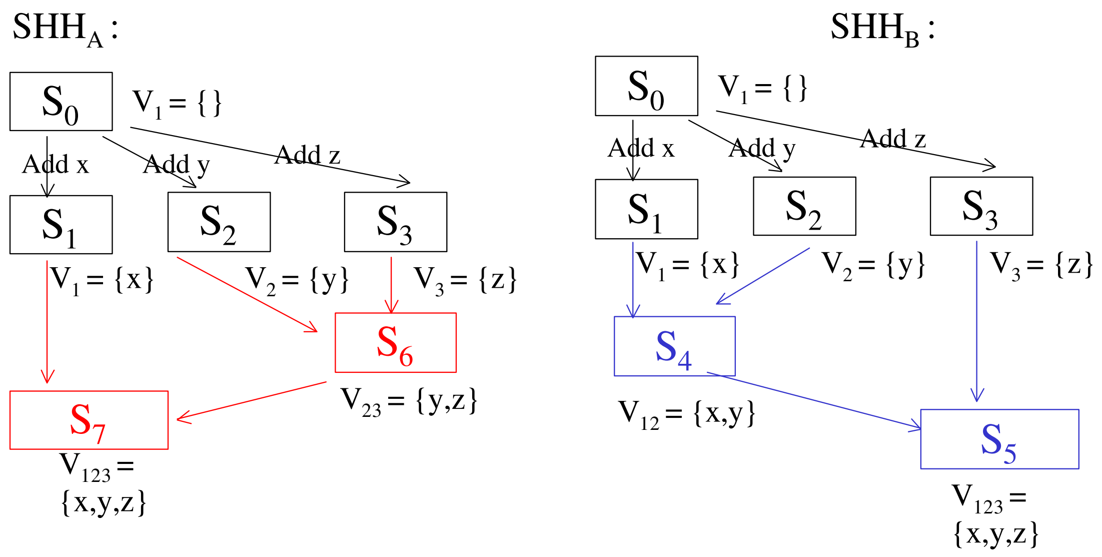


Figure 5.7: Deterministic, Commutative, and Associative Merge (DCA Merge) Example: Site A produces S_7 by merging S_1 with S_6. S_6 was previously produced at another site, Site C, by merging S_2 and S_3. Meanwhile, Site B produces S_5 by merging S_3 with S_4. S_4 was previously produced at another site, Site A, by merging S_1 and S_2. Since the merge (adding items to a set) is DCA, the content of S_7 and S_5 should be the same.

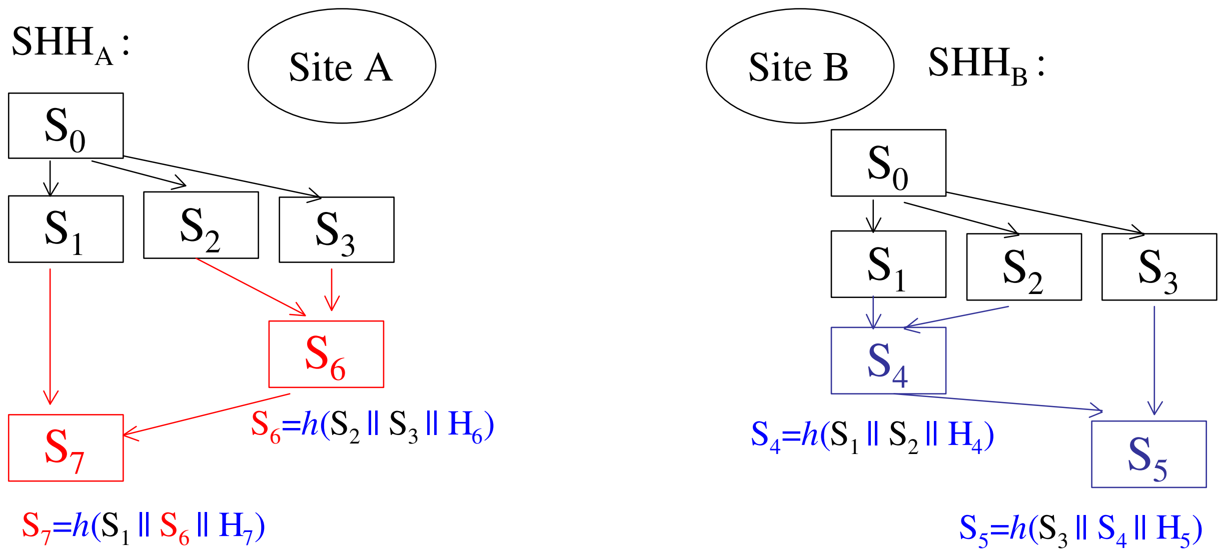


Figure 5.8: SHH assigns different identifier for DCA Merge: Even though, the content of S_7 and S_5 are the same due to DCA merge(i.e., $H_7 = H_5$), SHH assigns different identifiers for S_7 and S_5. S_6 was previously produced at another site, Site C, by merging S_2 and S_3. S_4 was previously produced at another site, Site A, by merging S_1 and S_2.

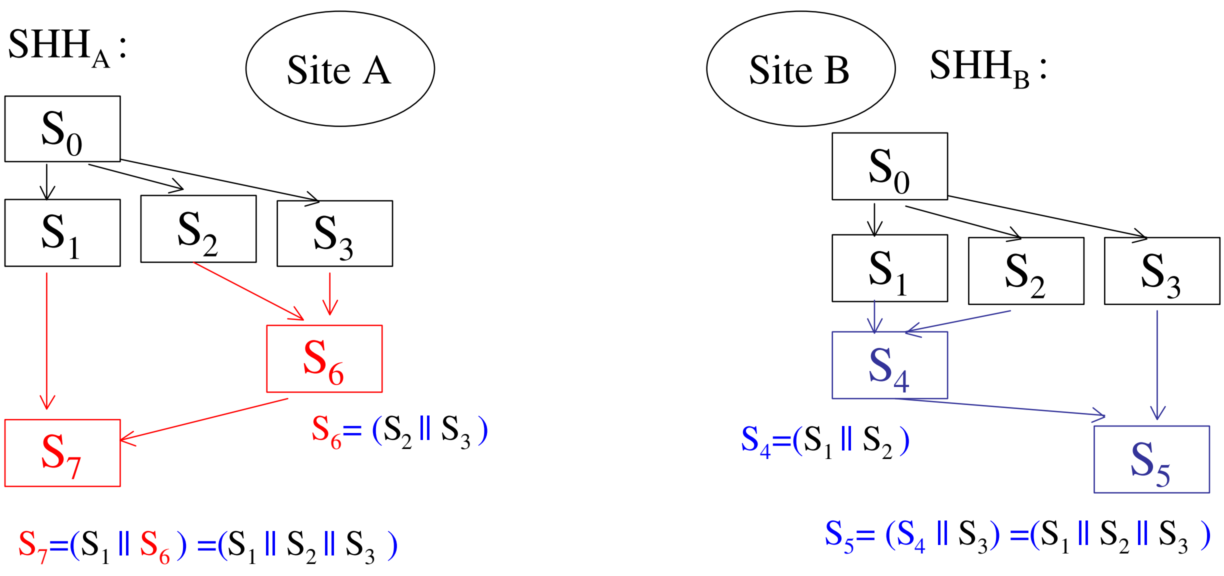
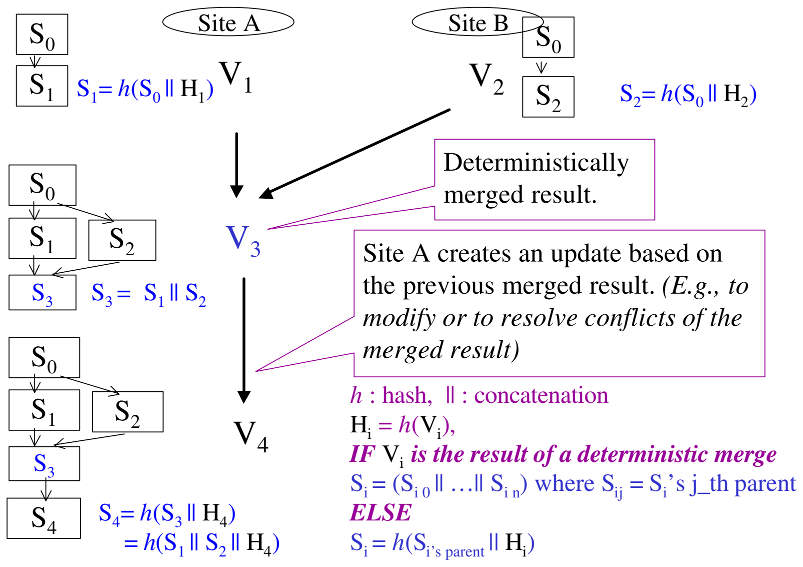


Figure 5.9: SHH assigns the same identifier for DCA Merge: If all the merge procedures are Deterministic, Commutative, and Associative, then, the summary hash for the merged version is the concatenation of parents summary hashes in lexicographically sorted order. Now, $S_5 = (S_1 \| S_2 \| S_3)$ and $S_7 = (S_1 \| S_2 \| S_3)$ have the same identifier with the same data content(i.e., $H_7 = H_5$).



Site A
Site B
S_0
S_1
$S_1 = h(S_0 \| H_1)$
V_1
V_2
S_0
S_2
$S_2 = h(S_0 \| H_2)$
Deterministically merged result.
V_3
S_0
S_1
S_2
S_3
$S_3 = S_1 \| S_2$
Site A creates an update based on the previous merged result. *(E.g., to modify or to resolve conflicts of the merged result)*
V_4
S_0
S_1
S_2
S_3
S_4
$S_4 = h(S_3 \| H_4)$
$= h(S_1 \| S_2 \| H_4)$
h : hash, $\|$: concatenation
$H_i = h(V_i)$,
IF V_i ***is the result of a deterministic merge***
$S_i = (S_{i\,0} \| \ldots \| S_{i\,n})$ where $S_{ij} = S_i$'s j_th parent
ELSE
$S_i = h(S_{i\text{'s parent}} \| H_i)$

Figure 5.10: Write after DCA Merge Example: This example shows that the new rule of assigning concatenated parent hashes as the summary hash identifier for the DCA merged version works well with previous summary hash construction mechanism. A new write after DCA merge is considered the same as the merged result when the merge operation is not guaranteed to be DCA.

produce the same merged output as $m(v_2, v_1)$. A merge operation is called associative iff $m(v_1, m(v_2, v_3))$ produces the same merged result as $m(m(v_1, v_2), v_3)$. A DCA example is shown in Figure 5.7.

The strict summary hash construction should assign different id when the merge history is different even though the merged results are the same. An example is shown in Figure 5.8. In some application, it may be desirable to assign the same id if the merge is DCA and the output is guaranteed to be the same if the input is the same. Figure 5.9 shows how SHH solves this problem by assigning the summary hash of the merged version as the concatenation of parents summary hashes in lexicographically sorted order when all the merge procedures are DCA. Finally, this new rule of assigning the concatenated parent hashes as the summary hash identifier for the DCA merge does conform well with previous rule of summary hash as shown in Figure 5.10.

5.5 Coincidental Equality and Convergence Across Partitioned Networks with SHH

As we discussed in section 3.4, it is typical that identical versions with the same content are produced at different sites. Indeed, in optimistic replication systems such as Bayou and Coda's disconnected operation system, such an occurrence can commonly happen when the same deterministic merge procedure is used to resolve the same set of conflicting updates. We defined such an occurrence a coincidental equality.

Coincidental equality may be divided into two classes. One is the case when the identical content is independently produced from the identical histories, which we call *total*

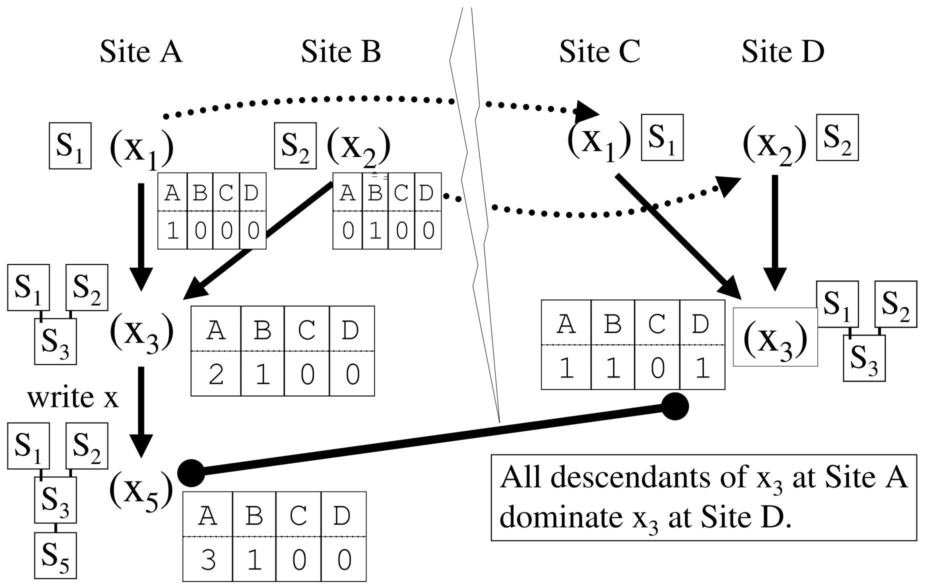


Figure 5.11: Convergence across Partitioned Networks: Site A creates X_3 by merging X_1 and X_2, while independently, site D creates X_3 based on the same predecessors. Suppose Site A's X_3's content is coincidentally equal to Site D's X_3, then all the descendant versions of Site A's X_3 will dominate Site D's X_3. Version vector cannot capture this total-equality (it declares conflict), which can cumulatively incur false-conflicts among descendant versions. However, summary hash includes content hash in order to capture the total-equality instantly even in the network partitioned environment.

equality. The other is the case when the identical content is independently produced from either identical or non-identical (i.e., different) histories, which we call *content equality.* By definition, the set of total equality cases is a subset of the set of content equality cases.

As shown in Chapter 4 (HH approach), the use of a content hash (i.e., the hash generated over data content) as a version identifier can capture the content equality. By capturing coincidental content equality, optimistic replication can converge faster producing no-false conflict. This has a vast cumulative effect in the false-conflict rate among descendant versions [24]. However, 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 an epoch number was needed to distinguish the different versions with the same content.

Unlike Hash History scheme which uses [content hash, epoch number] pairs as version identifiers, summary hash history (SHH) does not need an epoch number, since the summary hashes of versions with the same content will be different if their histories are different from each other. Thus, with SHH, sites will declare equality only when both the version content hash and its predecessors' histories are identical (i.e., the total equality case). In comparison, with Hash History , sites determine equality, regardless of the predecessors' histories, when both the content hash and the epoch number are identical (i.e., the total equality case and some content equality cases with the same epoch number). Note that hash-epoch pair scheme does not declare equality when the epoch numbers are different.

Of course, capturing the total equality using summary hash depends on the predecessors' summary hashes being deterministically included in the creation of the successor summary hash. We sort the predecessors' summary hashes based on the hex hash value.

Otherwise, we may have two different summary hashes with the same history. For example, $S_i = h(S_l \| S_r \| H_i) \neq h(S_r \| S_l \| H_i)$. Sorting the predecessors ensures that we are able to capture the total equality case where sites (in partitioned networks) independently applied the accrued non-commutative updates in the same order.

Interestingly, SHH's ability of capturing total equality can also allow distributed replica to converge even across partitioned networks, because with SHH each site can assign the same identifier to the versions with total equality without communicating at all. An example is shown in Figure 5.11.

5.6 Evaluation

In this section, we present and evaluate the SHH implementation in Java and study the SHH properties through an event-driven simulations. We modify the simulator that was used for studying the properties of HH in Chapter 4, instead of hash-epoch pairs, using summary hashes as version identifiers.

5.6.1 Event-Driven Simulation Setup

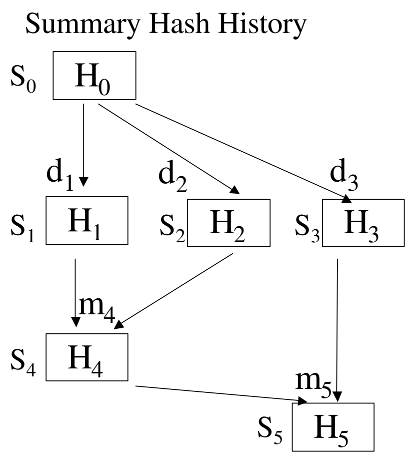
In our event-driven simulation, CVS traces from an open source project (sourceforge.net) are used for generating update events. Each writer represent a node(site). Each site has logs, hash histories(HHs), static version vectors(VVs), and summary hash histories(SHHs). The update event is considered to be independently created by the writer. If the user is new, we create a new site for the user. The simulator reads events in sorted order from a trace file and generates write events. The events are in the form of [time, user, filename].

Periodically, per given interval parameter, the simulator performs anti-entropy by picking two sites at random. By default, 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 are performed. The first site initiates the reconciliation with the second site by exchanging the VVs, HHs, and SHHs. The first site determines equality, conflict, and dominance. In the case of conflict, the first site merges the conflicts, and then sends the merged version back to the second site along with the updated VV, HH, and SHH. The ordering determination (i.e., equality, dominance and conflict) during reconciliation is calculated as a moving average over 100 cycles.

A conflict determination occurs when neither dominates the other and they find a common ancestral version. By default, both history should have the first version as a common ancestor. If they can't find a common ancestor, it means they belong to different history or their summary hash histories may have been pruned too aggressively.

In the case of conflict (i.e., neither update event is based on the other), each site merges the conflict. To simulate the independently merged result, the simulator choose one of the two deterministic merge procedures with 50% probability, and uses the chosen procedure to merge the conflict. This way, we can expect a total-equality would occur half of time when the sites independently merge the same conflict versions.

An "undecided" determination occurs when two sites cannot determine the dominance relation with the given histories of both sites. This typically happens when the history is pruned too aggressively.



Corresponding Hashtable Format

Summ Hash	Content Hash	Parents SHash	Data delta
S_0	H_0	ObjectID	d_0
S_1	H_1	S_0	d_1
S_2	H_2	S_0	d_2
S_3	H_3	S_0	d_3
S_4	H_4	S_1 :S_2	m_4
S_5	H_5	S_4 :S_3	m_5

Latest : S_5 , H_5

h : hash, $||$: concatenation, $H_i = h(V_i)$, $S_i = h(S_{i\text{'s predessors}} || H_i)$

Figure 5.12: Implementation of Summary Hash History with Hashtable

5.6.2 SHH Implementation and Performance

We have implemented SHH using Java's Hashtable data structure to enable efficient dominance checking. For example, Figure 5.12 shows the Hashtable format for a given SHH. The predecessor-successor relations are maintained as a pair in the table. The summary hashes are used as a key so that dominance checking is handled by lookup in the Hashtable. This Hashtable also serves as a verification cache so that a previously verified summary hash is not repeatedly verified.

During reconciliation, when a sender's version dominates the receiver's latest version, the dominating version and its associated summary hash histories are sent over so that the receiving site can incorporate these into its own SHH. SHH can be exchanged incrementally by sending the list of tuples in the Hashtable.

To collect a set of deltas needed for reconciliation, we maintained each delta next

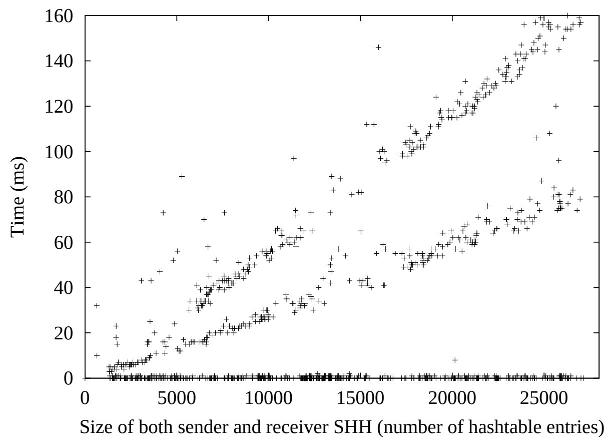


Figure 5.13: Reconciliation Time (for determine dominance + for add/combine SHHs) with SHH implementation using Java's Hashtable

to the corresponding summary hash. The deltas can be collected by traversing the trees until a common ancestor is reached, and then by visiting all the concurrent sibling deltas.

Since our purpose here is to understand the implementation of SHH data structure and implementation in Java's Hashtable, we measured the reconciliation time as the time to determine the version dominance and the time to manage the data structures for summary hash history such as adding the SHH of new version to the SHH of the old version.

We measured the elapsed reconciliation time to determine the version dominance, and the time needed to manage the data structures for summary hash history. The measurement was conducted on a x86 machine with 699MHz CPU, Linux os_release 2.4.710smp, and 1.4G main memory. We verified that the reconciliation time for hash history grows linearly as a function of the number of revisions; and for the version vector as a function of the number of sites.

Figure 5.13 shows that the reconciliation time grows linearly as the number of summary hash history entries increases. We plotted the pair of (SHH size, reconciliation time). Interestingly, the plotted pairs form into three distinctive groups. The bottom group (close to X-axis) shows the case in which the summary hash history look-up takes constant time (i.e., equality cases). The middle group indicates the case in which one dominates the other (i.e., one of the SHH has to be combined with the SHH of the other dominating version). The top group indicates the case in which both SHHs incorporate the other's. Note that with SVV, one still has to compare all the entries in the version vectors to determine equality, while doing so takes constant time with SHH implementation with Java's Hashtable. This performance evaluation shows that the major overhead in maintaining SHH data structure is in combining the summary hashes during reconciliation.

5.6.3 SHH Property Study

We studied the properties of SHH using an event-driven simulator to investigate the following issues.

1. Same Version ID attack: We sought to find the relation between the number of sites with correct data delivery and the number of attackers. We simulated a same version id attack by injecting a phony version and measuring the number of sites that received the original version through anti-entropy reconciliation.

2. Aggressive Pruning (Some sites keep only the latest summary hash):

   To understand how the light-weight SHH can efficiently support small devices with limited storage, we ran a stress test simulation. We measured how often the sites with no-log-history (i.e., with only the latest summary hash) had to reconstruct their own history to determine version dominance as we increased the percentage of no-history

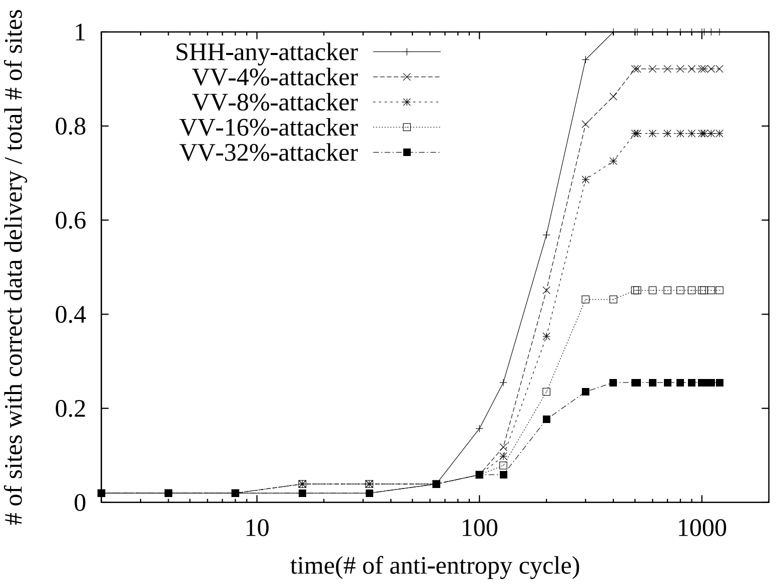


Figure 5.14: Correct delivery rate as function of attacker rate plotted over time: Random anti-entropies are used for propagating updates.

sites in the system. We varied parameters such as the number of anti-entropy cycles between updates.

Same Version ID Attack

We simulate the same version ID attack by injecting an original copy and a phony copy with the same version vector. Since SHH is based on content, the original copy and the phony copy will have different summary hash as identifier. We recorded which site receives which copy as the updates are propagated through the random pair-wise reconciliations (i.e., anti-entropy).

As depicted in Figure 5.14, the number of sites with correct data using SHH increases faster than any VV mechanism after around 100 cycles. SHH eventually delivers correct data regardless of the number of “same version id” attackers in the system. This

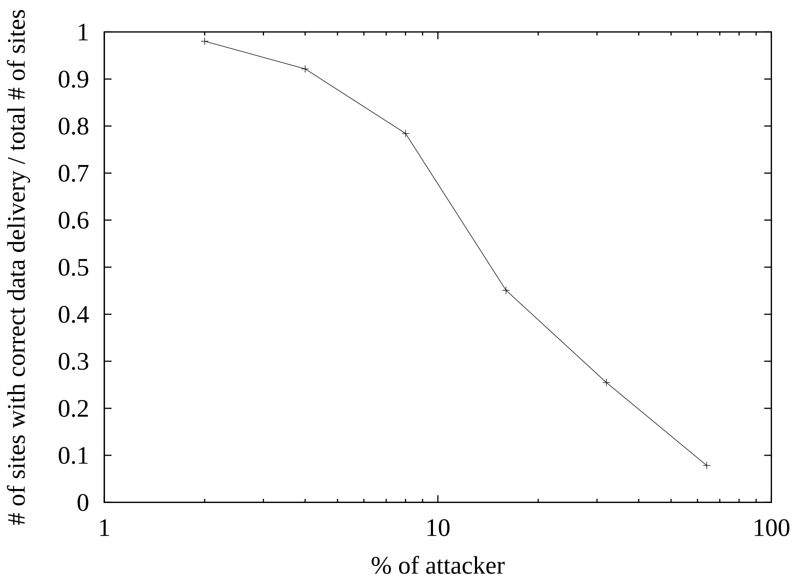


Figure 5.15: Final correct delivery rate as function of attacker rate: Random anti-entropies are used for propagating updates. It shows that the number of sites receiving bogus data grows *exponentially* as the number of attackers increases.

is due to the fact that the SHH is able to keep propagating both the correct and phony data, treating both as concurrent. In contrast, the VV mechanism cannot detect these two as concurrent. A modified VV, in which equality is determined by comparing the version hashes in addition to the VVs, can detect these two as concurrent. Without such modification, however, sites with the correct data cannot propagate their data to the sites that already received either correct or phony data.

We also found that the number of sites with correct data decreases *exponentially* as the number of attackers increases as shown in Figure 5.15.

Aggressive Pruning: A Stress Test

An "undecided" determination is made when two sites cannot determine the dominance relation with the given histories of both sites. In this case, the sites with no log-history

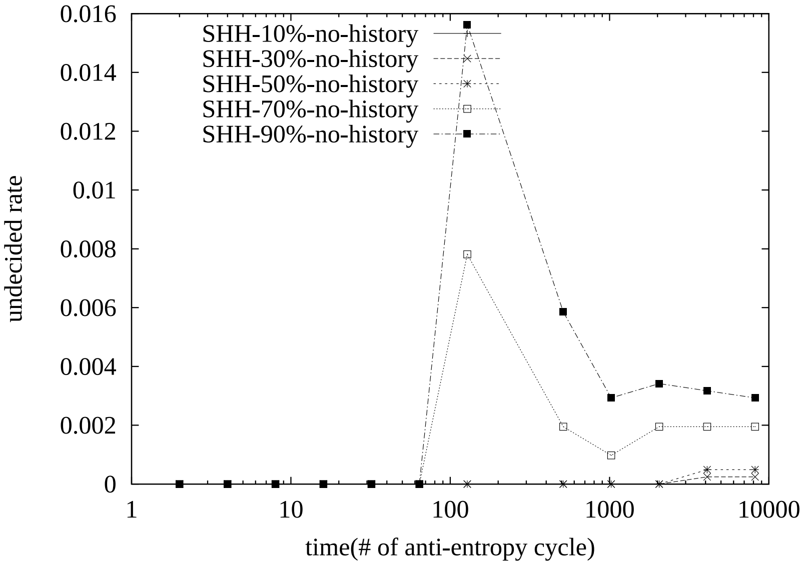


Figure 5.16: Undecided rate as function of sites with no-history rate: quadratic anti-entropy between update events.

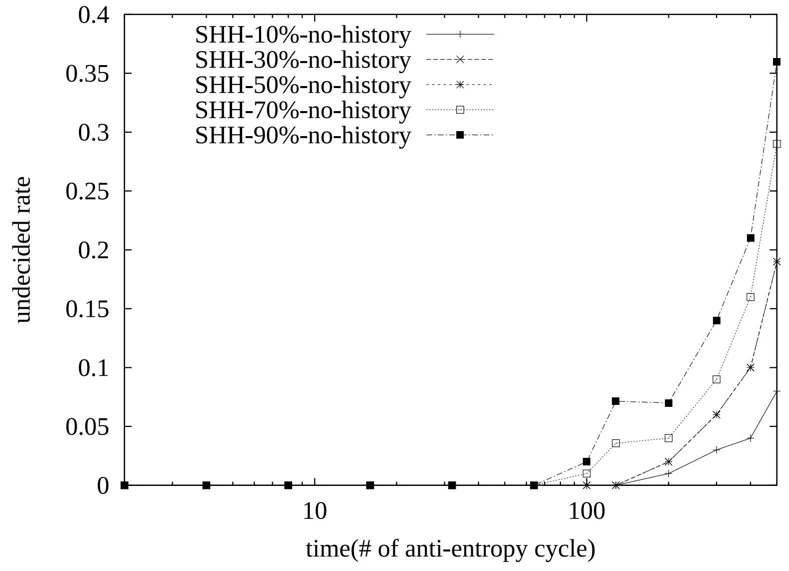


Figure 5.17: Undecided rate as function of sites with no-history rate: linear anti-entropy between update events.

have to reconstruct its own history to determine version dominance.

Figure 5.16 shows that the number of occasions requiring secure log reconstruction can be kept low if there is enough anti-entropy cycles between events. In Figure 5.16, the simulator performed ${}_nC_2$ (i.e., $n(n-1)/2$) number of anti-entropy between events (i.e., n is the total number of sites), and the undecided rate was kept quite low- below 0.016. The undecided rate is defined as the number of undecided determination results over a moving window of 10 anti-entropy cycles.

Interestingly, the undecided determination happens only when both sites are in the no-history category during reconciliation, because when a no-history site reconciles with a full-history site, the dominance relation is guaranteed to be determined. Even when a no-history site reconciles with a no-history site, they are able to determine equality. If there have been enough anti-entropy cycles between events, it is likely that there are many equality determinations among no-history sites. That is why the undecided rates were kept low in Figure 5.16.

In contrast, in Figure 5.17, when we manually forced the number of anti-entropy cycles between events to be linear to the total number of sites, there was exponential growth in undecided determination.

Finally, we measured the undecided rate using CVS logs. CVS (Concurrent Versioning System) is a versioning software system that enables different users to share mutable data by check-in and check-out at a centralized server. CVS provides serialized logs (update history) for each file in a shared project. We treated the project itself as under optimistic replication control and considered the individual files in the project as items of shared

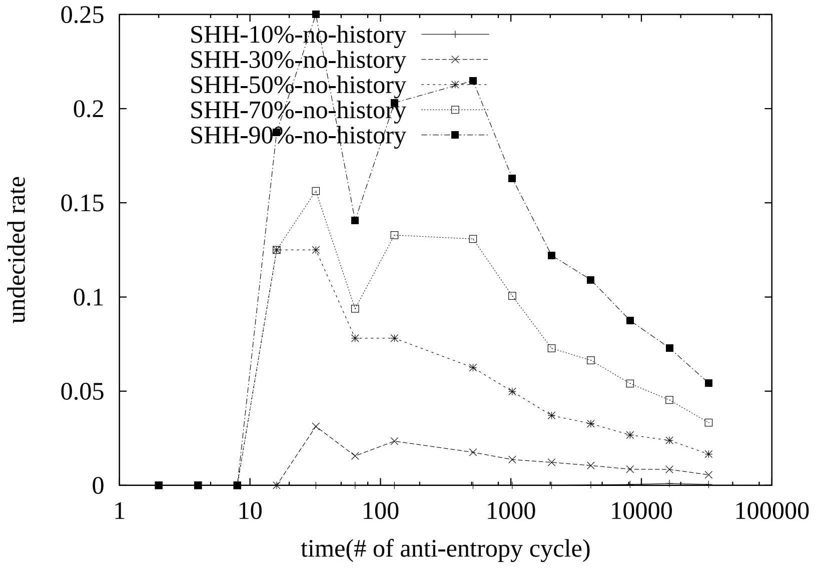


Figure 5.18: Undecided rate as function of sites with no-history rate: a CVS data (sourceforge.net/freenet) with anti-entropy every 1 minute.

document content. We treated each user as one replica site. Figure 5.18 shows that the undecided rates were kept well below 0.05 with 30% no-history sites when the anti-entropy period was 1 minute. Even with 90% no-history sites, the undecided rates were kept below 0.25.

5.7 Related Work: Securing Causal Relationship

Securing the causal relationship among historical events has been proposed in many places. For example, Merkle's tree [30] is used for authenticating large number of data blocks efficiently through tree hierarchy. Time stamping services(TSS) [6, 21] provide tamper-evident data structures that are similar to Merkle's trees to address the following denial and forgery attacks [40, 46]:

(i) Denying causal ordering: A malicious site can deny pre-existing partial orderings among updates by post-dating (i.e., forward-dating) or deleting an old version.

(ii) Forging causal ordering: A malicious site can forge a pre-existing causal relationship by pre-dating (i.e., back-dating) a new version or inserting it between pre-existing orderings.

Interestingly, Spreitzer et al. detailed the possible attacks against both data and meta-data (i.e., ordering state) and proposed countermeasures to deal with server corruption in weakly consistent replication. [47, 48]. The proposed mechanism essentially builds a tamper-evident audit trail that is composed of linked write-requests and write-replies along with digital signatures. In comparison, the SHH approach in Chapter 5 does not use version vector.

Also, time line entanglement [28] allows one to build trustworthy time stamping services through entangling disparate time lines that are maintained at independent systems.

Time stamping services in general are interested in archiving the tamper-evident timeline history of who made which changes and when. Thus, the writer's signature and the signed acknowledgment from the TSS are typically included in generating the recursive hash generation.

In contrast, our summary hash construction is mainly concerned with the what and the when, and not the who. Thus, the summary hash construction does not include signatures. Note that the sender signs the latest summary hash so that receiver can verify the authenticity; however, the received signature will not be included in generating the summary hash for the next to-be-modified versions.

If the summary hash construction for the current version includes a predecessor's signature in addition to the predecessor's summary hash, we cannot capture the case when different sites have independently merged predecessors into a coincidentally identical content. Because the signature is included, the summary hashes will be different depending on who has constructed in the past, even if the merged versions are coincidentally identical.

OpenCM [44], a configuration management system based on using cryptographic hashes for naming, includes predecessors' ids in the current version's meta-data, essentially providing a tamper-evident audit trail. However, the resulting hash is too distinctive to be useful as a decentralized dependency tracking mechanism for optimistic replication. In particular, it does not capture coincidental equality. OpenCM is designed for client-server (repository server) configuration management, so the meta-data includes other information such as server-assigned serial revision number that the decentralized dependency tracking mechanism may not need. Similarly, in OceanStore [26], the meta-data of a data object includes server assigned serial version number, and timestamp in addition to the predecessor's id.

5.8 Summary

Summary Hash History (SHH) is a causal history with summary hash as version identifier. By using summary hash as version identifier, SHH can efficiently maintain revision histories that are essentially required to deter various attacks on decentralized ordering correctness.

This chapter showed how SHH prevents the attacks while remaining scalable and light-weight.

First, SHH does not use version vectors; hence, SHH is free from version vector inherent overheads such as management complexity in site addition/deletion.

Second, the secure log reconstruction property allows each site to securely reconstruct the previous history from a given summary hash, providing an audit trail to deter various attacks decentralized ordering correctness. Thus, SHH allows aggressive pruning of SHH. Also, SHH can efficiently support sites with limited storage. The secure log reconstruction property allows sites with limited storage to hold a limited number of recent summary hashes while more substantive sites can be configured to hold the complete history.

Third, the recursive construction of SHH includes only the predecessor's summary hash, not a signature, or other meta-information such as modification time. Thus, SHH can capture the coincidental equality precisely, which is particularly useful feature in supporting weakly consistent replication for network-partitioned collaborations. This chapter showed that the replica can even converge across partitioned networks using SHH.

We implemented SHH and simulated the attacks and faults to better understand SHH properties. The performance evaluation demonstrated that the ordering determination is fast with the time to merge two diverged histories increasing linearly with the size of histories. The simulation results show that the number of sites receiving incorrect data increases *exponentially* as the number of "same version id" attackers increases. Also, we ran a stress test where some nodes were configured to have only the latest summary hash as in small mobile device. This simulation show that the number of occasions requiring the secure log reconstruction can be kept quite low – below 1.6 per 100 anti-entropy cycle, if there are enough anti-entropy cycles between updates.

Chapter 6

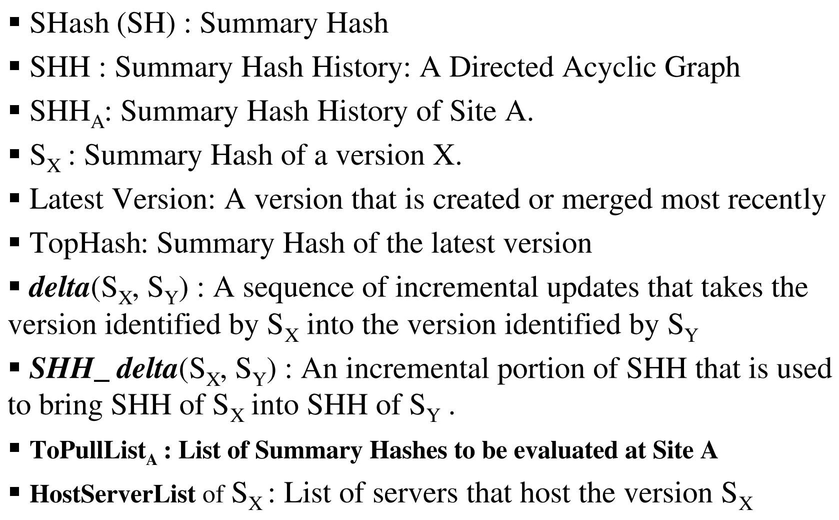
SHH-based Reconciliation over the Network

During reconciliation, each site needs to exchange the SHHs over the network. Being based on causal history, the size of SHH is unbounded, which can seriously limit the scalability in terms of network bandwidth consumption. This chapter describes how SHH solves this problem using "two-step reconciliation", taking advantage of SHH's secure construction property.

6.1 Basic Pull SHH Reconciliation

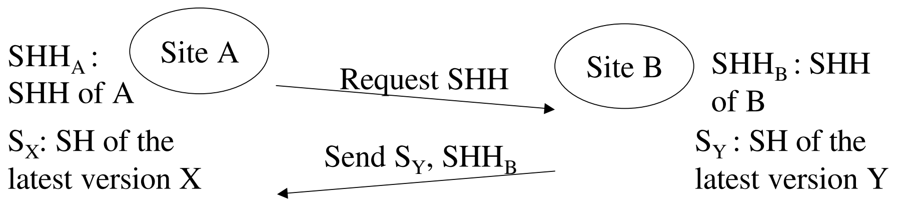
In the following sections, we describe SHH-based reconciliation protocols. Figure 6.1 lists the definitions and acronyms that we use to describe various protocols.

Figure 6.3 shows a pseudo code for basic pull SHH reconciliation between Site A and Site B.



- SHash (SH) : Summary Hash
- SHH : Summary Hash History: A Directed Acyclic Graph
- SHH_A: Summary Hash History of Site A.
- S_X : Summary Hash of a version X.
- Latest Version: A version that is created or merged most recently
- TopHash: Summary Hash of the latest version
- ***delta***(S_X, S_Y) : A sequence of incremental updates that takes the version identified by S_X into the version identified by S_Y
- ***SHH_ delta***(S_X, S_Y) : An incremental portion of SHH that is used to bring SHH of S_X into SHH of S_Y .
- **$ToPullList_A$: List of Summary Hashes to be evaluated at Site A**
- **HostServerList** of S_X : List of servers that host the version S_X

Figure 6.1: Definitions and Acronyms that are used for describing SHH reconciliation protocols.



Dominance Check at Site A:

Case 0: S_X equals S_Y => Stop

Case 1: S_Y appears in SHH_A => Stop

Case 2: S_X appears in SHH_B => Request ***delta***(S_X, S_Y)

Case 3: Neither 0, 1, 2. => Find a common ancestor (say S_Z) of SHH_A and SHH_B. Then, Request ***delta***(S_Z, S_Y).

Then, Site A ***may(or should)*** merge ***delta***(S_Z, S_X) and ***delta***(S_Z, S_Y).

Figure 6.2: Basic Pull SHH Reconciliation: A basic protocol where SHH is first sent over to Site A to check the dominance relations between the latest versions of the two sites.

```
BasicPullReconciliationSHH (ObjectID objID, Site SiteB)  at SiteA {
//From SiteA with shhA as summary hash history
 1.  Make connection to SiteB with objID,
 2.  shhB ← RequestSHH(SiteB, objID)
 3.  If (shhB.TopHash = = shhA.TopHash) { Case 0: Stop. }
 4.  Else if  (AppearIn (shhB.TopHash, shhA)) { Case 1: Stop. //A dominates B}
 5.  Else if  (AppearIn (shhA.TopHash, shhB)) { Case 2: // B dominates A
 6.          d1 ← RequestDelta(SiteB, objID, shhA.TopHash, shhB.TopHash)
 7.          ApplyDelta(objID, d1)
     }
     Else { Case 3: //Neither dominates the other
 8.          ca ← FindCommonAncestor(shhA, shhB)
 9.          d1 ← RequestDelta(SiteB, objID, ca, shhB.TopHash)
10.          d2 ← ComputeDelta(objID, ca, shhA.TopHash)
11.          ApplicationSpecificMergeProc (d1, d2)
     }
}

RumorAgentBasicPullSHH (Owner owner)  at SiteB {
1.   while (true) { //wait for requests from other site
2.           case "RequestSHH(objID)": send objID.SHH
3.           case "RequestSHHDelta(objID,fromID,toID)": send ComputeSHHDelta(objID,fromID,toID)
4.           case "RequestDelta(objID,fromID,toID)": send ComputeDelta(objID,fromID,toID)
     }
}
```

Figure 6.3: Pseudo Code for Basic Pull SHH Reconciliation: Site A pulls the data from Site B. BasicPullReconciliationSHH() routine at Site A first requests summary hash history from Site B at line 2. Once it receives the shhB, it determines the dominance between **shhA** and **shhB**. In the event of **shhB** dominates **shhA** (line 5-7), it requests delta that brings **shhA**'s TopHash into **shhB**'s. In the event of conflict (line 8-10), a common ancestor, **ca**, can be found and used as a **fromID** in requesting delta. Meanwhile, at Site B, the RumorAgentBasicPullSHH() procedure is running as a server to reply to the request of delta and SHH from the pulling site, Site A.

```
SHash shhA.findCommonAncestorBFS(SHash curID, SHH shhB, Set visited) {
1.  If (visited.contains(curID)) { return 0 }
2.  If (AppearsIn (curID, shhB)) { return curID }
3.  Put curID into visited
    //Now recursively traverse into parents
4.  ObjectArray parents ← shhA.getParents(curID)
5.  if (parents ≠ null) {
6.          for i ← 0 to parents.length-1 {
7.                  SHash found ← shhA.findCommonAncestorBFS(parents[i], shhB, visited)
8.                  if (found ≠ 0) return found   //else keep looking.
            }
    }
9.  return found
}

SHash shhA.findCommonAncestor (SHH shhA, SHH shhB) {
1.     return shhA.findCommonAncestorDFS(shhA.TopHash, shhB, new Set())
    }
}

Boolean AppearsIn (SHash curID, SHH shh) {
1.  return shh.hasKey (curID)
}
```

Figure 6.4: Pseudo Code for Finding Common Ancestor: The findCommonAncestorBFS() routine visits every item in shhA using breadth first traversal. It checks whether or not the item appears in the shhB. If so, it returns the item as a common ancestor. Otherwise, it keeps traversing the shhA until it finds a match. It is guaranteed to find a match since the initial root hash (i.e., objID) is a default common ancestor.

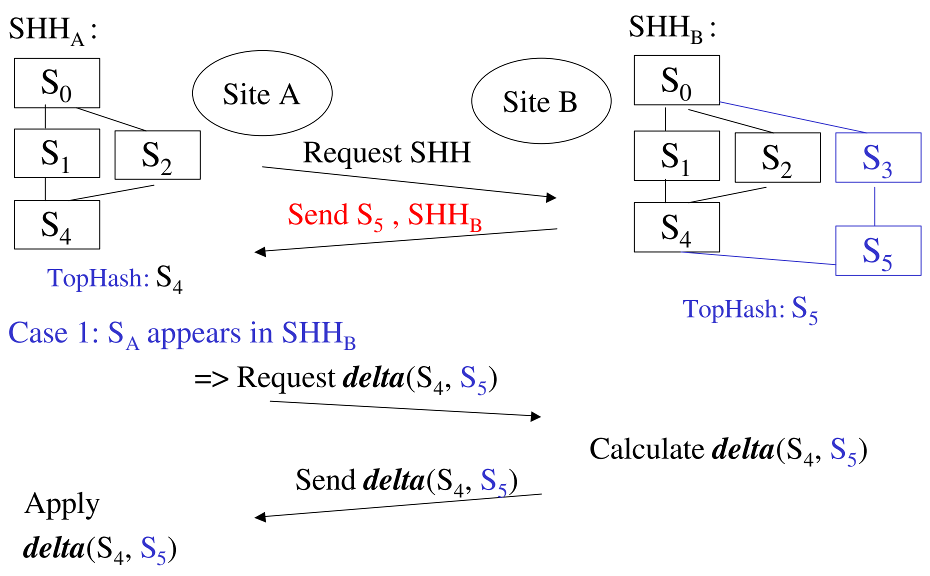


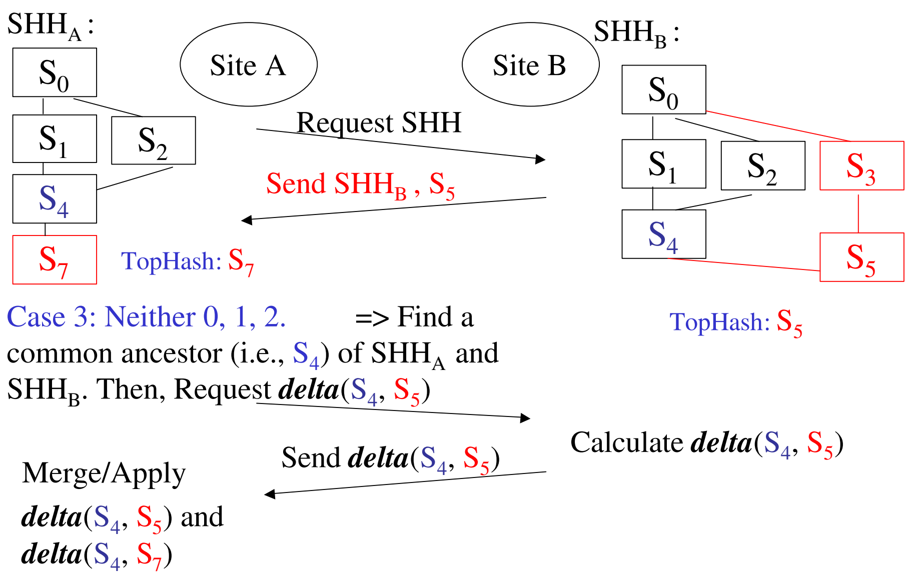
Figure 6.5: An Example of Basic Pull SHH Reconciliation with Dominance Case

Figure 6.4 shows a pseudo code for finding a common ancestor using breadth first traversal.

The basic pull protocol is described in Figure 6.2. To exchange updates, each site needs to determine the dominance relation among the latest versions. To do so, an SHH needs to be sent over to the other site so that one site can determine the dominance relation. In the basic pull method, first, the pulling site (Site A) receives the SHH from the other site and determines the version dominance by checking if Site A's latest version appears in the other's SHH or not. Figure 6.2 describes how the version dominance is determined per causal history approach.

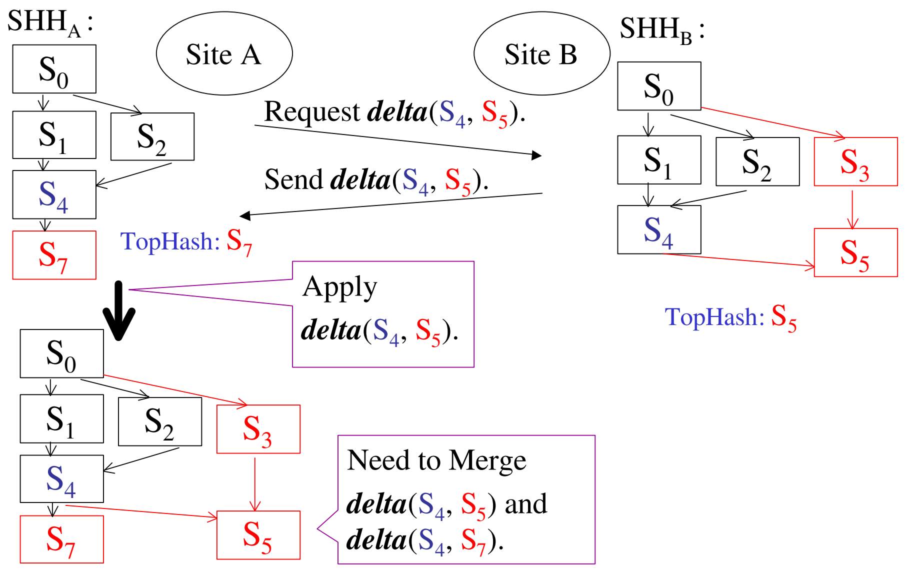
Figure 6.5 shows an example of basic pull reconciliation where Site A pulls updates from Site B since Site B's latest version S_5 dominates Site A's latest version S_4.

Figure 6.6 shows another example where Site A determines that Site A and Site



SHH_A:
S_0
S_1
S_2
S_4
S_7
Site A
Site B
SHH_B:
S_0
S_1
S_2
S_3
S_4
S_5
Request SHH
Send SHH_B , S_5
TopHash: S_7
TopHash: S_5
Case 3: Neither 0, 1, 2. => Find a common ancestor (i.e., S_4) of SHH_A and SHH_B. Then, Request ***delta***(S_4, S_5)
Calculate ***delta***(S_4, S_5)
Send ***delta***(S_4, S_5)
Merge/Apply ***delta***(S_4, S_5) and ***delta***(S_4, S_7)

Figure 6.6: An Example of Basic Pull SHH Reconciliation with Conflict Case (Before)



SHH_A:
S_0
S_1
S_2
S_4
S_7
Site A
Site B
SHH_B:
S_0
S_1
S_2
S_3
S_4
S_5
Request ***delta***(S_4, S_5).
Send ***delta***(S_4, S_5).
TopHash: S_7
TopHash: S_5
Apply ***delta***(S_4, S_5).
S_0
S_1
S_2
S_3
S_4
S_5
S_7
Need to Merge ***delta***(S_4, S_5) and ***delta***(S_4, S_7).

Figure 6.7: An Example of Basic Pull SHH Reconciliation with Conflict Case (After)

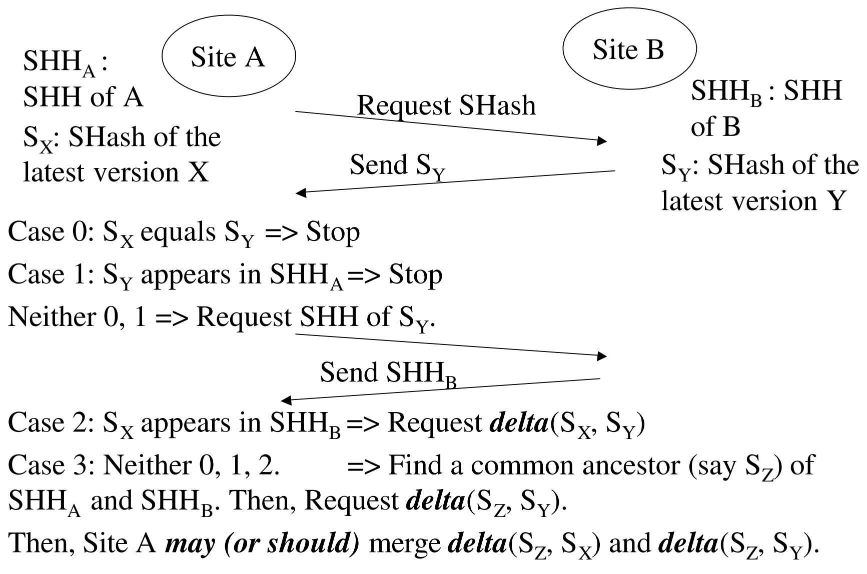

SHH_A : SHH of A
S_X: SHash of the latest version X
Site A
Site B
SHH_B : SHH of B
S_Y: SHash of the latest version Y
Request SHash
Send S_Y
Case 0: S_X equals S_Y => Stop
Case 1: S_Y appears in SHH_A => Stop
Neither 0, 1 => Request SHH of S_Y.
Send SHH_B
Case 2: S_X appears in SHH_B => Request ***delta***(S_X, S_Y)
Case 3: Neither 0, 1, 2. => Find a common ancestor (say S_Z) of SHH_A and SHH_B. Then, Request ***delta***(S_Z, S_Y).
Then, Site A ***may (or should)*** merge ***delta***(S_Z, S_X) and ***delta***(S_Z, S_Y).

Figure 6.8: Light Pull SHH Reconciliation Variation 1: A protocol improved upon the basic protocol. Instead of requesting SHH for every dominance determination, in this protocol, Site A requests the latest summary hash (SHash) of Site B. Site A requests SHH of Site B only when the Site A's latest SHash is not equal to Site B's latest SHash and Site B's latest SHash does not appear in SHH of Site A.

B are in conflict, and finds a common ancestor between SHH of Site A and SHH of Site B, and pulls updates from Site B based on the common ancestor. Once the delta from Site B is received, Site A applies the delta and merge it with Site A's latest version. Figure 6.7 illustrates how the conflicts can be merged at Site A.

6.2 Light Pull SHH Reconciliation

As the reader may have noticed, the SHH can be requested more sparingly. In particular, the pulling site would not require the complete SHH from the other site to check if the pulling site's latest version is the same as the one at the other site or if the pulling site's latest version dominates the latest version of the other site. Fortunately, a pulling

```
LightPullReconciliationSHHv1 (ObjectID objID, Site SiteB) at SiteA {
//From SiteA with shhA as summary hash history
 1. Make connection to SiteB with objID, send RequestTopHash(objID)
 2. If (shhB.TopHash = = shhA.TopHash) { Case 0: Stop. }
 3. Else if (AppearIn (shhB.TopHash, shhA)) { Case 1: Stop. //A dominates B }
 4. Else { shhB ← RequestSHH(SiteB, objID) //Now request sshB from Site B
 5.         if (AppearIn (shhA.TopHash, shhB)) { Case 2: // B dominates A
 6.                 d1 ← RequestDelta(SiteB, objID, shhA.TopHash, shhB.TopHash)
 7.                 ApplyDelta(objID, d1)
            }
            Else { Case 3: //Neither dominates the other
 8.                 ca ← FindCommonAncestor(shhA, shhB)
 9.                 d1 ← RequestDelta(SiteB, objID, ca, shhB.TopHash)
10.                 d2 ← ComputeDelta(objID, ca, shhA.TopHash)
11.                 ApplicationSpecificMergeProc (d1, d2)     }}
}

RumorAgentBasicPullSHHv1 (Owner owner) at SiteB {
1. while (true) { //wait for requests from other site
2.         case "RequestSHH(objID)": send objID.SHH
3.         case "RequestSHHDelta(objID,fromID,toID)": send ComputeSHHDelta(objID,fromID,toID)
4.         case "RequestDelta(objID,fromID,toID)": send ComputeDelta(objID,fromID,toID)
5.         case "RequestTopHash(objID)": send objID.TopHash //return 20 byte top hash. }
}
```

Figure 6.9: Pseudo Code for Light Pull SHH Reconciliation (Variation 1): Site A pulls the data from Site B. LightPullReconciliationSHHv1() routine at Site A first requests the TopHash (shhB.TopHash) from Site B at line 1 - 2. Once it receives the **shhB.TopHash**, it checks if **shhB.TopHash** appears in **shhA**. If not, then, it requests **shhB** of Site B (line 4) to figure out the dominance further. In the event that **shhB** dominates **shhA**(line 5-7), it requests delta that brings **shhA**'s TopHash into **shhB**'s. In the event of conflict (line 8-11), a common ancestor, **ca**, can be found and used as a **fromID** in requesting delta. Meanwhile, at Site B, the RumorAgentBasicPullSHHv1() procedure is running as a server to reply to the request of delta and SHH from the pulling site, Site A.

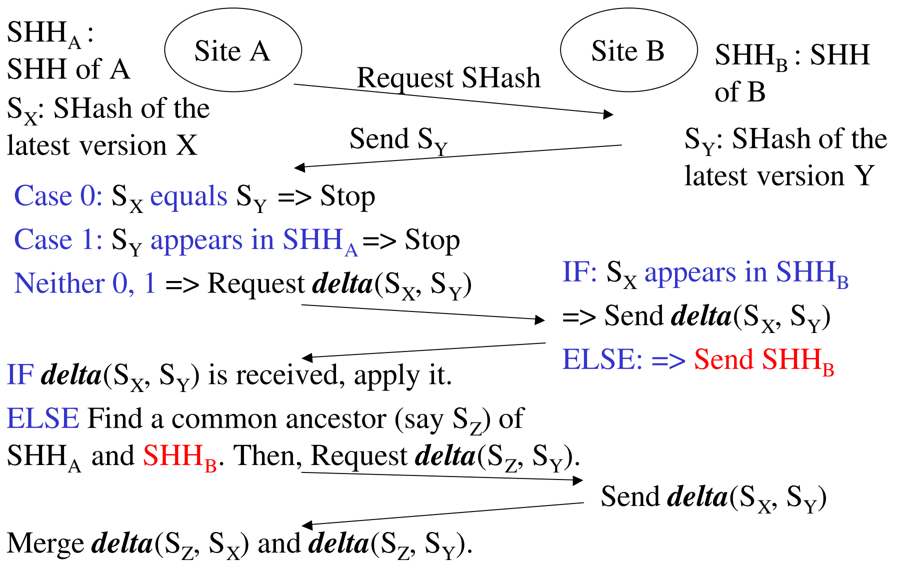

SHH_A : SHH of A
S_X: SHash of the latest version X
Site A
Site B
SHH_B : SHH of B
S_Y: SHash of the latest version Y
Request SHash
Send S_Y
Case 0: S_X equals S_Y => Stop
Case 1: S_Y appears in SHH_A => Stop
Neither 0, 1 => Request ***delta***(S_X, S_Y)
IF: S_X appears in SHH_B
=> Send ***delta***(S_X, S_Y)
ELSE: => Send SHH_B
IF ***delta***(S_X, S_Y) is received, apply it.
ELSE Find a common ancestor (say S_Z) of SHH_A and SHH_B. Then, Request ***delta***(S_Z, S_Y).
Send ***delta***(S_X, S_Y)
Merge ***delta***(S_Z, S_X) and ***delta***(S_Z, S_Y).

Figure 6.10: Light Pull SHH Reconciliation Variation 2: A protocol improved upon the light protocol variation 1. Site A requests the latest summary hash (SHash) of Site B as light pull variation 1 does. However, unlike variation 1, Site A does not request SHH of Site B when the Site A's latest SHash is not equal to Site B's latest SHash and Site B's latest SHash does not appear in SHH of Site A. Instead, Site A requests delta from Site A's latest hash to Site B's latest hash. And Site B determines if Site A's latest hash appears in Site B's SHH. If so, Site B returns the requested delta. If not, Site B sends Site B's SHH assuming Site A and Site B are in conflict. In this protocol variation 2, SHH is sent over the network only when neither site's latest SHash dominates the other's latest SHash.

```
LightPullReconciliationSHHv2 (ObjectID objID, Site SiteB)  at SiteA {
//From SiteA with shhA as summary hash history
 1.  Make connection to SiteB with objID,  send RequestTopHash(objID)
 2.  If (shhB.TopHash = = shhA.TopHash) {       Case 0: Stop. }
 3.  Else if  (AppearIn (shhB.TopHash, shhA)) { Case 1: Stop. //A dominates B}
 4.  Else { result ← RequestDeltaOrSHH(SiteB, objID, shhA.TopHash, shhB.TopHash)
 5.            if (result is Delta) {   d1 ← result       Case 2: // B dominates A
 6.                                     ApplyDelta(objID, d1)                }
 7.            if (result is SHH)  {    shhB ← result   Case 3: //Neither dominates the other
 8.                                     ca ← FindCommonAncestor(shhA, shhB)
 9.                                     d1 ← RequestDelta(SiteB, objID, ca, shhB.TopHash)
10.                                     d2 ← ComputeDelta(objID, ca, shhA.TopHash)
11.                                     ApplicationSpecificMergeProc (d1, d2)   }
    }}
RumorAgentBasicPullSHHv2 (Owner owner)  at SiteB {
1.  while (true) { //wait for requests from other site
2.   case "RequestSHH(objID)": send objID.SHH
3.   case "RequestSHHDelta(objID,fromID,toID)": send ComputeSHHDelta(objID,fromID,toID)
4.   case "RequestDeltaOrSHH(objID,fromID,toID)":
5.            if (AppearIn (fromID, shhB)) { send ComputeDelta(objID, fromID, toID)  }
6.            else {send shhB  }
7.   case "RequestDelta(objID,fromID,toID)": send ComputeDelta(objID,fromID,toID)
8.   case "RequestTopHash(objID)": send objID.TopHash  //return 20 byte top hash.
   }}
```

Figure 6.11: Pseudo Code for Light Pull SHH Reconciliation (Variation 2): Site A pulls the data from Site B. LightPullReconciliationSHHv2() routine at Site A first requests the TopHash (shhB.TopHash) from Site B at line 1 - 2. Once it receives the **shhB.TopHash**, it checks if **shhB.TopHash** appears in **shhA**. If not, then, it send DeltaOrSHH request, unlike variation 1 that requests **shhB** of Site B (line 4). Upon this request, Site B's RumorAgentBasicPullSHHv2 routine returns delta that brings **shhA.TopHash** into **shhB.TopHash**), if Site A's TopHash appears in Site B. Otherwise, it returns **shhB**, which signifies the event of conflict (line 7-11). Site A, then, finds a common ancestor, **ca**, and uses it as a **fromID** in requesting delta. Meanwhile, at Site B, the RumorAgentBasicPullSHHv2() procedure is running as a server to reply to the request of delta and SHH from the pulling site, Site A.

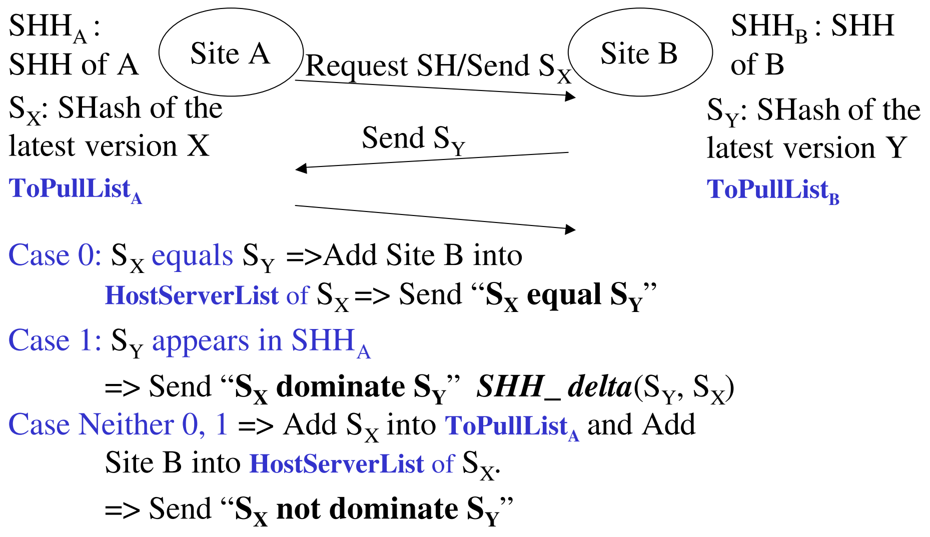
site can determine both of these cases by just checking whether the latest summary hash from other site appears in the pulling site's SHH. Hence, we introduce a more advanced protocol called *light pull SHH reconciliation.* Variation 1 of this protocol is shown in Figure 6.8. In variation 1, instead of exchanging all the SHH in every reconciliation, only the latest summary hash is exchanged first. The SHH from other site is requested only when the pulling site's latest version does not dominate the other site's latest version. In other words, the SHH is requested only when either the initiating site's latest version is dominated by the other site's or both sites are in conflict. Figure 6.9 shows a pseudo code for light pull SHH reconciliation (variation 1) between Site A and Site B.

Figure 6.10 describes another light pull variation, which is an improvement protocol upon variation 1. In variation 2, SHH is sent over to the pulling site only when both sites are in conflict. If the pulling site's latest version is dominated by the other site's, SHH is not sent over since such a determination can be performed at the other site (i.e., the being-pulled site). The SHH is sent over to the pulling site only when neither latest version dominates the other. Thus, with variation 2, the chance of sending SHH over the network is less than that of the variation 1.

Figure 6.11 shows a pseudo code for light pull SHH reconciliation (variation 2) between Site A and Site B.

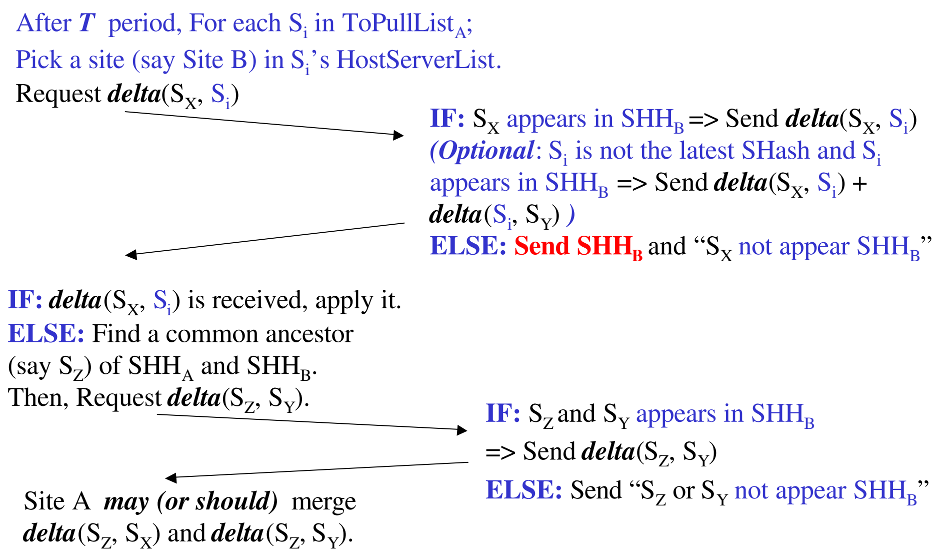
6.3 Two-Step SHH Reconciliation

Now we describe the two-step SHH reconciliation, which is an improved protocol based upon the light pull reconciliation variation 2. In the two-step SHH reconciliation



SHH_A : SHH of A
Site A
Request SH/Send S_X
Site B
SHH_B : SHH of B
S_X: SHash of the latest version X
Send S_Y
S_Y: SHash of the latest version Y
$ToPullList_A$
$ToPullList_B$
Case 0: S_X equals S_Y =>Add Site B into HostServerList of S_X => Send "S_X equal S_Y"
Case 1: S_Y appears in SHH_A => Send "S_X dominate S_Y" SHH_ delta(S_Y, S_X)
Case Neither 0, 1 => Add S_X into $ToPullList_A$ and Add Site B into HostServerList of S_X.
=> Send "S_X not dominate S_Y"

Figure 6.12: The First Step in Two-Step SHH Reconciliation (Top SHash Exchange)



After T period, For each S_i in $ToPullList_A$;
Pick a site (say Site B) in S_i's HostServerList.
Request delta(S_X, S_i)
IF: S_X appears in SHH_B => Send delta(S_X, S_i)
(Optional: S_i is not the latest SHash and S_i appears in SHH_B => Send delta(S_X, S_i) + delta(S_i, S_Y))
ELSE: Send SHH_B and "S_X not appear SHH_B"
IF: delta(S_X, S_i) is received, apply it.
ELSE: Find a common ancestor (say S_Z) of SHH_A and SHH_B.
Then, Request delta(S_Z, S_Y).
IF: S_Z and S_Y appears in SHH_B
=> Send delta(S_Z, S_Y)
ELSE: Send "S_Z or S_Y not appear SHH_B"
Site A may (or should) merge delta(S_Z, S_X) and delta(S_Z, S_Y).

Figure 6.13: The Second Step in Two-Step SHH Reconciliation (Top SHash Evaluation)

```
TopHashExchangeSHH (ObjectID objID, Site SiteB)  at SiteA  {// SiteA has shhA as SHH, ToPullListA
 1.  Make connection to SiteB with objID,  send RequestTopHash(objID), shhA.TopHash  //send SiteA's TopHash too.
 2.  If (shhB.TopHash = = shhA.TopHash) {        ToPullListA.add(shhB.TopHash)       //Case 0:
 3.                                              HostServerList(shhB.TopHash).add(SiteB)}
 4.  Else if  (AppearIn (shhB.TopHash, shhA)) {Stop. }          //Case 1: A dominates B
 }
TopHashEvaluationSHH (ObjectID objID)  at SiteA  {// SiteA has shhA as SHH
 1.  hashToPull ← PickHashToPull(ToPullListA)
 2.  hostServerList ← getHostServerList(hashToPull)  //get the HostServerList of hashToPull
 3.  site ← PickSiteToPullFrom(hostServerList)  //for example, SiteB can be picked
 4.  result ← RequestDeltaOrSHH( site, objID, shhA.TopHash, hashToPull )
 5.            if (result is Delta) {      d1 ← result                //Case 2: B dominates A
 6.                                        ApplyDelta(objID, d1)   }
 7.            if (result is SHH)  {       shhB ← result             //Case 3: Neither dominates the other
 8.                                        ca ← FindCommonAncestor(shhA, shhB)
 9.                                        d1 ← RequestDelta(site, objID, ca, hashToPull)
10.                                        d2 ← ComputeDelta(objID, ca, shhA.TopHash)
11.                                        ApplicationSpecificMergeProc (d1, d2)  }
 }
RumorAgentTwoStepSHHv2 (Owner owner)  at SiteB  { // SiteB has shhB as SHH, ToPullListB
1.  while (true) { //wait for requests from other site
2.    case "RequestSHH(objID)": send objID.SHH
3.    case "RequestSHHDelta(objID,fromID,toID)": send ComputeSHHDelta(objID,fromID,toID)
4.    case "RequestDeltaOrSHH(objID,fromID,toID)":
5.            if (AppearIn (fromID, shhB)) { send ComputeDelta(objID, fromID, toID)  }
6.            else {send shhB  }
6.    case "RequestDelta(objID,fromID,toID)": send ComputeDelta(objID,fromID,toID)
7.    case "RequestTopHash(objID), shhA.TopHash": send objID.TopHash   //i.e., shhB.TopHash
                    ToPullListB.add(shhA.TopHash)  HostServerList(shhA.TopHash).add(SiteA)       }}
```

Figure 6.14: Pseudo Codes for Two Step SHH Reconciliation: Site A pulls the data from **site** that is picked at Step 2 TopHashEvaluationSHH() (line 1-3). TopHashExchangeSHH() routine at Site A first requests the TopHash (shhB.TopHash) from Site B at line 1 - 2. Once it receives the **shhB.TopHash**, it stores **shhB.TopHash** into **ToPullListA** and it also stores **SiteB** into HostServerList of **shhB.TopHash**. Later, TopHashEvaluationSHH() picks a TopHash to evaluate from **ToPullListA**, and it also pick a **site** to pull the data from (line 1-3). Then, it send DeltaOrSHH request at line 4. Upon this request, Site B's RumorAgentTwoStepSHHv2() routine returns delta that brings **shhA.TopHash** into **shhB.TopHash**), if Site A's TopHash (**shhA.TopHash**) appears in Site B's SHH (**shhB**). Otherwise, it returns **shhB**, which signifies the event of conflict (line 4-6). Site A, then, finds a common ancestor, **ca**, and uses it as a **fromID** in requesting delta (line 7-11). Meanwhile, at Site B, the RumorAgentTwoStepSHHv2() procedure is running as a server to reply to the request of delta and SHH from the pulling site, Site A.

protocol, to achieve more efficient bandwidth consumption and to have more optimization possibilities, we separate the reconciliation process into two steps: (i) frequent exchange of top summary hash and (ii) lazy selective pull of data, logs, and summary hash history.

The exchange of the top summary hash is performed frequently and periodically. Since the size of summary hash is only 20 byte (for SHA-1), much smaller than the data, random selection would not saturate the bottleneck bandwidth, but propagates the news of new data fast. This first step is illustrated in Figure 6.12. Note that, in practice, it may be more advantageous to send more summary hashes (i.e., some recent SHH) in the first step, since the single top hash takes up only 20 byte and the default packet size for a physical network protocol may be 512 byte. In such case, we can piggy back around 25 more summary hashes into the same packet with no additional cost.

The data pull is lazily performed. Since we collect the information of who has data, we can selectively contact a site. For example, if the site is a computationally weak device that has low bandwidth connection, we could avoid such a site in the data pull process. This second step is illustrated in Figure 6.13.

We now describe the two step reconciliation procedure in detail.

Frequent Top Hash Exchange (Step 1): Periodically, given an object ID, two sites exchange the top summary hash (the latest in SHH). For each shared object, each site maintains a *ToPullList* that keeps list of summary hashes that are not yet pulled and the corresponding *HostServerList*, that is, the list of data sources from which the site can pull the data.

Given the other's top hash, each site can decide whether the other's top hash appears in its own summary hash history that is maintained locally.

- If the other's top hash is not in the SHH, each site assumes that the other has (or has received a hash of) a concurrent or recent revision that each site needs to pull later. Each site adds the other's top hash into its own ToPullList and records the other's IP address into the HostServerList of the other's top hash.
- If the other's top hash appears in the SHH, do nothing.

Lazy Selective Data Pull (Step 2): At a pre-determined interval, each site picks the latest summary hash from its ToPullList. By picking the latest summary hash, each site may not have to pull data and SHH for other hashes. For example, if there are three concurrent revisions and one of them merges the other two, then each site does not need to pull other concurrent hashes by pulling the revision that merged the other two first. Each site may not know the relations among concurrent revisions until it pulls their associated summary hash histories. Each site simply picks the best candidate revision based on arrival order as a hint.

After picking the summary hash, each site picks the best IP address from the HostServerList of the summary hash. The selection policy can be a random selection, a local IP address sub-matching, or based on previous delay or available bandwidth.

Figure 6.14 shows pseudo codes for TopHashExchange (Step 1) and TopHashEvaluation (Step 2) that we just described above.

Figure 6.15 shows an example of the first step: Frequent Top SHash (TopHash) Exchanges. Figure 6.16 shows an example of the second step where the dominance relation was determined and the data is being pulled into the pulling site. Figure 6.17 shows an

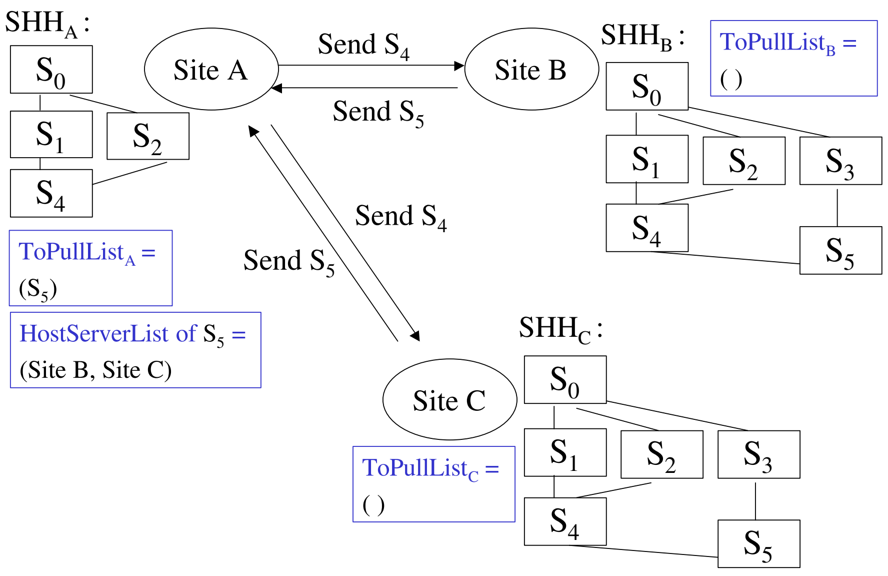
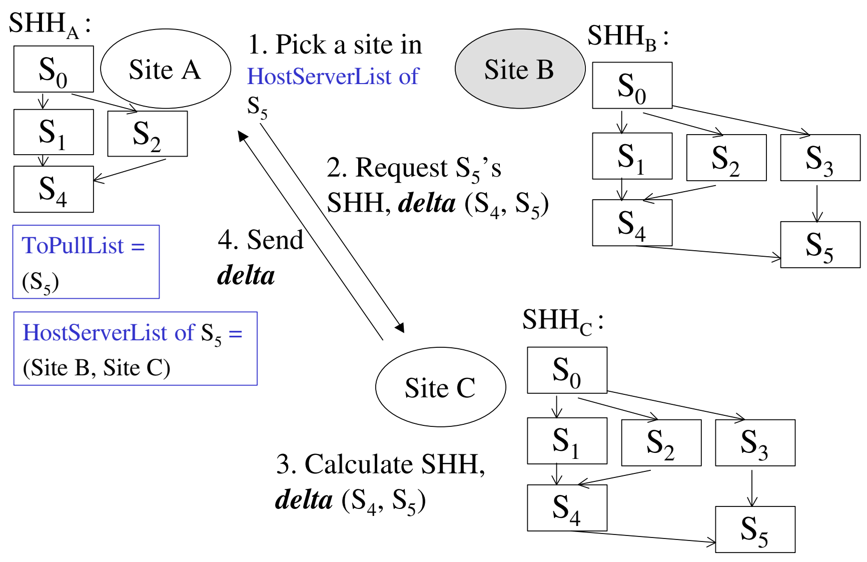
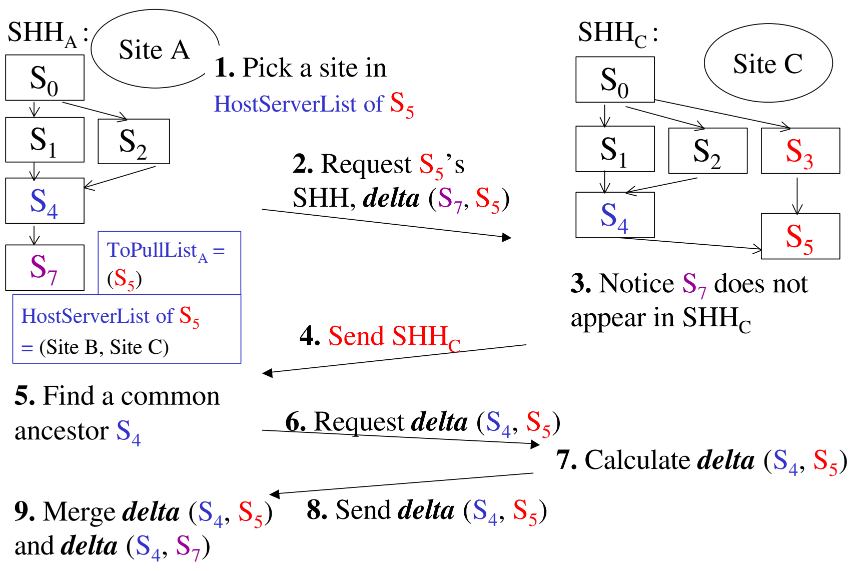


Figure 6.15: An Example of First Step in Two Step SHH Reconciliation (Top SHash Exchange)



SHH$_A$:
Site A
S_0
S_1
S_2
S_4
ToPullList = (S_5)
HostServerList of S_5 = (Site B, Site C)
1. Pick a site in HostServerList of S_5
Site B
SHH$_B$:
S_0
S_1
S_2
S_3
S_4
S_5
2. Request S_5's SHH, ***delta*** (S_4, S_5)
4. Send ***delta***
SHH$_C$:
Site C
S_0
S_1
S_2
S_3
S_4
S_5
3. Calculate SHH, ***delta*** (S_4, S_5)

Figure 6.16: An Example of Second Step in Two Step SHH Reconciliation (Top SHash Dominance Evaluation)



SHH_A :
Site A
S0
S1
S2
S4
S7
ToPullList_A = (S5)
HostServerList of S5 = (Site B, Site C)
1. Pick a site in HostServerList of S5
2. Request S5's SHH, delta (S7, S5)
SHH_C :
Site C
S0
S1
S2
S3
S4
S5
3. Notice S7 does not appear in SHH_C
4. Send SHH_C
5. Find a common ancestor S4
6. Request delta (S4, S5)
7. Calculate delta (S4, S5)
8. Send delta (S4, S5)
9. Merge delta (S4, S5) and delta (S4, S7)

Figure 6.17: An Example of Second Step in Two Step SHH Reconciliation (Top SHash Conflict Evaluation)

example of the second step where the conflict relation was determined and the data is being pulled into the pulling site and further needed to be merged by the pulling site, Site A.

6.4 Evaluation of SHH Reconciliation

Here we show a simulation result for comparing anti-entropy with version-vector and the two-step protocol with SHH.

6.4.1 Network Simulator

We implemented an event-driven simulator that performs anti-entropy reconciliation using version-vector and two-step SHH reconciliation among sites. The simulator is written in Java and the code base contains approximately 2200 statements. The main components of our simulator are **scheduler** and **event**. Event has name, time, and handler members. The scheduler dequeues the event from the head of the queue and invokes the appropriate handler attached to the event. The events are sorted by the time field in the queue. The handler implements the handle method so that it performs the requested task.

This simulator supports several physical topologies including the Transit-Stub model [55], which is generally known to reflect real internet topologies. The simulator imports the physical topology and creates a graph data structure in memory. The simulator has three layers. The bottom layer models the physical topology, the middle layer models the overlay node, and the top layer models applications. In our simulation experiments, we have only one application, so the relationship between the application and the overlay node is one-to-one. The binding between a physical node and an overlay node depends on the assignment. For example, in the Transit-Stub model, we can choose physical nodes uniformly in the network, choose them uniformly in stub domains, or choose them heavily biased in some particular domains.

6.4.2 Network Bandwidth Consumption with SHH Two-Step Reconciliation

We compare lazy selective data pull (*LSData*) with anti-entropy reconciliation of data (*AEData*) based on version vector. In *AEData*, a data sender first receives a version vector from a data receiver, and then sends the difference to the receiver. In *LSData*, sites exchange top summary hash and a list of data sources that hold the latest data; the data are pulled later, when needed, from a data source with a certain policy. The allowed policies are random, maximum-bandwidth, and minimum-delay policies. The random policy chooses a site randomly from the candidate source list. The maximum-bandwidth policy chooses a site

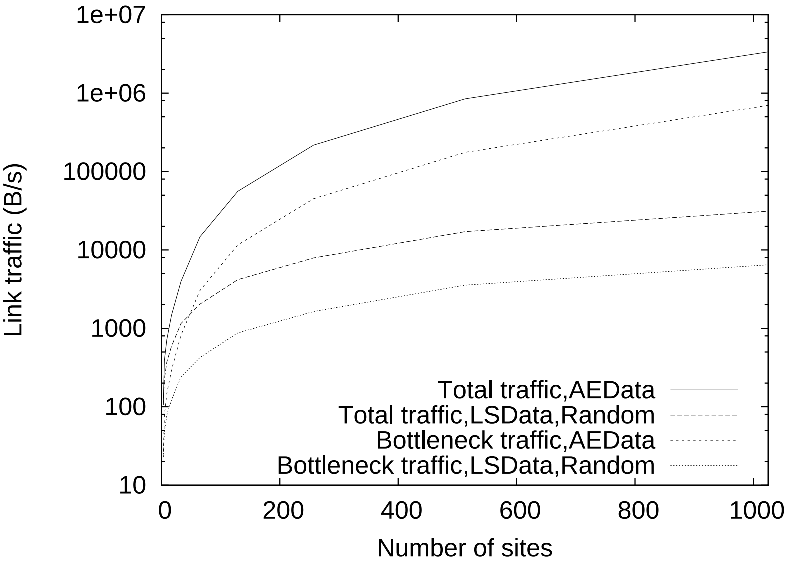


Figure 6.18: Link traffic for data size 1KB and RP 1 minute. We choose a pull period of 1 minute for *LSData*. *AEData* consumes two orders of magnitude more bandwidth than *LSData*.

that has the highest bottleneck link bandwidth. The minimum-delay policy chooses a site that has the smallest delay. The metric we use is the total and bottleneck link bandwidth consumption to compare the scalability of two schemes and to evaluate the different policies.

Figure 6.18 shows the total and bottleneck link traffic for data size of 1KB and reconciliation period (RP) of 1 minute. We varied the number of sites from 3 to 1025. In two schemes, one of the writers writes one update, and we ran our simulations until all sites receive the update. *LSData* used the random policy, and the data pull period was one minute. *AEData* consumed two orders of magnitude more bandwidth than *LSData* when the number of mergers was large. Even with 64 sites, *AEData* consumed about 5 times more bandwidth than *LSData*. In *AEData*, sending the version vector information consumed much bandwidth when the number of mergers is large, even though we used the smallest possible version vector size. Since we use a pull period of one minute, which is the same

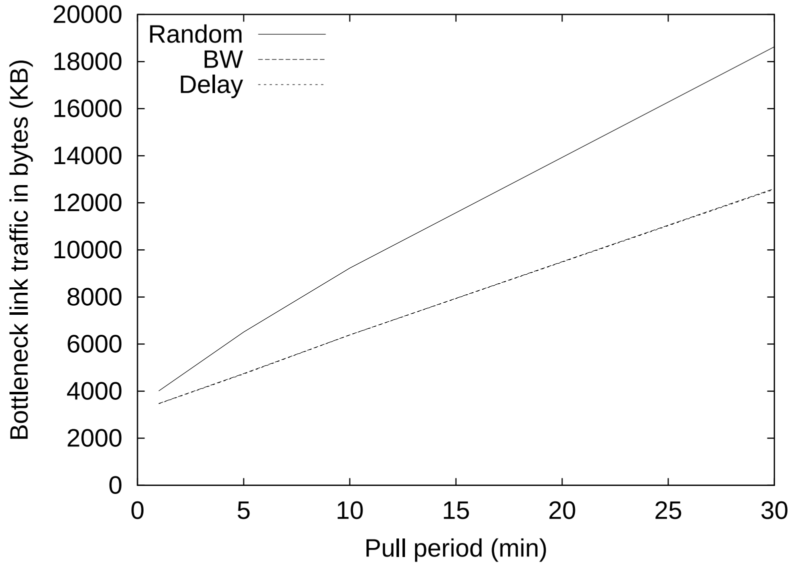


Figure 6.19: Bottleneck link traffic in bytes for data size 1KB, RP 1 minute, and 1025 sites. The bandwidth and delay-based optimizations transfer smaller total traffic in bytes than the random data pull.

as the reconciliation period, the benefit was made possible by the compact representation of metadata as the top summary hash, which is only 20 bytes for any number of writers and mergers. Since *AEData* performed worse than our base case of the lazy pull, we then focused on the performance of our *LSData* afterwards.

Figure 6.19 shows the benefit of bandwidth and delay-based data source selection in the lazy pull. We varied the pull period to be 1, 5, 10, and 30 minutes. Note that the y-axis shows the bottleneck link traffic sent in bytes until all sites receive the new data. The metric-based selection showed 32% lower traffic than the randomized selection when the pull period was 30 minute, even though the sites were randomly located in the physical network. Both optimization schemes showed similar performance.

We examined the benefit of the lazy pull as the data size increases. Figure 6.20 shows the bandwidth consumption of the minimum-delay pull policy with a data size of

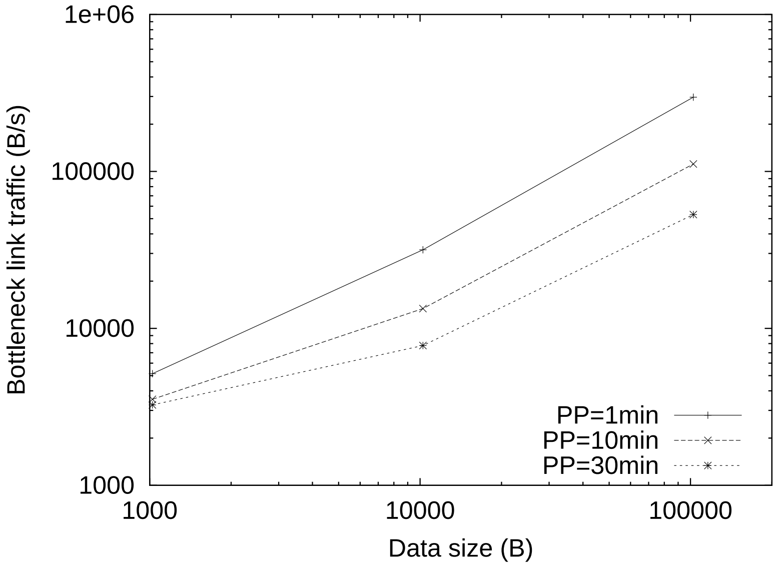


Figure 6.20: Bottleneck link traffic varying data size. Lazy data pull is used and the number of sites is 1025. Pull policy is delay-based. As the data size increases, the benefit of lazy data pull (high pull period) increases.

1KB, 10KB, and 100KB. Note that the y-axis is in log scale. When both the pull period and the data size increase, the benefit of lazy selective pull becomes large. The lazy selective pull allows the retrieval of data when it is needed. Therefore, it can exchange the data over a longer period of time incrementally, and it can choose data source sites positioned nearby, as it collects data source candidates over a longer period of time.

6.5 Related Work

The two-step SHH reconciliation protocol is similar to the general technique of servers' sending invalidation to the cached replica for them to fetch an updated version from the server. Interestingly, Pangaea disseminates updates in two steps; it sends harbinger (a meta data of an update) first to build an efficient spanning tree of fat-links with which data are pushed later. Bi-Modal multicast [8] demonstrated a reliable and scalable update dissemination using two steps: multicast for pushing updates and anti-entropy for exchanging missing update packets. SHH's two-step reconciliation protocol is different from these schemes. It pulls data lazily after pair-wise periodic exchange of hashes, and it does not rely on the multicast tree.

6.6 Summary

The size of the summary hash history that needs to be exchanged grows in proportion to number of update instances, which can limit the scalability of SHH based reconciliation. To overcome this scalability limitation, we introduced various reconciliation protocol from the basic-pull to the light-pull variation 2, incrementally optimizing the protocols to exchange SHH sparingly (only when it is necessarily required). Finally, we presented the two-step SHH reconciliation. Only the top summary hash needs to be exchanged frequently; the data/logs can be lazily and selectively pulled later from a local site, saving expensive bottleneck bandwidth.

The simulation results shows that the lazy selective data pulling of two-step SHH reconciliation consumes the network bandwidth (both total and bottleneck) in orders of magnitude lower than the traditional version-vector based anti-entropy data exchanges.

Part III

Deployment

Chapter 7

S2D2 Framework

To make SHH readily available for various applications, we developed S2D2, a framework based on SHH that provides scalable and secure optimistic replication. S2D2 serves as a substrate upon which diverse distributed applications can be readily built. This chapter describes S2D2 and its architecture, interface (API), the application specific conflict resolution, and service components such as the secure object naming and the access control. It also discusses *hash typing*, a technique to facilitate the use of hashed data structures and describes several application prototypes built on top of S2D2 framework.

7.1 S2D2 Architecture

This section presents the architecture of S2D2, the S2D2 application programming interface (API), hash typing, conflict resolution upcall, and each component of S2D2: secure object naming, access control, the secure log management, and the two-step SHH reconciliation.

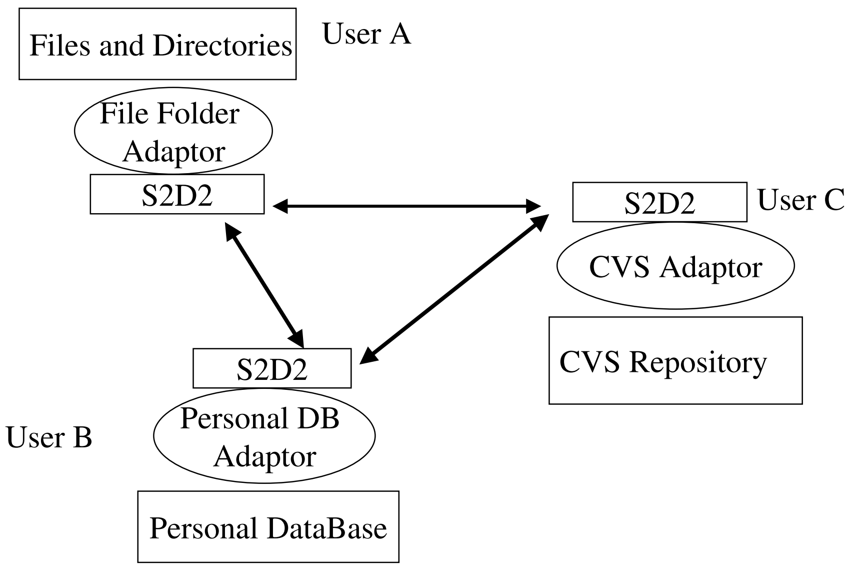


Figure 7.1: S2D2 Network Example: Through the application specific adaptors, different but compatible applications can exchange updates using S2D2. In this example, User C uses a database interface to publish and incorporate the update to the shared object, while User A uses a file system interface and User B uses a CVS (a version control system) interface to manage the shared object.

7.1.1 Architecture Overview

The S2D2 framework is designed to readily support diverse distributed applications with support for optimistic replication. It employs an adaptor architecture so that diverse applications can be readily built on top of S2D2 layer. By simply building an application specific adaptor that connects application to S2D2 functionality, various applications can exchange updates through the S2D2 framework as shown in Figure 7.1.

The S2D2's adaptor architecture is shown in Figure 7.2. It exports an API for application adaptors. Major API components are methods to publish updates and call-backs to notify changes of data. Adaptors interface applications with S2D2 and utilizes S2D2 features to implement application-specific policies, such as when to publish, when to

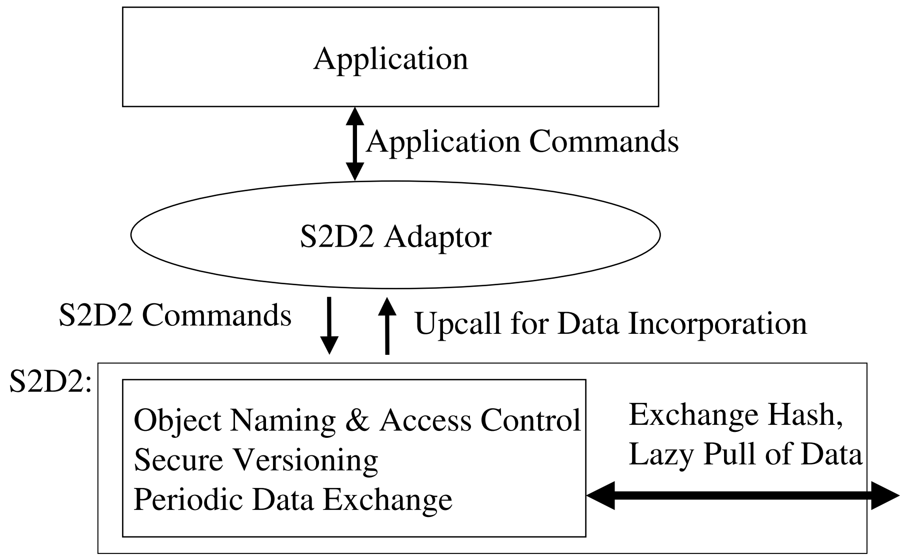


Figure 7.2: S2D2 Architecture: S2D2 is a common generic data sharing mechanism that provides global object naming, secure versioning, and scalable data exchanges. The application specific adaptor bridges the S2D2 commands to various application semantics.

incorporate, and how to merge updates. With this adaptor architecture different but compatible applications can exchange updates through S2D2. We present example adaptors in Section 7.2.

Generic handling of updates is the task of S2D2. S2D2 names objects securely using the public key of the name space owner and the object name. Object revision is securely tracked using summary hash history. The updates are exchanged by the two-step SHH reconciliation process (by default). In this process, the top summary hash is exchanged frequently to notify sites of the latest updates, the summary hash history, logs, and data are lazily pulled from sites that are selected by metrics such as delay and bandwidth. The S2D2 API also provides other basic SHH reconciliation protocols such as basic pulling and light pulling variations described in Chapter 6. Since S2D2 is based on SHH, the application adaptors can implement an aggressive log pruning management policy by utilizing the secure log management property of SHH .

S2D2 uses hash typing to be discussed below, to avoid error prone uses of various hashes and represent a structural information for a composite object.

We now describe the S2D2 API and how the adaptors can use this S2D2 API.

7.1.2 S2D2 Application Programming Interface

The most important application programming interfaces of S2D2 interface are the following. More complete API can be found in the Appendix.

Class Owner:

- owner = new Owner(keypair, keystorepasswd); // creates an **owner**
- owner.addNewObjectID(objectID); //add new objectID into **owner**'s replication control
- owner.setTwoStepPullPeriod (objectID, rumorperiod); //configure the frequency of two-step reconciliation protocol for the given objectID by setting the pulling period.
- owner.publish(objectID, local-revision-file); // creates a new revision and its summary hash under the objectID, publishes the latest summary hash and pulls the data according to the policy specified by an adaptor.
- owner.registerCallBack(adaptor); // registers its call back for incorporation. S2D2 will call the registered adaptor for data incorporation, for automatic merging, and for resolving conflicts. For example, S2D2 will call the adaptor.incorporate() for data incorporation.
- owner.addRevToHistory(objectID, file); //add new update to the objectID's revision

history that is implemented in SHH data structure. This function can be used typically after receiving a dominating update or after merging diverged versions. For example, this function is likely called from the adaptor.incorporate() routine.

- owner.getRevHistory(objectID); //get the the ObjectRevHistory of the objectID. An adaptor need to use this function to look at the previous version history of the objectID.
- owner.addWriter(objectID, writer-publickey); // adds writer to the authorized writers of the objectID. Each objectID has a default writerSet called objectID-WriterSet. The writer is specified with the writer's public key.
- owner.removeWriter(objectID, writer-publickey); // removes writer from the authorized writers of the objectID.

Class ObjectID:

- ObjectID (pubkey-of-the-common-name-creator, common-name); // creates ObjectID that is used as, a globally unique object identifier.

Class Rumor:

- Rumor (**owner**); // creates Rumor object with the **owner**.
- startRumorAgent(exchange-hash-period, pull-data-peridod); //start the Rumor Agent that exchanges summary hash at every exchange-hash-period, and pulls the data at every pull-data-period.
- setRumorPeriod(exchange-hash-period, pull-data-peridod); //set the new exchange-hash-period and the new pull-data-period for the Rumor Agent to use for perform the two-step SHH reconciliation.

An S2D2 adaptor first needs to instantiate an **owner** using the Owner class. The owner is instantiated by a public key, which is presumably owned by a principal or an entity. The public key is used as an unique identifier of the owner. The owner class defines the domain of security policy, and are required to access various S2D2 components such as managing access control, adding shared object into S2D2, and configuring two-step SHH reconciliation. For example, an adaptor can specify its rumoring policy by using "setRumorPeriod" functions provided by the Rumor class. And the Rumor class requires an instantiated Owner class at initialization. The "Rumor" object manages the periodic reconciliations based on the rumoring policy specified by the adaptor.

The owner class maintains the list of S2D2 objects. Each S2D2Object is instantiated with an ObjectID class. The objectID is composed of the concatenation of the public key of the name creator and the object name itself. For each objectID, by default, S2D2 creates the "ObjectRevHistory" class to manage the logs of revisions, summary hash history, access control list (using writerSet class), concurrent revisions, and the list of data host servers for lazy-selective pulling in the two-step SHH reconciliation. An adaptor needs to call owner.getRevHistory(objectID) to access the revision history of objectID.

7.1.3 S2D2 Service Components

Secure Global Naming for the Shared Object

An S2D2-ID is a globally unique identifier for a shared object whose updates are exchanged using S2D2 Framework. The S2D2-ID comprises of two parts: the public key of the name space owner and the object name created by this name space owner. We assume

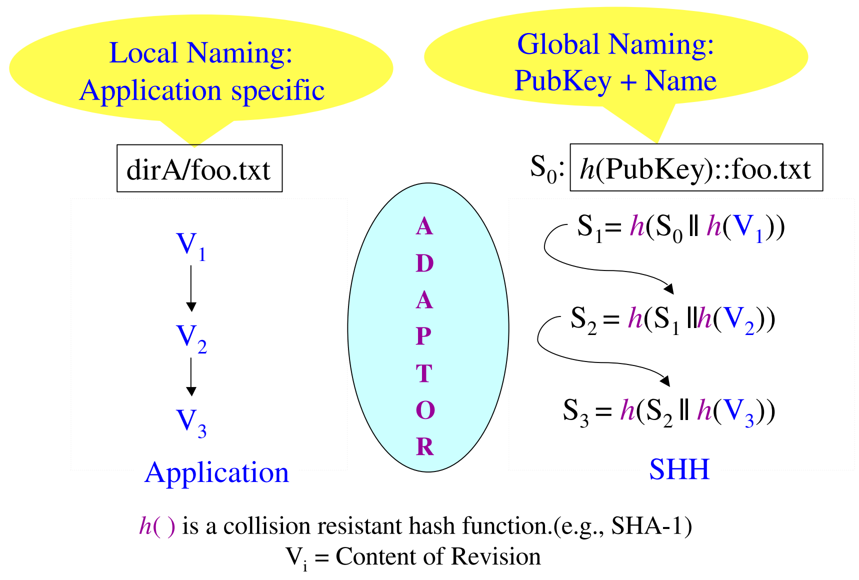


Figure 7.3: Secure Object Naming with Summary Hash History: The S2D2-ID on the right is composed of the name space owner's public key and the string "foo.txt"that is given by the name space owner. The S2D2-ID serves as the initial parent summary hash (i.e., S_0), which is included in creating the summary hash (i.e., S_1) of the first revision, so that the S2D2-ID securely refers a series of revisions. The S2D2 adaptor maintains the mappings between S2D2-IDs and the local object names, and the mappings between summary hashes and the local revisions.

the public key is unique through a Public Key Infrastructure (PKI) and the owner assigns an object name that is unique within the owner's name space. Thus, S2D2-ID is globally unique by ensuring those two assumptions hold.

The summary hash of the initial revision includes the S2D2-ID as a parent hash, so that S2D2-ID can securely refer to a series of revisions of the shared object. The adaptor maintains the mapping between each local object name and the corresponding S2D2-ID. Likewise, the adaptor can also map each local revision into a corresponding summary hash that is used as a global version identifier. An example is shown in Figure 7.3. Thus, it is readily possible for different users' applications to exchange updates by using the S2D2-ID and the SHH as a common mechanism. An example is shown in Figure 7.4.

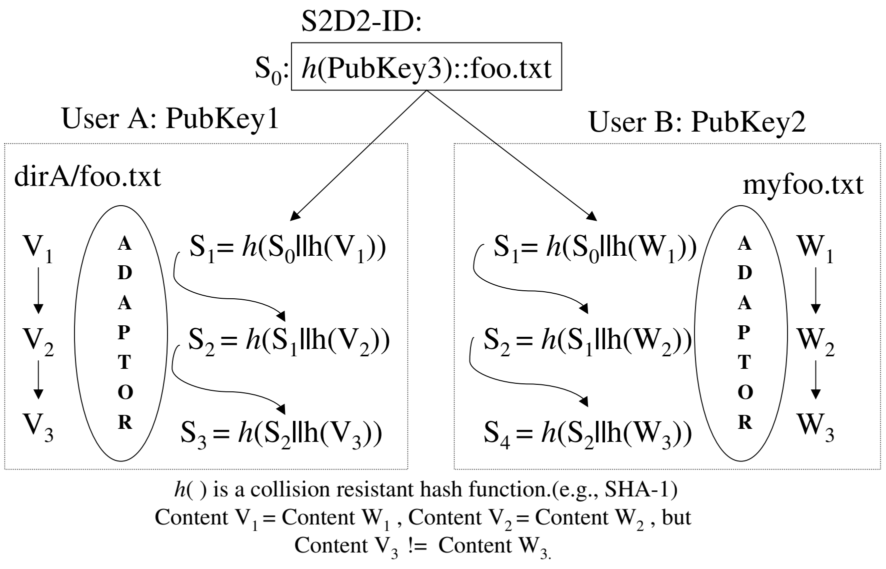


Figure 7.4: Two Different Local Object Names with The Same Global S2D2-ID: User A uses a local name "dirA/foo.txt" and User B uses a local name myfoo.txt to refer to the shared object that is globally identified as the string "h(PubKey3)::foo.txt". Each user's adaptor maintains the mapping between the global names and the local names.

Access Control to the Shared Object

S2D2 provides access control for a shared object. The name space owner's public key authenticates the Writer-Set that contains a list of authorized writers' public keys. Each revision should be signed by one of the public keys in the writer set. We can achieve such authentication by simply signing the summary hash. Each site can check the validity of the signed summary hash of the version and discard unauthorized updates. In the event a site negligently includes an unauthorized update, other sites can readily detect such event later since SHH provides an audit trail.

Our current implementation allows only the owner of the object name to authorize the Writer-Set; however, it is easy to extend the implementation to allow the owner to authorize an ACL-Manager-Set that has a list of public keys authorized to modify the

Writer-Set. In the same way, collaborators can define Reader-Set among themselves and enforce the readership at their S2D2 adaptors.

7.1.4 Hash Typing

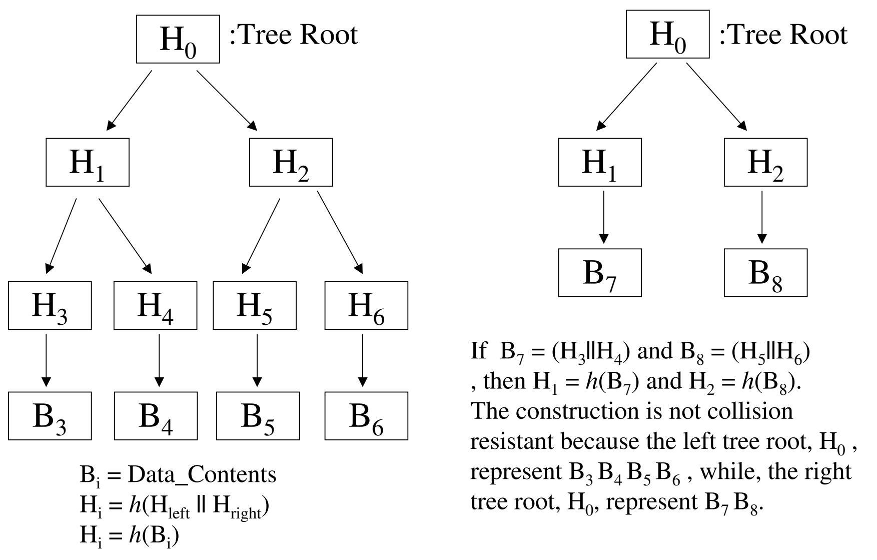
In this section, we introduce hash typing, a technique to guard against error-prone use of hashes and to identify the shared structure of the hash construction method.

In hash typing, we annotate hashes with the *hashtype* information as we annotate data with the date type information. For example, if the data, say D_0, is a "set" type, then we use ("!set!" $\|$ D_0) to annotate the data D_0. Likewise, if the hash, say H_0, is a "set-hash" type, then we use ("!set-hash!" $\|$ H_0) to annotate the hash H_0.

A hash type is a name to refer to the structure or the hash construction method. For example, if a hash, H_0, is annotated with "set-hash", this means the hash, H_0, is constructed using "set-hash" construction method. The "set-hash" can be constructed by concatenating all the set members' data by lexicographically sorted order. Or, more efficiently, the "set-hash" can be constructed by concatenating the set member's unique identifiers.

Guarding against Error-Prone Inclusion of Hashes

Hash typing eliminates the ambiguity that results from the same hash being generated from different hash construction methods. For example, it is possible that the hash from "set-hash" construction is the same as the hash from the "list-hash" construction. However, with hash typing, one can readily disambiguate ("!set-hash!" $\|$ H_0) from ("!list-hash!" $\|$ H_0).



H_0 :Tree Root
H_1
H_2
H_3
H_4
H_5
H_6
B_3
B_4
B_5
B_6
B_i = Data_Contents
$H_i = h(H_{left} \| H_{right})$
$H_i = h(B_i)$
H_0 :Tree Root
H_1
H_2
B_7
B_8
If $B_7 = (H_3 \| H_4)$ and $B_8 = (H_5 \| H_6)$, then $H_1 = h(B_7)$ and $H_2 = h(B_8)$.
The construction is not collision resistant because the left tree root, H_0, represent $B_3 B_4 B_5 B_6$, while, the right tree root, H_0, represent $B_7 B_8$.

Figure 7.5: Erroneous Use of Hashes in a Merkle Tree: Since B_7 and B_8 are data block, B_7's content can be $(H_3 \parallel H_4)$ and B_8's content can be $(H_5 \parallel H_6)$. Then, H_0 on the left represents B_3,B_4,B_5,B_6, while H_0 on the right represents B_7,B_8. The block hash is mistakenly interpreted as tree hash.

Figure 7.5 illustrates an example of the erroneous use of hashes in a Merkle Tree construction that many systems employ to authenticate the list of data blocks in a scalable manner. With Merkle tree constructed blocks, one does not have to reapply the hashing over a complete list of blocks when a change to a block happens; instead, only the affected nodes from the changed block to the root need to be hashed again [26, 18]. (The idea of including the hashes of predecessors appears in Merkle's tamper-evident hash tree [30] and hash chaining [6, 21].) However, a naive inclusion of predecessors in a hierarchical tree or in iterative hash chaining is prone to having a collision even though the hash function is collision resistant.

There have been many fixes to address this weakness [14, 51, 29, 22], for example, an additional inclusion of length of each input, adding a depth counter in recursive

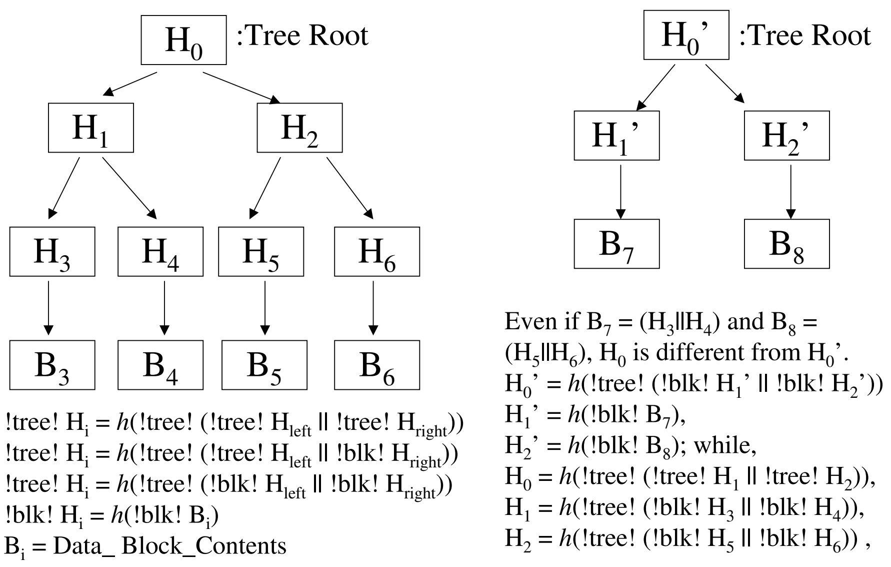

H_0 :Tree Root
H_1
H_2
H_3
H_4
H_5
H_6
B_3
B_4
B_5
B_6
!tree! $H_i = h$(!tree! (!tree! H_{left} || !tree! H_{right}))
!tree! $H_i = h$(!tree! (!tree! H_{left} || !blk! H_{right}))
!tree! $H_i = h$(!tree! (!blk! H_{left} || !blk! H_{right}))
!blk! $H_i = h$(!blk! B_i)
B_i = Data_ Block_Contents
H_0' :Tree Root
H_1'
H_2'
B_7
B_8
Even if $B_7 = (H_3||H_4)$ and $B_8 = (H_5||H_6)$, H_0 is different from H_0'.
$H_0' = h$(!tree! (!blk! H_1' || !blk! H_2'))
$H_1' = h$(!blk! B_7),
$H_2' = h$(!blk! B_8); while,
$H_0 = h$(!tree! (!tree! H_1 || !tree! H_2)),
$H_1 = h$(!tree! (!blk! H_3 || !blk! H_4)),
$H_2 = h$(!tree! (!blk! H_5 || !blk! H_6)) ,

Figure 7.6: Use of Data Typed Hash in a Merkle Tree: Even if B_7's content can be $(H_3 \parallel H_4)$ and B_8's content can be $(H_5 \parallel H_6)$, the block hash is interpreted as block hash not intermediate tree hash. Hash typing guides correct usage of hashes in various recursive construction.

construction of hash tree, or adding 0 to the block and 1 to the intermediate tree hash generation.

We found the hash typing is more general solution to avoid this problem. Figure 7.6 illustrates how the erroneous use of hashes can be avoided using the hash typing in a Merkle Tree implementation.

Representing Composite Object Hash

Hash typing is also useful to represent composite objects. A composite object is an object that is composed of related objects each of which can be referenced and used individually. For example, a directory object can be a composite object that contains file objects and directory objects. A directory object hash can be represented using a

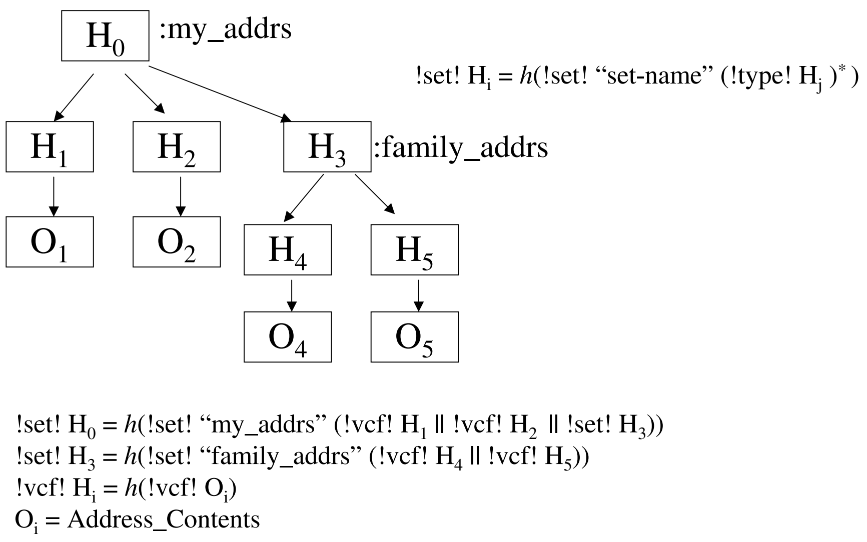

H0 :my_addrs
!set! Hi = h(!set! "set-name" (!type! Hj)*)
H1
H2
H3 :family_addrs
O1
O2
H4
H5
O4
O5
!set! H0 = h(!set! "my_addrs" (!vcf! H1 || !vcf! H2 || !set! H3))
!set! H3 = h(!set! "family_addrs" (!vcf! H4 || !vcf! H5))
!vcf! Hi = h(!vcf! Oi)
Oi = Address_Contents

Figure 7.7: Set Type Hash for a Set of Objects Example: A set named my_addrs contains two vcf (a standard format for name card) object and another set named family_addrs that contains two vcf objects O_4 and O_5.

composite object hash over concatenated hashes of the files and subdirectories that the directory contains.

We present three hash typing examples, which are important hash types of our system. Figure 7.7 shows a hash type example. Set type is used for collection of address objects. Figure 7.8 shows a summary hash type example. Figure 7.9 shows a composite hash for a file-directory example. Interestingly, the composite object's structure can be used at the S2D2 layer, so that S2D2 can exchange each element of the composite object separately. For example, foo1.txt can be fetched locally as an S2D2 object while foo2.txt needs to be fetched remotely as an S2D2 object. Otherwise, a directory needs to be zipped into a single object that has to be exchanged as a whole, as is common in email attachments. In a peer to peer distributed file system, the directory structure is a given structure that every layer in the system knows how to deal with. For handling replicated composite objects, hash typing

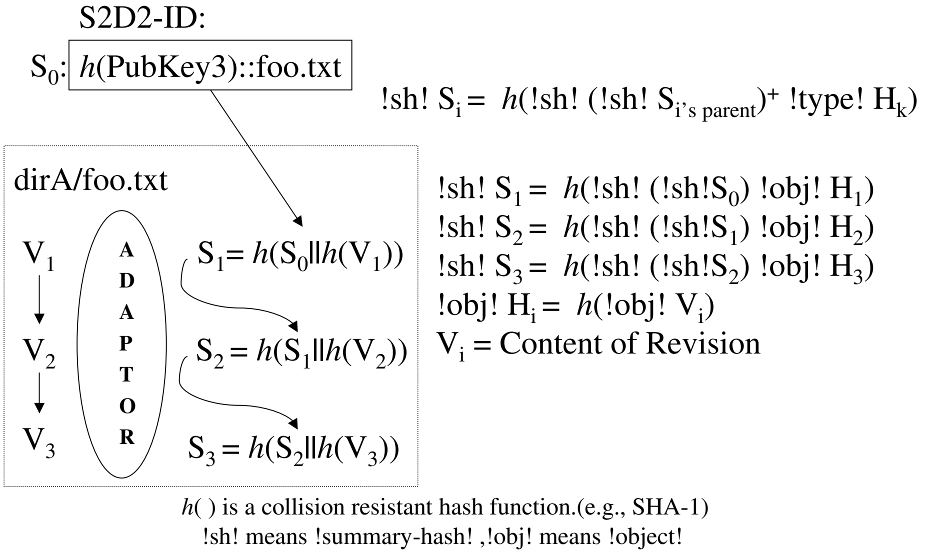

S2D2-ID:
S0: h(PubKey3)::foo.txt
!sh! Si = h(!sh! (!sh! Si's parent)+ !type! Hk)
dirA/foo.txt
!sh! S1 = h(!sh! (!sh!S0) !obj! H1)
!sh! S2 = h(!sh! (!sh!S1) !obj! H2)
!sh! S3 = h(!sh! (!sh!S2) !obj! H3)
!obj! Hi = h(!obj! Vi)
Vi = Content of Revision
V1
V2
V3
ADAPTOR
S1= h(S0||h(V1))
S2 = h(S1||h(V2))
S3 = h(S2||h(V3))
h( ) is a collision resistant hash function.(e.g., SHA-1)
!sh! means !summary-hash! ,!obj! means !object!

Figure 7.8: Summary Hash Type Example: The summary hash contains predecessor's summary hash recursively. The S2D2 ID is used as an initial predecessor. By using hash typing, the object V_i cannot be mistakenly interpreted as a summary hash.

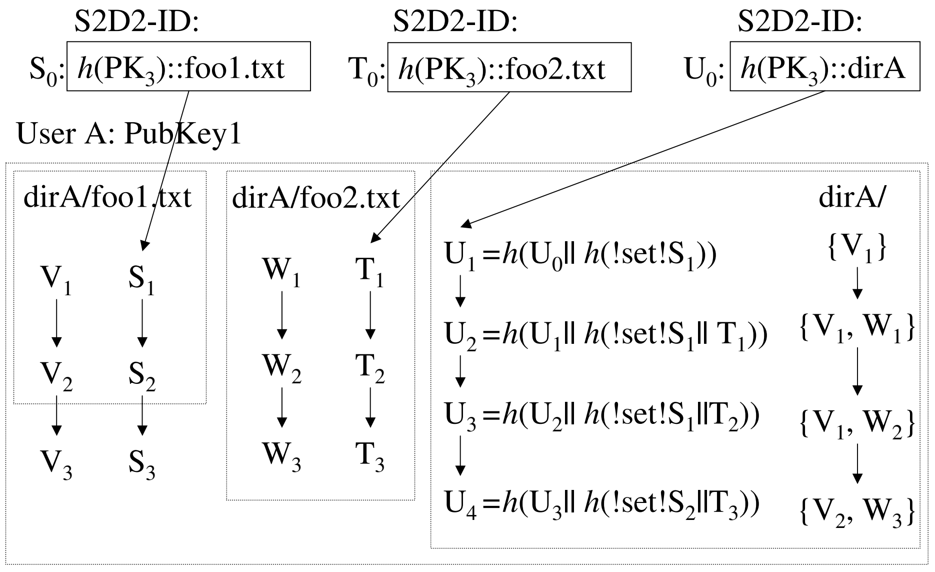


Figure 7.9: Composite Object Hash for File Directory Example: S_i is summary hash of V_i(local revisions of foo1.txt) and T_i is summary hash of W_i(local revisions of foo2.txt). Because these two files are in a directory called "dirA", a composite object hash that comprises of S_i and T_i can be built to represent the directory state. Interestingly, this directory hash can also represent the relations between foo1.txt and foo2.txt that they are in the same directory. S2D2 can use these structure and relational information for efficient data exchanges.

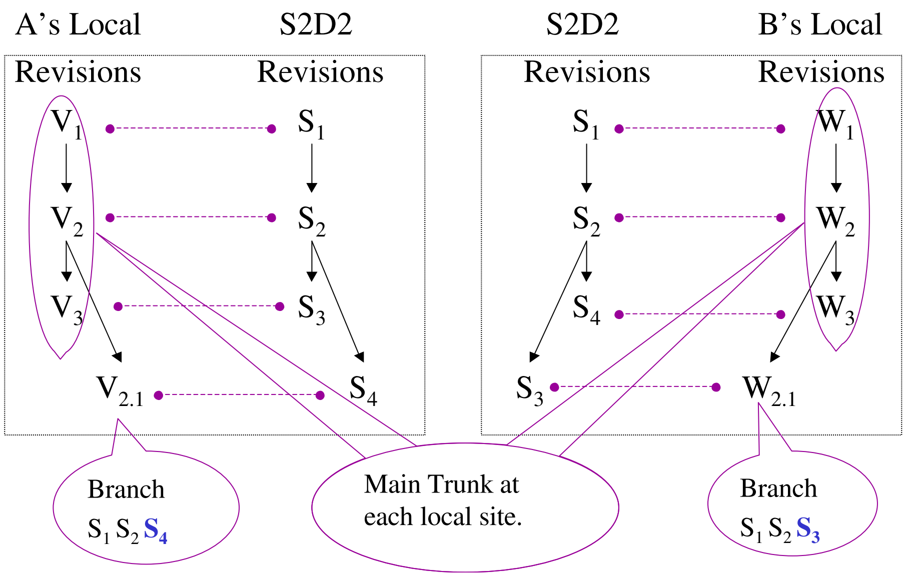
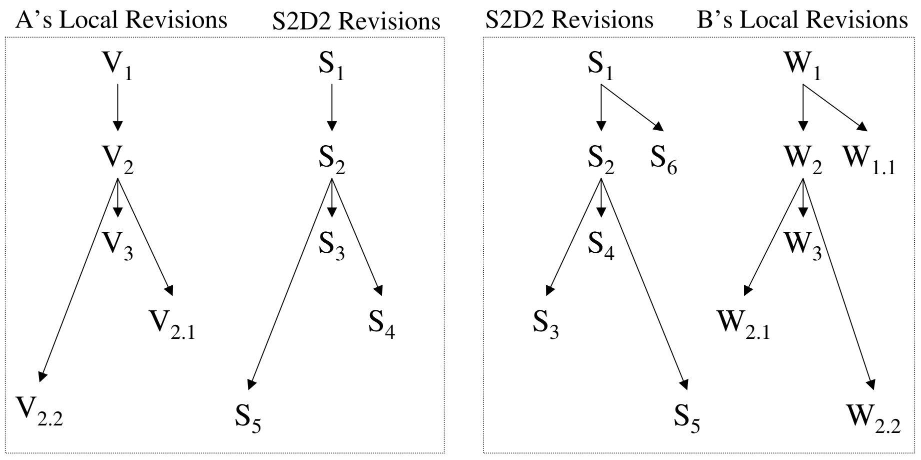


Figure 7.10: Main Trunk and Branch in S2D2 Adaptor: In S2D2 adaptor, the concurrent revisions can be stored as local branches; Site A stores S_4 as a branch while Site B stores S_3 as a branch. Site A thinks S_3 as the latest version in the main trunk since Site A has received S_3 before S_4. In contrast, Site B consider S_4 as the latest version in the main trunk since Site B has received S_4 before S_3.

provides a general framework. Thus, a directory object in one's file system can exchange data securely and scalably with a composite object in another's data base. For example, one user may keep a list of addresses in a file directory while another keeps the shared addresses in a PDA database; yet, they can exchange objects via this common abstraction provided by S2D2.

7.1.5 Upcall for Data Incorporation and Conflict Resolution

In mutable data sharing, users need to be notified about concurrent versions; then users have to merge concurrent versions with the help of application specific resolvers. In some systems such as Bayou and Coda's disconnected files system, such application-specific merges are automatically applied upon detection and users are only notified of the conflicts



$S_1 : (V_1 = W_1)$. $S_2 : (V_2 = W_2)$. Because $V_3, W_3, V_{2.2}, W_{2.2}$ are concurrent and have the same ancestor S_2. $S_3 : (V_3 = W_{2.1})$. $S_4 : (W_3 = V_{2.1})$. $S_5 : (V_{2.2} = W_{2.2})$.
Later, site B received a $S_6{:}(W_{1.1})$ that is concurrent with $S_2{:}(W_2)$.

Figure 7.11: Deterministic Merge of Concurrent Revisions: This example shows many concurrent revisions that need to be merged at each site. Site A created S_3 while site B created S_4 concurrently and a third site produced another concurrent revision, S_5. Site A stored S_4 and S_5 as branches, while Site B stored S_3 S_5 as branches. Later, Site B receives another concurrent revision, S_6, to an ancestor version in the main trunk, S_2. To produce the same merged result at each site, the merge procedure needs to be applied in deterministic order. In this example, the concurrent revisions need to be applied according to the lexicographically sorted order of summary hash value. S2D2 adaptors at both Site A and B need to merge most recent concurrent revisions first. Thus, Site B will merges S_6 into the trunk after it merges S_3,S_4 and S_5 in lexicographically sorted order.

that cannot be automatically resolved.

In S2D2, the application specific merge is provided by the adaptor, which determines when to merge, how to merge and whom to notify of any conflict that cannot be resolved automatically at the adaptor. An S2D2 adaptor has to register the callback incorporation() routine so that S2D2 can call the adaptor-provided application-specific incorporation routine for every incoming updates.

In some applications the concurrent update is not merged immediately and further distributed as an isolated revision thread, we call such revision thread as *branch* while we call the main revision threads *trunk*. Figure 7.10 shows an example of a trunk and a branch. As shown in this example, trunk and branch are site specific definition – a version can be considered as a branch at one site, while the same version can be regarded as a version in the main trunk. Later when users want to combine the branch into the main trunk, the applications can merge the branch into the main trunk. We call this merge operation *branch-join*.

Here, we provide a guideline how the incorporation routine should apply deterministic merge procedure to concurrent revisions. In merging concurrent revisions, S2D2 adaptor should apply a deterministic merge tool in a deterministic order so that the result can be the same regardless of which site performs the merge. Instead of merging the branch into a main trunk (e.g., branch-join), S2D2 adaptor should apply the merging in a deterministic order. Figure 7.10 shows an example, in which both sites merge S_3 into S_4. Without deterministic order, one site may merge S_3 (branch) into S_4 (main trunk) while the other site merges S_4 (branch) into S_3 (main trunk). Then, the merged result could be different.

This deterministic merge order becomes even more crucial when the site needs to merge three or more concurrent revisions as shown in Figure 7.11. Moreover, it is also possible that there are other concurrent revisions (branches) to another ancestor as well. In this case, S2D2 adaptor needs to merge the concurrent revisions (branches) of the latest ancestor first and to the oldest as shown in Figure 7.11.

Now, we describe when the adaptor should merge the concurrent revisions. The adaptor can ask the user to merge the concurrent updates at the time the user (i) publishes her new update to others, or (ii) incorporates others' updates into her own local copy. Once the divergence is detected (typically through frequent summary hash exchanges in the two-step SHH reconciliation), the adaptor can immediately notify the application user of the concurrent revision without waiting for publish or incorporate requests from the application user. Some applications may require such immediate notification while this immediate notification may be unnecessary to other applications. They may prefer to wait since other sites can merge the divergence and the merged revision can be delivered to S2D2 repository even before the user notices the divergence.

In some application, concurrent versions need to be merged immediately with application-specific merge process without user's involvement. Because the merge was applied immediately upon detection, the S2D2 adaptor may undo this previous merge when it tries to merge another concurrent revision into the previous merge.

7.2 Applications Built Based On S2D2

We present three example applications that are based on S2D2. Universal-Folder and P2P-CVS are examples demonstrating the S2D2 provided benefit – the highly available mutable data sharing. Update-Mine-In-Yours (e.g., Address Book) is an example showing that the owner of a replicated object can actually update the replica in other domains at different application in a secure and scalable way. For each application, we discuss the semantics of S2D2 adaptors. Each adaptor implements a different policy concerning when to publish, how to incorporate the incoming updates and how to record the concurrent revisions (e.g., as branch or as special file) and how to merge them.

7.2.1 Universal Folder

Files and directories are well-known ways of managing data. Hence, many collaborative tasks can be performed through simply sharing a directory or a set of files. Traditionally, a central server provides a directory (also called "file folder") that can be accessed by many collaborating writers and readers.

To avoid a single point of failure, and the dependency on a central server, we build a peer-to-peer version of file directory sharing using the S2D2. Each site publishes or subscribes to a set of S2D2 objects; concurrent updates are recorded as a special file; a user is asked to resolve the conflicting updates.

In Unix file semantics, a read should return the most recent write, and concurrent writes should not be visible to the user. In the event of concurrent updates, the system does not ask the user to merge; instead, the system always overwrites the write with earlier time-stamp with the write with the latest time-stamp. For example, in a Network File System, the writes are propagated "immediately" to the server upon user's save. However, concurrent writes do happen because of write-cache optimization such as NFS 30sec window – each write is locally cached for 30 seconds before it is saved onto the NFS server. In such event, the user may experience a lost update since the write with the latest time-stamp overwrites the other concurrent writes with older time-stamp.

In our Universal Folder, a weakly consistent folder replication, a read returns the most recent write that the site has received so far. Upon user's save, the fresh write is published out by propagating writes to neighbor's S2D2. Hence, the S2D2 adaptor needs to externally monitor the user's save operation to determine the publishing points on behalf of the user. Upon receiving a new write, the receiving site incorporates the write into the folder objects immediately. Unlike Unix file semantics, in Universal Folder, concurrent writes need to be notified to users, who have to review the automated merge and resolve conflicts with the help of application specific resolvers. This is similar to the semantics of Bayou and Coda's disconnected operation.

Universal Folder was built by implementing a S2D2 adaptor. The adaptor maintains the mapping between the local file and the S2D2 object. In the following, we present partial pseudo code for the important parts of the S2D2 adaptor for Universal Folder.

```
Class UnivFolderAdaptor {
  Prepare(LocalFileName) { // initialization
    create an owner object;
    create an objectID ownerpubkey:LocalFileName;
    create a mapping between LocalFileName and objectID;
  }
  Publish(LocalFileName) { //publish revision
    locate objectID for LocalFileName;
    call owner.publish(objectID, LocalFileName);
  }
  Incorporate(objectID) { //callback
    locate LocalFileName for objectID;
    load ORH = ObjectRevisionHistory for objectID;
    locate the ORH.branch;
    if (ORH.branches != null){//need to merge concurrent updates
      save all branches into LocalFileName#branch's summary hash#
      apply a merge tool in deterministic order;
      ask user to resolve conflicts if there is one;
    }
    //case: no concurrent updates or the merged update
    save ORH.latest-version into LocalFileName;
  }
}
```

7.2.2 Update-Mine-In-Yours

In mutable data sharing and replication, once one's object becomes replicated at another site, the update to the replica has to go through the application that manages the replica. Oftentimes, human users are involved in updating this replica for security or due to the lack of common update framework. S2D2 can provide a common substrate for scalable update delivery with access control based on a public key infrastructure, and it also provides secure versioning with undo capability.

For an address book application, the adaptor maintains the mapping between S2D2 object and the user.vcf file. The user.vcf file is exchanged as a S2D2 object. The adaptor also knows how to put the user.vcf file into the corresponding data item in the user's local addressbook database (e.g., PDA address book).

In the following, we present partial pseudo code for the important parts of the S2D2 adaptor for the Address Book Adaptor.

```
Class AddressBookAdaptor {
  Prepare(user.vcf) { // initialization
    create an owner object;
    create an objectID ownerpubkey:User.vcf;
    create a mapping between User.vcf and objectID;
  }
  Publish(User.vcf) { //publish revision
    locate objectID for User.vcf;
    call owner.publish(objectID, User.vcf);
  }
  Incorporate(objectID) { //callback
    locate User.vcf for objectID;
    load ORH = ObjectRevisionHistory for objectID;
    locate ORH.branch;
    if (ORH.branches != null){//need to merge concurrent updates
      save all branches into User.vcf#branch's summary hash#
      apply a merge tool in deterministic order;
```

```
        ask user to resolve conflicts if there is one;
      }
      //no concurrent updates or updates have been merged above
        save ORH.latest-version into User.vcf;
        import User.vcf into MyAddressBook;
          }
  }
```

7.2.3 P2P CVS

In traditional CVS, writes are published and incorporated upon users' explicit commands. By maintaining a complete revision history, concurrent writes are always detected at a central repository server and need to be resolved by the user at the time of committing updates to the server. Instead of the central repository server, in P2P-CVS, each user has her own local repository that is being shared under S2D2 control.

We describe two ways of implementing P2P-CVS. In one way, the adaptor supports the traditional CVS interface. Users use the traditional CVS interface to manage (e.g., check-in, check-out) their own local repositories using CVS revision number(e.g., r1.1, r1.2). Figure 7.12 shows an example. However, since the revision number is assigned locally, the number cannot be used as an global identifier. Hence, the local CVS repository cannot be directly replicated as it is. The revision number in the local CVS repository needs to be replaced with the globally unique summary hash to produce a globally sharable CVS repository. The adaptor also needs to maintain the mapping between the revision number

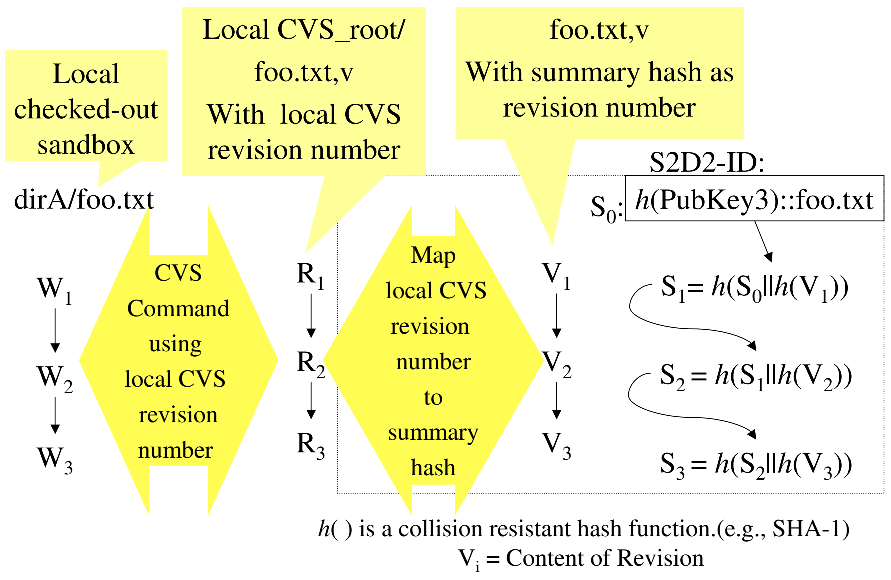

Local checked-out sandbox
Local CVS_root/ foo.txt,v With local CVS revision number
foo.txt,v With summary hash as revision number
dirA/foo.txt
S2D2-ID:
S_0: h(PubKey3)::foo.txt
W_1
W_2
W_3
CVS Command using local CVS revision number
R_1
R_2
R_3
Map local CVS revision number to summary hash
V_1
V_2
V_3
$S_1 = h(S_0 || h(V_1))$
$S_2 = h(S_1 || h(V_2))$
$S_3 = h(S_2 || h(V_3))$
h( ) is a collision resistant hash function.(e.g., SHA-1)
V_i = Content of Revision

Figure 7.12: S2D2 Adaptor for Traditional CVS Example: Each site maintains its own local CVS repository. To produce a sharable repository, the adaptor replaces the local specific revision number in the local repository with the corresponding summary hash that is globally unique. The sharble repository contains summary hashes and are replicated at other collaborating CVS users' sites using S2D2. The adaptor at each site needs to maintain the mapping between the local revision number in local CVS repository and the summary hash in S2D2. The mapping is implemented by tagging the CVS local revision number in the local CVS repository with the corresponding summary hash.

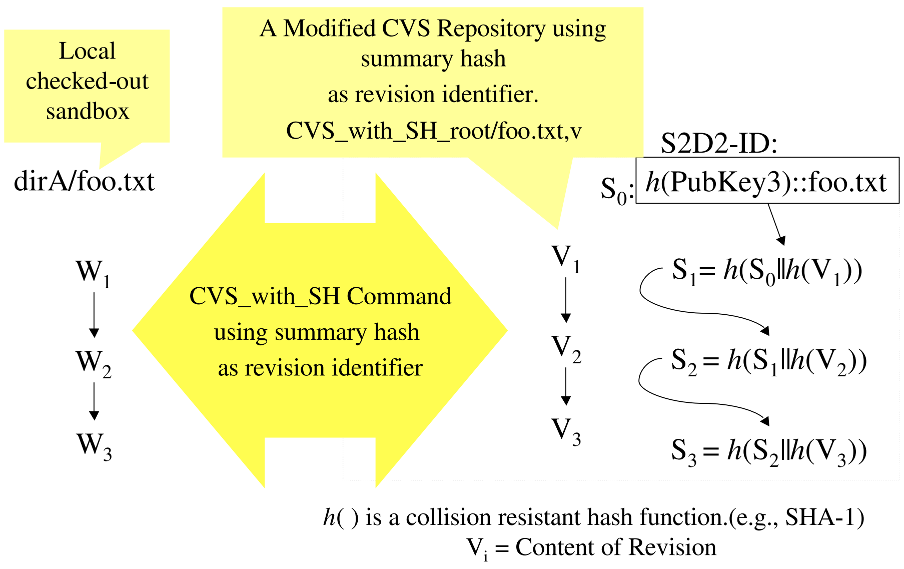

Local checked-out sandbox
A Modified CVS Repository using summary hash as revision identifier. CVS_with_SH_root/foo.txt,v
dirA/foo.txt
S2D2-ID:
S_0: h(PubKey3)::foo.txt
W_1
W_2
W_3
CVS_with_SH Command using summary hash as revision identifier
V_1
V_2
V_3
$S_1 = h(S_0 \| h(V_1))$
$S_2 = h(S_1 \| h(V_2))$
$S_3 = h(S_2 \| h(V_3))$
$h(\,)$ is a collision resistant hash function.(e.g., SHA-1)
V_i = Content of Revision

Figure 7.13: S2D2 Adaptor for a Modified CVS Example: Each site maintains its own local CVS repository. Since this modified CVS repository uses globally unique summary hashes as revision identifiers, sites can share with this CVS repository as it is.

and corresponding the summary hash. If the update to a globally sharable CVS repository is received, the adaptor needs to modify local CVS repository accordingly by assigning local revision number.

This method may work if there is few concurrent updates; however, it may introduce confusion among users in identifying the concurrent revisions. Some user may regard a version as a main trunk revision; while, some users may regard the same version as a branch.

To avoid this confusion, the traditional CVS interface needs to be modified. Instead of local revision number, an modified CVS can use summary hash as revision identifer. An example for this approach is shown in Figure 7.13. Users are expected to use summary hash as revision identifier in working with CVS commands. For example, in modified CVS, “cvs update -rLocal_1.2” should be the same as “cvs update -rSUMMARYHASH_of_Local_1.2”.

In the following, we present partial pseudo code for the important parts of the S2D2 adaptor for Modified CVS.

```
Class P2P-Modified-CVSAdaptor {
  Prepare(RepositoryFileName) { // initialization
    create an owner object;
    create an objectID ownerpubkey:RepositorylFileName;
    create a mapping between RepositoryFileName and objectID;
  }
  Check-in(RepositoryFileName, revision) { //publish revision
    locate objectID for RepositoryFileName;
    if (ORH.branches != null) ask if user can merge now;
    do cvs commit revision into RepositoryFileName;
    call owner.publish(objectID, RepositoryFileName);
  }
  Check-out(RepositorylFileName) { //contacting the local CVS repository to retrieve re-
vision
    locate objectID for RepositorylFileName;
    if (ORH.branches != null) ask if user can merge now;
    if (ORH.branches != null) ask user which branch to check out;
    do cvs checkout RepositoryFileName;
  }
```

Update(RepositorylFileName) { //contacting the local CVS repository to update local file with latest version in the CVS repository. If local file has been modified, CVS merge applies.

```
    locate objectID for RepositorylFileName;
    if (ORH.branches != null) ask if user can merge now;
    if (ORH.branches != null) ask user which branch to check out;
    do cvs update RepositoryFileName;
  }
  BranchMerge(){ //called to merge concurrent updates
    pick branch in deterministic order;
    save branch into a CVS branch named with summary hash;
    apply cvs -join in deterministic order;
    ask user to resolve conflicts if there is one;
    }
  Incorporate(objectID) { //callback
    locate RepositoryFileName for objectID;
    load ORH = ObjectRevisionHistory for objectID;
    locate ORH.branch;
    if (ORH.branches != null){// we have concurrent versions
      ask if user can merge now
      If yes, call BranchMerge(); else, do nothing;
    }
    else {//no concurrent updates
      save ORH.latest-version into RepositoryFileName;
      by using cvs commit ORH.latest-version;
    }
  }
}
```

As we shown in pseudo code above, the check-out gives an option for a user to choose which branch to check out if there are concurrent revisions. With CVS, users can tag the branch with a mnemonic symbol such as "final-draft". Similarly, in a P2P-Modified-CVS adaptor, users can tag each published object with a composite hash that was created to express the dependency among objects (i.e., files). The "composite hash" can be created by hashing the concatenated summary hashes of all objects that are related to each other. This composite hash can guide P2P-CVS to select which revision of each object needs to be checked out together.

7.3 Evaluation

We evaluate the benefits of our S2D2 mechanism in terms of data availability.

7.3.1 Event-Driven Simulator

The event-driven simulator uses S2D2's two-step SHH reconciliation on a simulated network topology. As the underlying topology we use 1954 node transit-stub physical network topologies generated using the GT-ITM library [55]. The links in the network are categorized into 1.5Mbps transit-stub links, 45Mbps transit-transit links, and 100Mbps stub-stub links. The bottleneck links are the transit-stub links in the topology. Given the physical topology, we choose nodes from the underlying topology in which to place sites randomly. In figures excluding a trace, each data point is the average of 10 simulations.

7.3.2 Data Availability

We simulated the updates of multiple writes to examine the data availability for a centralized server-based system and our S2D2-based system. The data availability (*DA(t)*) is defined as the percentage of on sites that know the latest update at time t. The average data availability ($<DA>$) is the average of DA(t) over the entire simulation period. More specifically:

$$< DA >= \frac{1}{T}\int_0^T DA(t)dt \tag{7.1}$$

We first explain experimental settings. The centralized server-based system has a central server and all clients contact the central server every one minute to retrieve new updates. The central server and clients are periodically on and off given the percentage of on time. If the server is off, the server cannot respond to the requests of the clients. For the S2D2 case, we use the data exchange mechanism of S2D2. Sites are periodically on and off given the percentage of on time. A site can reconcile with any site that is currently on. In both systems, the interval between writes is exponentially distributed with mean of 20 minutes. The duration of on and off time is two hours. One simulation runs for 168 hours

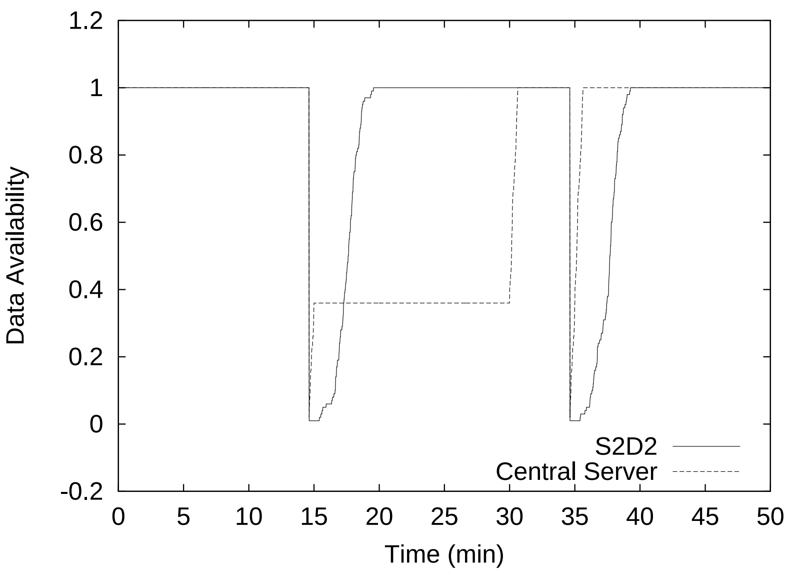


Figure 7.14: Data availability trace for 100 sites for S2D2 and 100 clients for the central server. The updates occur at 14.6 minute and 34.6 minute. The central server cannot be reached between 15 minute and 30 minute. At 15 minute, 36% of sites received the update from the central server. Until the central server can be reached, the other sites cannot exchange updates. On the contrary, with S2D2, sites can continuously reconciles with each other.

(7 days) simulated time.

Figure 7.14 shows a part of the trace of data availability for 100 sites of S2D2 and 100 clients for the central server. In this trace, the updates occur at 14.6 minute and 34.6 minute. After the update, the data availability increases from 0 to 1. The slope of the increase is different for the central server and S2D2. In the central server, every client can contact the server within one minute after the update if the server is on. However, S2D2 data exchange needs a few rounds of reconciliation to reach the data availability of one. In the trace, the central server is not reachable between 15 minute and 30 minute. Until the central server can be reached, the other sites cannot progress. On the contrary, S2D2 continuously reconciles the update among sites, which is not possible for the centralized

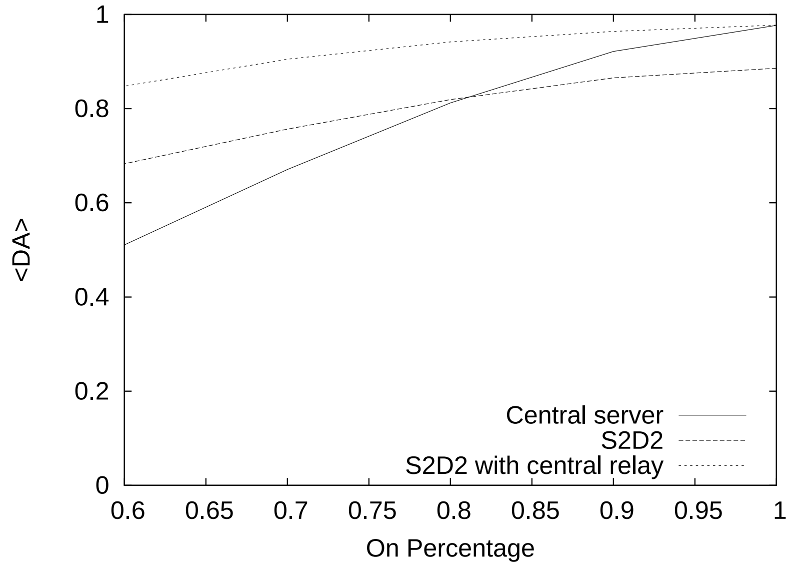


Figure 7.15: Average data availability for 100 sites of S2D2 and 100 clients of the central server based system. The percentage of on time of the central server and all sites of S2D2 is varied from 0.6 to 1.0. The graph shows S2D2 with central relay provides best data availability by taking advantage of central relay when central relay is available. When central relay is not accessible, S2D2 exchanges updates through pair-wise reconciliations, while the clients that depend only on the central server cannot exchange updates.

server-based system.

Figure 7.15 shows the average data availability varying the percentage of on time. S2D2 with central relay is the case where there is a central point to check updates first in S2D2. In our experiments, the central server and clients are on and off, all sites in S2D2 are on and off, and the central relay is on and off periodically. The central server has a higher average data availability than S2D2 when the server is highly available, since S2D2 anti-entropy exchange of hash takes longer to converge. However, as the percentage of on time decreases, S2D2 shows higher average data availability. By adding a central relay in S2D2, we can get the benefits of the central server when the relay is highly available and the benefits of S2D2 when the percentage of off time increases (i.e., the time that the relay is not available increases).

S2D2 can automatically select highly available sites as the reconciliation points more often. These sites play the role of relaying the updates to other sites, thus mimicking the central server. Maintaining such relay sites will help other sites converge faster, thus getting higher data availability.

7.4 Prototype Experience

7.4.1 Implementation

We implemented S2D2 prototype in Java language (6638 lines). The major classes are following. Owner class is instantiated with the owner's key pair and provides interfaces to adaptors. ObjectRevHistory class manages SHH, Branches, and ToPullList. Rumor class provides two-step reconciliations. In addition, it provides push data code so that sites behind firewall can push the published data out to a site from which other sites can pull the data. As we learned from Bayou, S2D2 does not need to worry about the deadlock since we use asynchronous pulling of data. SecurityUtil class provides DSA signing/verification and hash typing for composite object. KeyManagement class manages certificates with X.509DN using KeyStore class from SUN.

Once we implemented the Universal Folder adaptor, it was quite easy to implement other applications adaptors by simply changing policy code. We found the adaptors can use the frequent exchanges of hashes as quick invalidation, guiding a good work-flow that does not introduce unnecessary divergence.

7.4.2 Microbenchmark

We performed the microbenchmark of the execution time of our S2D2 prototype. Receivers pull data from our machine A. Our focus of these measurements was execution time at these end hosts.

The sender side machine A is the 1GHz SPARCstation running Solaris OS connected with 100Mbps located in west coast of US (at MIT). We experimented with three different receiver configurations – Receiver R1: 1.2GHz Pentium IV, Linux OS, with 100Mbps LAN located in east coast of US (at Berkeley), Receiver R2: 667MHz powerbook G4 with DSL connection, and Receiver R3: Pentium III Dell 4100, Win2000 OS, 100Mbps LAN with which machine A is connected.

Table 7.1 shows the execution time microbenchmark results varying data size for the receivers. The major tasks of the sender side are base64 encoding into a file.base64 (Encode), computing the SHA-1 hash of data (Hash), signing the hash (Sign), and sending the data to the network while reading a file (Send). The major tasks of the receiver side are receiving the data from the network and writing to a file (Receive), base64 decoding (Decode), and the signature verification of the 20B hash (Verify).

As shown in the Table, the most significant overhead is the base64 encoding/decoding of data (we used sun.misc.Base64Encoder/Decoder), because we encoded/decoded into a file. When the data size is 1MB, the encoding time is almost 38s. When we encode into 10MB memory buffer instead of file, the encoding takes 1782ms for 1MB, 677ms for 100KB, and 329ms for 10KB. Since the hash is generated over the base64 encoded data, we can further optimize by streaming the encoded data into SHA generation and over to the socket

Receiver	Data size	Encode	Hash	Sign	Send	Receive	Decode	Verify
R1	10K	461	8	227	456	326	3	95
	100K	4252	73	210	876	844	32	88
	1000K	42214	515	241	4633	4514	536	90
R2	10K	833	9	238	369	364	26	119
	100K	4802	54	278	1085	1205	400	121
	1000K	38800	527	256	11175	11230	3343	97
R3	10K	451	9	226	344	140	12	40
	100K	4019	52	254	400	260	45	70
	1000K	39328	516	230	892	761	330	50

Table 7.1: Results of execution time microbenchmark. The unit of data size is byte, and the unit of other fields in the table is ms.

as well. Note that the sender is the same machine in all three cases. Encoding time takes orders of 100 longer than decoding, although decoding time differs by the resources on the receiver machine. Signing and signature verification are fast regardless of data size, since only 20B hash is signed and the signature size is 46B. These measurements show that our prototype implementation shows reasonable performance for different computation and network environments.

7.5 Related Work

7.5.1 Weakly-Consistent Replication Systems

S2D2 is similar to Bayou [49, 50, 15] in that it is a substrate mechanism that provides flexible update exchange to shared objects such as files, data items, and sets of files, not as a distributed file system. However, S2D2 is based on SHH; while, Bayou is based on dynamic version vectors. Being based on SHH, S2D2 can overcome version vector inherent limitations in terms of security and scalability.

S2D2's divergence control is similar to the use of application specific resolvers in

Bayou and Coda. However, S2D2 uses SHH to detect and guide deterministic merges based on data contents; while, Bayou uses version vectors to extract deltas (incremental updates) that need to be exchanged during the reconciliation process.

Recently, a number of peer-to-peer wide-area file systems such as Pangaea [41], FarSite [2], and Ivy [31] have been built. In Pangaea, the concurrent updates are merged according to version vectors and conflicts are resolved with last-writer-wins rule based on a loosely synchronized clock. FarSite provides a single logical file system view, harnessing the distributed untrusted computers in a scalable and secure way. However, unlike S2D2, FarSite is not designed for scalable write sharing; hence, FarSite uses ping-ponging leases or a central redirecting server [33] for serializing concurrent updates.

In Ivy, each site maintains the logs of each record, which has a pointer to the previous log record. Although version vectors are used to put a total order when reading from multiple logs, Ivy can detect some of the same version id attacks as concurrent, since they treat identical version vector entries as concurrent. However, Ivy can be still vulnerable to a lost update problem. For example, R_1 from site A and R_2 from site B have the same version vectors. Site C creates a new revision R_3 based on only R_1. When site C reads R_1 to create R_3, it might have read R_2 as well if site B was reachable at that time. However, if not, R_3 from site C is solely based on R_1 not both R_1 and R_2. But, R_3 falsely overwrites R_2 of site B as well. This can lead into a lost update of R_2 in the system.

7.5.2 Use of Application Specific Knowledge for Conflict Resolution

Bayou[35] and Coda [25, 27] provides tools so that the reconciliation of conflicts can be resolved based on application specific knowledge without user's repeated and unnecessary involvement in merging conflicts. With these systems, the application specific knowledge was proven to be useful to determine the policy in merging conflicts. Interestingly, an application-neutral approach [1] has been proposed. Also, the application specific knowledge is useful in optimizing the size of incremental updates that need to be exchanged during reconciliation. For example, many optimistic replication systems presented a method of compacting self-canceling deltas using application specific knowledge [11, 23, 32, 54].

7.6 Summary

S2D2, a framework based on SHH, provides various service components that are required for global-scale optimistic replication. It provides the SHH based reconciliations, a global naming for shared object and an access control mechanism based on a PKI (Public Key Infrastructure). S2D2 uses hash typing, which is used to enforce the correctness in the error-prone usage of various hashes and to communicate composite object's structural information for scalable data exchanges.

Applications built on top of S2D2 prototype show that the adaptor architecture flexibly supports various application specific consistency and usage semantics. Our simulations show that, unlike central server-based systems, S2D2 can continuously exchange up-to-date data efficiently under network partition or server failure.

S2D2 employs a useful principle, hash typing to avoid error-prone usage of various hashes. Hash typing is also used to communicate the application specific composite object structure information to the S2D2 layer so that both S2D2 and application adaptor can utilize the structure information for scalable, incremental, and optimal data exchanges.

We built the S2D2 prototype and three diverse applications based on S2D2: Universal Folder, P2P CVS, and Update-Mine-In-Yours. Our prototype implementations show that the adaptor architecture is quite useful in separating the various applications' different usage semantics from S2D2's scalable and secure mutable data sharing features. For example, the S2D2-adaptor for universal folder were built with about 100 lines of Java code implementing the callback routine for upcall and mapping information between local file path and the S2D2 object ID.

We performed the latency micro-benchmark of our prototype. The benchmark shows that the hash generation, signing, and verification takes tens of ms. The major execution time is the base64 encoding of data. We also evaluated our S2D2's high data availability and scalable data exchange through simulations. In the event of network partition or server failure, S2D2 can continuously exchange up-to-date data efficiently, unlike central server-based systems. In addition, the lazy-selective data pulling of S2D2 consumes orders of magnitude lower bandwidth than the traditional version-vector based anti-entropy data exchanges.

We evaluated the S2D2 in terms of the data availability through simulations and performed latency micro-benchmark of our prototype. The benchmark shows that the hash generation, signing, and verification takes tens of ms. The major execution time is the base64 encoding of data. In the event of network partition or server failure, S2D2 can continuously exchange up-to-date data efficiently unlike the central server-based system.

Chapter 8

Toward A Model of Self-administering Data

In this chapter, we describe a model of self-administering data as a novel collaborative application model. After we discuss the problem with current collaboration tools available for simple document sharing, we describe our proposed data management model, where a declarative description of how a data object should behave is attached to the object, either by a user or by a data input device. This self-administering data model requires a widespread infrastructure of self-administering data handlers, which can be realized using the S2D2 framework.

8.1 Limitations of Current Tools for Data Management

In this section, we consider a number of scenarios that motivate our design. Mostly, we express our frustration with current tools available for simple processes, and suggest what, to us, seem like more attractive scenarios. Below we present the data processing model we designed to enable these scenarios.

8.1.1 Co-Authoring across administering domains

Example Problem: Suppose a web-page designer is commissioned to create some web pages from a customer. The customer somehow communicates what is desired to the designer, who then creates an initial version of these pages. Perhaps the designer sends these page drafts to the customer by email as attachments, or has the customer download the web-pages from the designer's web site, or uses some other protocol, like ftp, to move copies about. Later, the customer makes some modifications and returns the pages to the designer, and the process iterates.

As a result, both users' email boxes, or file spaces, etc, get filled with email attachments of versions. These versions are often hard to manage because there is no built-in version management support for email attachments, HTTP, or FTP. Our collaborators could instead try to use some collaboration tool designed for this purpose. Heavy weight document management system like Lotus Notes or even Xerox's Docushare are probably overkill for this purpose; moreover, they may require administrative commitments neither user can make.

Web-based file sharing system such as www.desktop.com, www.hotoffice.com, WebEdit, I-drive and BSCW, and synchronization services, such as FusionOne, provide an interesting alternative. However, such services don't provide control over important aspects of data management, such as back-up, conversion, and merging. Moreover, the users are at the mercy of a potentially overloaded server, perhaps at a precariously financed dot com.

Also, adding a third party to the interaction introduces increased vulnerability: Users are not able to perform their sharing operation when the central server is down even though their local machines and services are functioning, and have introduced a new security concern. In addition, they are subject to various, and, we think, avoidable, human errors, such as forgetting to transmit the shared copy to the web repository after every change.

8.1.2 Desired Properties and Proposed Data Model

The above scenario suggests to us the following properties of an ideal system for this task:

P1: No repeat user involvement in routine data management

P2: No unnecessary dependence on shared resources, such as shared data repositories or file servers

P3: No prior administrative set up costs

P4: Ability to exploit minimal use of central server as only required

P5: Undo/Redo capability within user's domain

P6: Secure and safe incorporation of updates at user's domain

P7: Lightweight enough to be widely deployed

A Proposed Solution: We propose a way of accessing and managing data to achieve the above desiderata. We introduce an infrastructure of Self-administering Data Handlers, which are deployed wherever users wish to take advantage of their services. These Self-administering Data Handlers (SD Handlers) administer data according to an attached Self-administering Data Description. The Self-administering Data Description (SDD) is meta-data describing how, where, and to whom the data are to be copied, updated and otherwise administered. In other words, the SD Handlers are daemon processes that administer data by honoring attached self-administering descriptions.

Consider how the task above might be performed if SD Handlers were available to the collaborating parties. When the web-page designer creates web-pages, she saves them into a directory or folder somewhere on her local disk, as is her standard practice. Her SD Handler detects this action and pops up a UI with a self-administering description for the saved web pages, probably representing her defaults. She examines the default preferences, checks a couple of choices and adds a new destination, in this case, a location specifier provided by the client. Then the SD Handler attaches to the data objects their respective self-administering description.

Suppose the designer specified that these pages should be delivered to client's public web folder whenever she updates one of those. When a page is updated, the SD Handler will automatically sign it with the designer's private key and encrypt the result with the client's public key. The signed and labeled data object is deposited into the network of SD Handler infrastructure.

The client's SD Handler receives and verifies the authenticity of the self-administering description. In this case, it interprets the description as instructing incorporation of the data object into client's web folder. The client's SD Handler logs this event of data incorporation. If the designer's name is not found in the client's trustee list, the incorporation is denied. If the recipient's SD Handler is not available, the SD Handler could retry or delegate the retrying of delivery to a pre-negotiated server.

If the designer prefers strong update serialization, the SD Handler might be configured to first contact a pre-negotiated central serialization service, (say, a CVS [7] server or the Oceanstore [26] service) and have her changes merged according to the arrival order at the central serialization server. The merged data are then delivered back to the designer's SD Handler, which forwards the merged data to the destinations specified in the self-administering description.

Such a network of SD Handlers provides a lightweight asynchronous collaboration infrastructure for sharing data in a secure way. Centralized servers may be exploited in this process, but only when there is some particular need that justifies the cost, such as a desire for strong serialization of updates.

8.1.3 A Less Desktop-Centric Scenario

Let us briefly consider a less desktop-centric scenario. Suppose a botanist takes pictures of plants in a field with her digital camera. She wants to transfer these to multiple remote designations, including her own web page, her research group's database, and her collaborator's disk. To do so, she must go to her office desktop and download the image from the digital camera into some buffer space, and then copy it into her own web page folder. She must then open up a database connection, authenticates herself, and then upload the data into database. She would also pop up an email client, create a new email message and upload the image data as the message's attachment. Then she sends the email to her collaborators, asking them to download the attached image onto their disk space. She repeats these procedures whenever she takes a picture or pictures she wishes to so incorporate.

This scenario provides comparable desiderata to the initial one, except that one would like to deploy our proposal as close to the data as possible. Thus, we must modularize SD Handler functionality so we can implement its services within a device's limited resources. For example, the camera might be enabled with a simple interface for using some pre-downloaded self-administering descriptions. Services that the camera couldn't perform locally could be performed by an affiliated proxy server. The camera need only reach the proxy server for the rest of the tasks to be automated as above.

We envision data collection involving SD Handlers from a wide variety of simple special purpose devices, include scanners, smart cards, smart mobile phones, PDAs, and lightweight widely distributed sensors. These devices, perhaps together with a helpful proxy, simply deposit their tagged data into the infrastructure, which takes care of all routine data management and transport issues.

8.2 Self-administering Data Model

As suggested above, we envision a network of SD Handlers, each "close" to a user or device that it serves. To a first approximation, there would be one SD Handler per networked device, perhaps more. Some would be associated primarily with users, some with data collection in devices, others with services, such as digital object repositories, each supporting basic SD Handler functionality, but perhaps implementing services associated with the particular characteristics of its application.

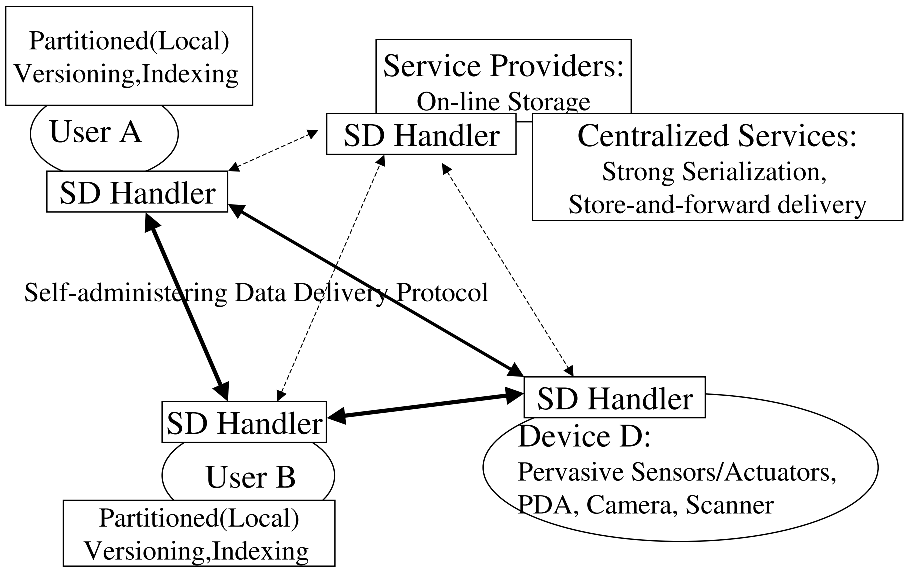


Figure 8.1: SDH Network with a Shared Service: Each SDH may contact a pre-determined SDH that provides a shared service. The shared service is not required for the correctness of SDH interaction rather for efficiency. Since the shared service is a soft state, any SDH can provide the shared service functionality.

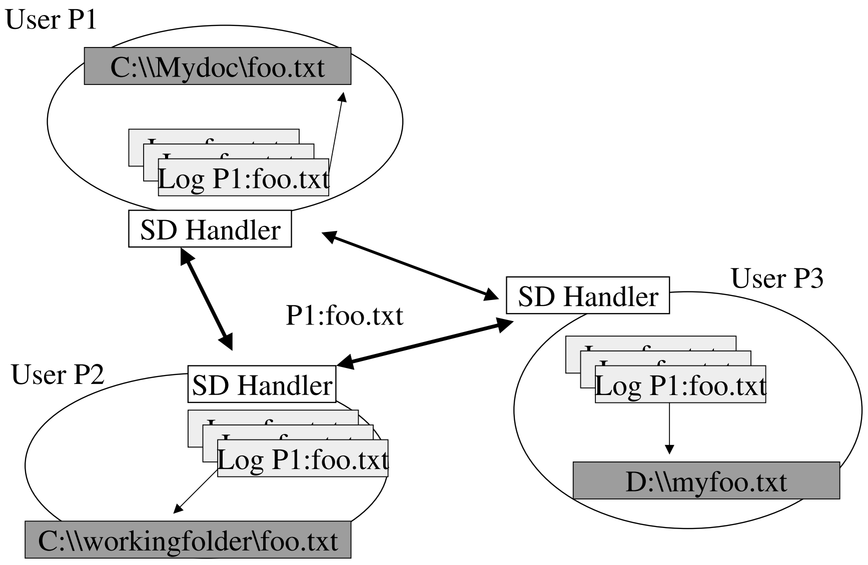


Figure 8.2: SDH Network of Revisions: Each SDH maintains revision history so that incremental delivery is performed based on revision differences. The revision history can be represented using *Summary Hash History.*

8.2.1 Network of SD Handlers

SD Handlers form a network, within which data is moved in accordance with the SD Handler's discipline. In addition, each SD Handler may provide an interface to a local collection or stream of data. The data may be a user's file system, web space, database, or other collection, administered by some mechanism other than the SD Handler. Such a network is illustrated in Figure 8.1 and Figure 8.2.

While these may be administered by a wide variety of mechanisms, the data looks the same once it is with the SD Handler network. We refer to each diverse collection of data as a data realm. In effect, the SD Handler bridges a realm into the SD Handler infrastructure.

8.2.2 Basic Functionality

Figure 8.3 presents an overview of the functionalities of each SD Handler. SD Handlers are required to implement the bottom tier of the functionalities, we define its basic functions. These are named bottling, floating, popping, and logging, and are described further below. To exploit capabilities fully, however, it is recommended that SD Handlers also implement an additional tier of functions on top of the basic services. These are called notifying, doffing, and versioning. Applications of various sorts may be built on top of these functions. In addition, GUIs and API need to be provided, to communicate with the user, and to form a bridge between the SD Handler and the user's data realms.

Here we present the basic building blocks of SD handling. These are bottling, floating, popping, and logging. We then describe how other functions can be built on top

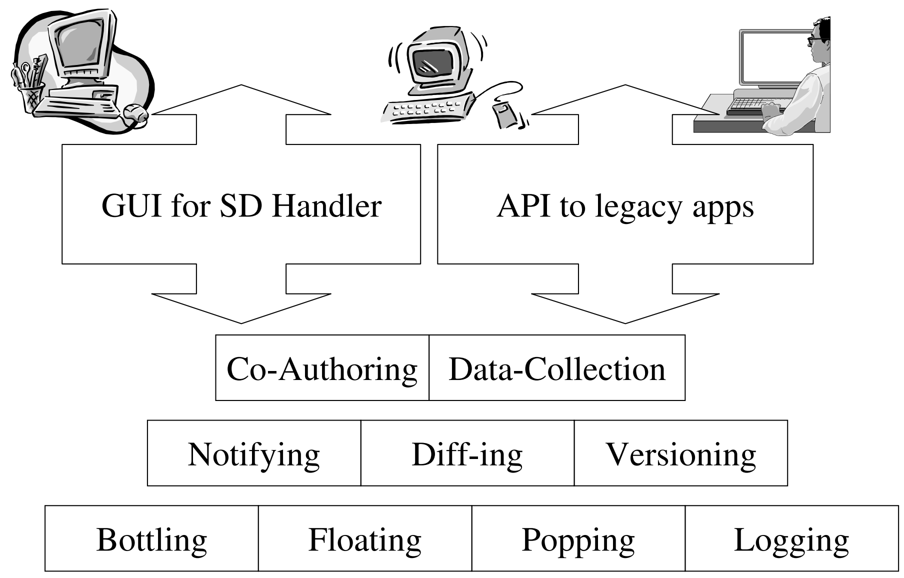


Figure 8.3: SDH functionalities: SDH provides basic functionalities: bottling (data encapsulation), popping (data incorporation), floating (data delivery) and logging (revision history). Additional tier of functions are optional.

of these basic functions.

Prior to this process, a self-administering data description is attached to the data object, akin to creating a packing slip for a shipment. This description contains the shipper's preferences for handling the data, as well as the lists of recipients and/or destinations. The data preference can include high-availability, strong-serialization and default archival support. The destinations can be an on-line storage of collaborator's, a database, a PDA, a smart phone, a speaker/media device and pervasive sensors/actuators. In accordance with this packing slip, at some point, the data object is bottled, i.e., prepared for shipping. To make a data bottle, the labeled data object is signed and encrypted. Then the bottle is floated across a sea of data. Finally, the bottle is popped at its destination(s), and the data extracted. All events are logged, so that support for other services, e.g., version control, and be readily accommodated.

Preparation

Prior to a SD Handler performing any operation on a data object, the object must be bridged into the infrastructure. I.e., a SD Handler has to be made aware of the object, and of the user's specifications for it. This is done by attaching a Self-administering Data Description (SDD) to the data. Since a self-administering data description can have many options and get quite complicated, we assume that most users never work with one directly. Instead, users interact with a UI. We have implemented a standard UI for a SD Handler running on desktop, which we discuss below. We assume that a different UI would be suitable for different devices, and that there would be default description templates for each user and each device, perhaps inherited or cascaded together as a function of the user and device environment.

Once an object has a SDD attached to it, the SD Handler aware of it will begin monitoring the object and attempting to enforce the specification of the SDD. Doing so typically results in sending a copy of the object to one or more recipients.

Bottling

When a SD Handler decides it must send a data object to a recipient, it first prepares a data bottle. It does so by signing the self-administering description and its data with its user's private key, for authenticating the sender at the recipient's SD Handler. The result is then encrypted with the destination's public key so that only the real destination can access the description and the data. The sender's credentials are checked against receiver's trustee list to allow appropriate access in incorporating the data at the destinations.

Then the bottle is floated, i.e., dropped into the SD Handler network. We describe floating below, but first examine the inverse operation of bottling, popping, which occurs when a SD Handler receives a bottle destined for a known user.

Popping

A delivered bottle is inspected for its integrity and the sender is authenticated for appropriate access right. Then the bottle is uncapped with matching encryption keys to be incorporated into the destination realm according to the packing slip. For safe incorporation, each incorporation process is logged for undoing or redoing operations.

The trustee-list maintained by SD Handler is used for giving or denying the delivery action from the sender. When SD Handler daemon process receives the bottled data, it authenticates the sender with trustee-list and decrypts the self-administering description to guide the incorporation activities. Incorporation is based on appending; SD handling never overwrites data, but may shadow it. Since every incorporation process is logged, it is always possible to undo the changes back to a specific version.

The bottled data is incorporated through SD Handler into any number of places, and in any number of different manners: onto a user's desktop, PDA, collaborator's domain, online-storage (NFS, Web), database entry, and even subdocument elements, such as anchor points in HTML page. The SD Handler running on a desktop computer maintains the history for the versioned content, and the incorporation activities. If the destination is database, the incorporation could comprise adding new entry; if the destination is a collaborator's online storage, the incorporation may create a newly updated file in a sandboxed location.

The followings are the examples of incorporations at various destinations.

A bottled data delivered

- onto UNIX file system, creates an i-noded file.
- onto a database, creates an updated (appended) database entry.
- onto a repository, creates a new index entry and is moved into repository space.
- onto a speaker device, creates voice data at the speaker
- onto another trusted user's file system, creates an i-noded file in a sandboxed location.
- onto a calendar/address book in a personal information managing application, creates anchor contains new data or new hyperlink pointing to a file in a sandboxed location.
- onto an anchor in a HTML document owned by another trusted user's, creates an anchor contains new data or new hyperlink pointing to a newly updated data in a sandboxed location.
- onto a writable CD, creates a newly added data on the writable CD

Floating

A bottled data object is dropped into the SD Handler network infrastructure. The infrastructure provides the delivery of the bottled data to the destination. SD Handler has its own delivery protocol (SDDP: Self-administering data Delivery Protocol) but SD Handlers can also use legacy protocols such as HTTP, FTP, and SMTP by tunneling SDDP.

The "floating" functionality of SD Handler provides store-and-forward service for delivering the data to a recipient who is not available at the time of delivery. It also provides the update serialization service, where the updates from the multiple participants are serially ordered according to the arrival time at the server. The bottled data has to flow into the serialization server and flow out to its destinations. Finally, the infrastructure of network of SD Handler provides a naming service to map the user's SD Handler's location into its current IP address. Each SD Handler that does not have static IP address, register its current IP address to the name resolution server in the infrastructure. And the SD Sender would cache the latest mapping and use it until the host becomes unreachable, and then it contacts the name resolution server for the current IP address of the participant's SD Handler.

Logging

The data and packing slip and its bottling/popping activities are logged for undoing/redoing and auditing purposes. The logging history can be flushed to a designated archival repository from the local space of the bottling and popping point. The log can be incremental in that the delta of changes is recorded. Doing so, of course, increases the dependency between logged objects, although it saves the disk space.

Versioning

Each SD Handler maintains its own version tree at its own realm by enhancing the logging feature. Decentralization is achieved by naming the same resource uniquely along with its version number across different administrative domain. This is done by prefixing the owner-name to a local name of resource, as each SD Handler has its own name space per given owner-name. We use the owner name to uniquely locate its public key. The typical owner name could be email address where the uniqueness is being maintained at its organization or email service provider. In CVS [7] and WebDAV [53] there is one version tree that is maintained at the central server with its single name space. In contrast, in SD Handler, different version trees are maintained at each realm. They share only a naming convention that uniquely addresses identifiable resources across different version trees. The shared naming convention that each SDH has to follow can be provided by S2D2's secure naming.

8.2.3 Self-administering Data Description (SDD)

The basic responsibility of a SD Handler is to interpret SDDs, i.e., a Self-administering Data Description attached to a data object. SDDs are written in SDDL that is based on XML. Figure 8.4 provides a typical example. We stipulate that every SDD has one owner, but may contain multiple users as sharers. Each user in the sharer element can have its own multiple self-administering data server (SD server) locations. The central server element given by the owner provides the central server location for SD Handler's "floating" operations (to be described below), such as store-and-forward data delivery, serializing the updates at a central location, and dynamically mapping the user's SD server name into its current IP address.

8.3 Related Work

8.3.1 Comparison to "Legacy" Applications

We compared SD Handler with Email, CVS, and FTP, according to the seven desiderata listed in section 8.1.2.

```
<SELFDATA owner="B.Hoon Kang"  UAN =”dlib2001 selfdata paper”
    archivalsupport="yes“ accesscontrol="yes“ change-notification="always">
<UAN name = “dlib2001 selfdata paper”>
    <ITEM location = “selfdata01.doc”/>
    <ITEM location = “diagram-image01.gif”/>
    <ITEM location = “diagram-image02.gif”/>
</UAN>
<SHARER coherency = "SERIALIZE" data-availability="high" >
    <USER name="B. Hoon Kang" id="hoon@cs.berkeley.edu">
            <SELFDATASERVER name="alpine.cs.berkeley.edu:7070"  />
            <SELFDATASERVER name="sb.index.berkeley.edu:7070" />
            <ARCHIVALSERVER name="dlibarchiver.cs.berkeley.edu:7070" />
    </USER>
    <USER name="Robert Wilensky"   id="wilensky@cs.berkeley.edu" >
            <SELFDATASERVER name="bonsai.cs.berkeley.edu:7070" />
            <SELFDATASERVER name="mobile-ip.cs.berkeley.edu:7070" />
            <SELFDATASERVER name="home-ip.eecs.berkeley.edu:7070" />
            <ARCHIVALSERVER name="myarchiver.eecs.berkeley.edu:7070" />
    </USER>
     <CENTRALSERVER  name="galaxy.cs.berkeley.edu:7070" />
</SHARER>
</SELFDATA>
```

Figure 8.4: SDD Example: This shows an SDD example owned by the user “B. Hoon Kang”, and shared with two users “B. Hoon Kang” and “Robert Wilensky”. The user “Robert Wilensky” contains four different SelfData (SD) server locations, the last of the four being his own archival server. The archival server is an instance of SD server where the SD Handler governs the archival repository for its subscribed users. The owner “B. Hoon Kang” provides its own archival server to be accessible by the sharer.

P1. No repeat user involvement in routine data management All of the applications except SD Handler require repeated user involvement in copying, moving, and sending the data. SD Handler requires SDD creation once for repeated usages.

P2. No unnecessary dependence on shared resources, such as shared data repositories or file servers Email does not require shared resources for collaboration; CVS require a shared server location.

P3. No prior administrative set up costs Both CVS and FTP provide the password-controlled access to the data that is being shared among collaborators. Either group account or individual account needs to be set up by an administrator, and need to be distributed to each collaborator to access the data prior to the collaboration. In Email, ICQ (www.icq.com) and AIM (www.aim.com), however, the password is not required to send or receive the message and its attached data. The access is purely controlled by the user's discretion whether to accept or refuse the attachment. An orthogonal end-to-end security method, for example, PGP (Pretty Good Privacy) email, could be added. Both the SD Handler and Groove (www.groove.net) provide public/private key based access control to the data without requiring prior administrative account set up. The user's discretion is guided by the key issuer's certificate or web-of-certificates.

P4. Ability to exploit minimal use of central server as only required ICQ utilize this property to support the scalable use of central data server. The central data server is minimally used only for name resolution of recipient's current IP address and store-and-forward data delivery to the unavailable recipient. By this measure, web-based file sharing systems over-utilize their central server in terms of the network bandwidth, processing power and disk space.

P5. Undo/Redo capability within user's domain CVS and SDH support this property.

P6. Secure and safe incorporation of updates at user's domain Email could use DSA (Digital Signature Algorithm) for end-to-end security but the incorporation of email attachment is not sand-boxed. ICQ and FTP do not provide safe-guarded incorporation either. CVS's undo capability could provide a safe incorporation since one can go back to the previous change in the case of an incorporation error.

P7. Lightweight enough to be widely deployed All the applications above are considered to be lightweight since they do not require a heavyweight server infrastructure.

8.3.2 Declarative vs. Session-Based Data Management

FTP, NFS, HTTP, and derivative applications (e.g. WebDAV) require a session with a resource controlling a data object in order to create, update, move, delete, or otherwise manage that object. Moreover, during this session, the data are managed by procedural commands. Network file systems, e.g., AFS and NFS, basically provide file semantics in sharing data, so, once again, intentions are expressed procedurally. Ficus [34] , Bayou [49] (peer-to-peer optimistic file replication), and Rumor [20](user-level replication system) use file sharing semantics, and hence are fundamentally procedural as well. Also the overwriting semantics of file systems does not provide the knowledge about who made which changes. Hence one would have to use versioning software like CVS in an explicit way, requiring the user's involvement in setting up check-in, check-out and copying.

In contrast, the SD Handler model provides a declarative way of managing data across administrative domains in a wide area scale. The SD Handler model also enables the user to specify that the data needs to be versioned at the different administrative domains. We believe this model can simplify data management, achieving our goals of minimizing the user's participation in routine tasks.

8.3.3 Scripted Email Attachment

A SD Handler can perform incorporation of received data into the recipient's internal data storage. A similar effect can be achieved by running a script (VBScript, UNIX shell script) with an email attachment. However, as is well-known, doing so is dangerous since the script can run any arbitrary command. However, the SD Handler's incorporation operation is sand-boxed within SD Handler's address boundary where the access is limited only through the sanitized SD Handler's incorporation functionality.

8.3.4 P2P (Peer to Peer) Collaborative Systems

ICQ, AIM provide a peer to peer instant messaging with infrastructure services such as identity (user account) management, store-and-forward delivery and dynamic mapping of user's current IP address. We have found that these infrastructure services are common to most P2P systems. For example, Groove provides collaborative P2P software tools with just these infrastructure services. The "shared space" in Groove provides an interactive collaborative environment where various applications (tools) can be built upon such as instant messaging, file sharing, free form drawing, and real-time conversation.

However, unlike SD Handler, the management of data still requires repeated user interactions. The delivered files (attachments) have to be manually downloaded and saved.

Versioning and logging are not provided since the incorporation of data is not automated but depends on manual end-user commands. Moreover, Groove and ICQ/AIM assume each peer to be an end user; in SD Handler the peer could be a personal repository server, a back up server, and a device in addition to other desktop users.

Finally, Groove is focused on building a collaboratively shared space (or workspace) in a P2P way. SD Handler is focused on providing new semantics and controls for managing data with minimal user interactions. The co-authoring application is an example of using self-administering data model in a collaborative scenario.

8.4 Summary

We described a self-administering data model, in which a declarative description of how a data object should behave is attached to the object, either by a user or by a data input device. This self-administering data model assumes an widespread infrastructure of self-administering data handlers (SDH). S2D2 can enable the required infrastructure by building SDH based on S2D2. The handlers (SDHs) are responsible for carrying out the specifications attached to the data. Typically, the specifications express how and to whom the data should be transferred, how it should be incorporated when it is received, what rights recipients of the data will have with respect to it, and the kind of relation that should exist between distributed copies of the object.

We suggest that this model can provide superior support for common cooperative functions. Because the model is declarative, users need only express their intentions once in creating a self-administering description, and need not be concerned with manually performing subsequent repetitious operations. Because the model is peer-to-peer, users are less dependent on additional, perhaps costly resources, at least when these are not critical.

Chapter 9

Summary and Conclusion

In this dissertation, we identified the needs for scalable and secure optimistic replication to effectively support global-scale server replication, to enable open collaboration using shared data, and to manage pervasive/ubiquitous data replication. Then, we showed that the design of a scalable and secure decentralized ordering mechanism is the foundation for building a system capable of supporting such optimistic replication. To this end, we developed a novel decentralized ordering mechanism, called Summary Hash History (SHH). SHH uses summary hashes as version identifiers to provide simple management in site membership changes, scales regardless of the number of sites, and guarantees the correctness of decentralized ordering in a scalable way.

Using summary hashes as version identifers provides SHH with many useful properties. Because of the collision resistant property of the hashing function that we use for creating a summary hash, a site can always verify the associated history/data in a tamper-evident way. This verifiable summary hash allows two-step SHH reconciliation and aggressive log pruning utilizing SHH's secure log reconstruction property.

SHH can be thought of a kind of causal history approach that overcomes the an inherent limitation of a straightforward realization of causal history (i.e., unbounded growth of history) while providing decentralized ordering correctness and verifiability. Being based on the causal history approach, the size of a SHH that needs to be exchanged grows in proportion to the number of update instances. To effectively manage this overhead, we developed two-step SHH reconciliation, which makes SHH a practical technique. Only the latest summary hashes in SHHs are exchanged frequently; the data/SHHs can be lazily pulled from any local site, with the latest summary hash verifying the associated data and SHH. Since we can verify the validity of the given history, the data/SHH need not come from trusted remote sites.

The verifiable nature of summary hash provides us with many other optimization opportunities. For example, with a verifiable summary hash, a site can aggressively prune its log history. The pruned log history can be reconstructed incrementally and securely from the latest summary hash. This secure log reconstruction property of summary hash allows the SHH to be pruned aggressively, which is especially useful in supporting optimistic replication among sites with limited storage.

SHH's ability of capturing a coincidental equality is a fascinating property that we found. It can allow distributed replicas to converge even across partitioned networks. Such ability also enables distributed replica to converge faster than other mechanisms that may be susceptible to creating false conflicts among descendant versions. (SHH creates absolutely no false conflicts.)

After the simulation study of SHH properties, we concluded that SHH can be a technique of choice for designing a scalable and secure optimistic replication. Our simulation confirmed that SHH with two-step reconciliation consumes orders of magnitude lower bandwidth than previous version vector based approaches. Our simulation also showed that replicas can converge faster than other mechanism that are prone to create false conflicts. We find it rather remarkable that SHH is able to provide scalability, security and fast convergence all at once by simply using summary hash as version identifier.

We believe it would be interesting to apply SHH to application domains other than optimistic replication. For example, SHH's ability to converge during network partition with no false conflicts can be utilized for scaling the performance of distributed parallel computations. SHH may reduce the amount of network traffic required to synchronize partial computation results or may be able to speed up the computation by assigning the same identifier for the independently created partial computation results that are equivalent.

We are currently deploying SHH-based optimistic replication technique to various distributed systems applications via the S2D2 framework that we developed. S2D2 is a framework with which diverse applications can readily implement SHH-based optimistic replications. S2D2 employs an adaptor architecture to support the application-specific data management. Being based on SHH, S2D2 provides various service components that are required for global-scale optimistic replication: the two-step SHH reconciliation, a global secure naming for shared object and an access control mechanism based on a PKI (Public Key Infrastructure). In implementing S2D2/SHH, we developed a hash typing principle, which is proven to be useful in enforcing the correctness in the error-prone usage of various

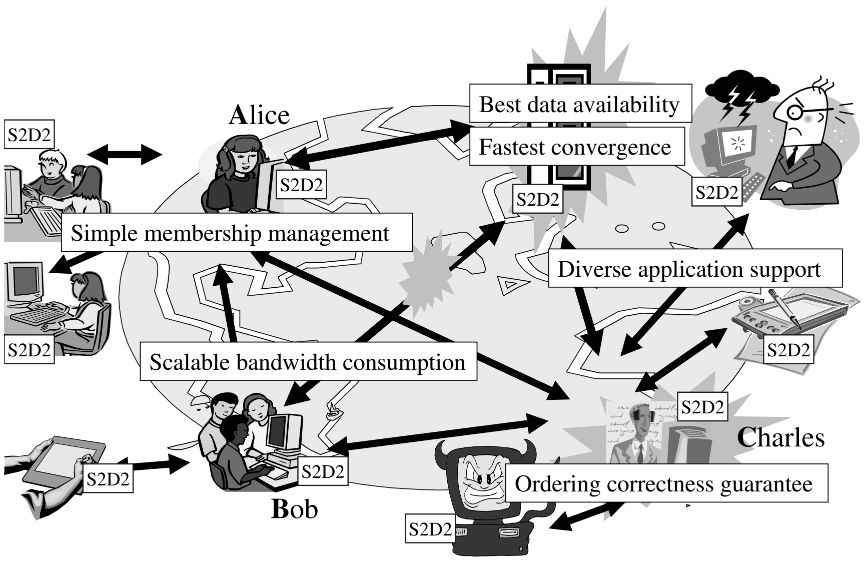

Alice
Bob
Charles
S2D2
Best data availability
Fastest convergence
Simple membership management
Diverse application support
Scalable bandwidth consumption
Ordering correctness guarantee

Figure 9.1: Scalable and Secure Optimistic Replication based on S2D2: Being based on SHH, S2D2 can provide scalable bandwidth consumption, ordering correctness guarantee, faster convergence, simple membership management, and high data availability to diverse applications with global-scale and secure optimistic replication support.

hashes and to identify composite object's structural information for scalable data exchanges. Figure 9.1 summarizes what S2D2 can provide: scalable bandwidth consumption, ordering correctness guarantee, faster convergence, simple membership management, and high data availability to diverse and pervasive applications.

We believe SHH-based optimistic replication, as realized via S2D2 framework, is a fundamental technique that can provide the scalability and security required for supporting global-scale, highly dynamic, and pervasive/distributed systems.

Bibliography

[1] *The IceCube approach to the reconciliation of divergent replicas*, 2001.

[2] A. Adya et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th USENIX Symposium on OSDI*, Boston, Massachusetts, December 2002.

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

[4] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Version stamps- decentralized version vectors. In *Proc. of IEEE ICDCS*, 2002.

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

[6] D. Bayer, S. Haber, and W. S. Stornetta. Improving the efficiency and reliability of digital time-stamping. In *Sequences'91: Methods in Communication, Security, and Computer Science*, pages 329–334. SpringerVerlag, 1992.

[7] B. Berliner. CVS II: Parallelizing software development. In *Proceedings of the USENIX*

Winter 1990 Technical Conference, pages 341–352, Berkeley, CA, 1990. USENIX Association.

[8] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

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

[10] 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.

[11] L. Cox and B. Noble. Fast reconciliations in fluid replication. In *The 21st International Conference on Distributed Computing Systems, April 2001, Phoenix, AZ*, pages 449–458.

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

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

[14] Ivan Bjerre Damgård. A design principle for hash functions. In *Proceedings on Advances in cryptology*, pages 416–427. Springer-Verlag New York, Inc., 1989.

[15] A. J. Demers, K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and B. B.

Welch. The bayou architecture: Support for data sharing among mobile users. In *Proceedings IEEE Workshop on Mobile Computing Systems & Applications*, pages 2–7, Santa Cruz, California, 8-9 1994.

[16] 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.

[17] 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.

[18] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems (TOCS)*, 20(1):1–24, 2002.

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

[20] Richard G. Guy, Peter L. Reiher, David Ratner, Michial Gunter, Wilkie Ma, and Gerald J. Popek. Rumor: Mobile data access through optimistic peer-to-peer replication. In *ER Workshops*, pages 254–265, 1998.

[21] Stuart Haber and W. Scott Stornetta. How to time-stamp a digital document. *Journal of Cryptology*, 3(2):99–111, 1991.

[22] IETF Crypto Forum Research Group. Discussion re: Merkle hash tree weakness - request for advice. https://www1.ietf.org/mail-archive/working-groups/cfrg/current/index.html.

[23] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Building reliable mobile-aware applications using the rover toolkit. In *Second ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, 1996.

[24] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 670. IEEE Computer Society, 2003.

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

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

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

[28] Petros Maniatis and Mary Baker. Secure History Preservation Through Timeline Entanglement. In *Proc. of the 11th USENIX Security Symposium*, 2002.

[29] Ralph C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, pages 218–238. Springer-Verlag New York, Inc., 1989.

[30] R.C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology Crypto '87*, 1987.

[31] Athicha Muthitacharoen, Robert Morris, Thomer Gil, and Benjie Chen. Ivy: A

read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[32] Brian Noble, Ben Fleis, Minkyong Kim, and Jim Zajkowski. Fluid replication. In *The Network Storage Symposium, Seattle, WA, 14-15 Oct., 1999.*

[33] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. The sprite network operating system. *Computer Magazine of the Computer Group News of the IEEE Computer Group Society, ; ACM CR 8905-0314*, 21(2), 1988.

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

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

[36] Thomas A. Phelps and Robert Wilensky. Multivalent annotations. In *European Conference on Digital Libraries*, pages 287–303, 1997.

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

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

[39] David Ratner, Peter Reiher, and Gerald J. Popek. Dynamic version vector main-

tenance. Technical Report CSD-970022, University of California, Los Angeles, June 1997.

[40] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.

[41] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *Proc. of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

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

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

[44] Jonathan S. Shapiro and John Vanderburgh. Access and integrity control in a public-access, high-assurance configuration management system. In *USENIX Security Symposium, 2002, San Francisco, CA, 2002.*

[45] Jonathan S. Shapiro and John Vanderburgh. Cpcms: A configuration management system based on cryptographic names. In *USENIX Annual Technical Conference, FreeNIX Track, Monterey, CA, 2002.*

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

[47] Mike J. Spreitzer et al. Dealing with server corruption in weakly consistent, replicated data systems. In *Proc. of the third annual ACM/IEEE international conference on Mobile computing and networking*, pages 234–240, 1997.

[48] Mike J. Spreitzer et al. Dealing with server corruption in weakly consistent replicated data systems. *Wireless Networks*, 5(5):357–371, 1999.

[49] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Symposium on Operating System Principles*, pages 172–183, December 1995.

[50] Douglas B. Terry, Karin Petersen, Mike Spreitzer, and Marvin Theimer. The case for non-transparent replication: Examples from bayou. *Data Engineering Bulletin*, 21(4):12–20, 1998.

[51] Gene Tsudik. Message authentication with one-way hash functions. In *INFOCOM (3)*, pages 2055–2059, 1992.

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

[53] E. James Whitehead, Jr. and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the web. In *Proc. of the Sixth European Conf. on Computer Supported Cooperative Work (ECSCW'99), Copenhagen, Denmark, September 12-16, 1999*, pages 291–310.

[54] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *Proc. of ACM SOSP*, 2001.

[55] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proceedings of IEEE INFOCOM*, 1996.