Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data

Michael B. Greenwald[1], Sanjeev Khanna[2], Keshav Kunal[2],
Benjamin C. Pierce[2], and Alan Schmitt[3]

[1] Bell Labs, Lucent Technologies
[2] University of Pennsylvania
[3] INRIA

Abstract. Current techniques for reconciling disconnected changes to optimistically replicated data often use version vectors or related mechanisms to track causal histories. This allows the system to tell whether the value at one replica dominates another or whether the two replicas are in conflict. However, current algorithms do not provide entirely satisfactory ways of *repairing* conflicts. The usual approach is to introduce fresh events into the causal history, even in situations where the causally independent values at the two replicas are actually equal. In some scenarios these events may later conflict with each other or with further updates, slowing or even preventing convergence of the whole system.
To address this issue, we enrich the set of possible actions at a replica to include a notion of explicit conflict resolution between *existing* events, where the user at a replica declares that one set of events dominates another, or that a set of events are equivalent. We precisely specify the behavior of this refined replication framework from a user's point of view and show that, if communication is assumed to be "reciprocal" (with pairs of replicas exchanging information about their current states), then this specification can be implemented by an algorithm with the property that the information stored at any replica and the sizes of the messages sent between replicas are bounded by a polynomial function of the number of replicas in the system.

1 Introduction

Some distributed systems maintain consistency by layering on top of a consistent memory abstraction or ordered communication substrate. Others—particularly systems with autonomous nodes that can operate while disconnected—must relax consistency requirements to make progress, depending instead on a notion of *causal history* of events. If a replica learns of different updates to the same object, then the most causally recent update is considered "best" and is preferred over the others. However, if it happens that the replicas held at two sites are modified simultaneously, then neither update will appear in the other's causal history, and neither these sites nor any others that hear from them will be able to prefer one update over the other until the conflict has been *reconciled.*

In standard approaches based on causal histories (e.g. [1, 2]), this reconciliation is itself an *event*— a new update that causally supersedes all of the conflicting ones. Unfortunately, this reconciliation event can create new conflicts. Until it propagates through the whole system, any update created on another replica before it hears of the resolution will be causally unrelated to the reconciliation event and will thus conflict with it. Indeed, as has been noted before [1, 3], in some systems, the very same conflict might be resolved, independently, by inserting new reconciliation events at different sites, thus raising new conflicts even though the reconciled values may be identical. Most existing systems have found this potential behavior acceptable in practice—conflicts are infrequent or communication frequent enough to ensure that reconciliation events usually propagate throughout the system quickly. However, in some settings (described in detail below), conflicts due to reconciliation events can delay convergence or force users to manually reconcile the same conflict multiple times.

To improve the convergence behavior of such systems, we propose adding a new kind of *agreement event* that labels a set of updates as equivalent, together with a mechanism for declaring that one existing event dominates another. Our goals are to reduce the number of user interventions needed to bring conflicting updates into agreement and to speed global convergence after conflict resolution.

Beyond Causal Histories. Standard causal histories are an attractive way of prioritizing events in a distributed system, partly because they capture a natural relationship between updates and partly because their causal relationships can be represented very efficiently. In particular, it is well known that causal histories can be efficiently summarized using *vector clocks* [1]. Each replica R^α maintains a monotonically increasing counter n^α that is incremented at least once per update event on R^α. Each R^α also maintains a vector (the vector clock), indexed by replica identifiers β, that indicates the latest update of R^β that R^α has heard about (all previous updates of R^β are also in the causal history of R^α). If each update is associated with the local vector clock at the time of its creation, then we can determine the causal relationship between two events: if every entry in one vector clock c_1 is less than or equal to the corresponding entry in another vector clock c_2, then the update v_1, corresponding to c_1, is in the causal history of the update v_2, corresponding to c_2, and v_2 may safely overwrite v_1.

To record the resolution of a conflict using vector clocks, the local vector clock must be changed to reflect the fact that all the conflicting updates are now in the causal past. This can be achieved by first setting the local vector clock to the pointwise maximum of all the vector clocks associated with the conflicting updates and then incrementing the local counter [1].

Unfortunately, this technique can give rise to situations where the system cannot stabilize without further manual intervention—or indeed, in pathological cases, where it can never stabilize. In particular, if, at any point in time, two distinct sites resolve a conflict, even in an identical way, the system will consider the two identical resolutions to be in conflict. Consider the example in Figure 1. From an initial state where all replicas are holding the same value (ϵ), replicas R^a, R^b, and R^c all independently set their value to x at (local) time 1. Although

	Replica R^a		Replica R^b		Replica R^c	
Event	Local time	value (vc)	Local time	value (vc)	Local time	value (vc)
	0	ϵ (0,0,0)	0	ϵ (0,0,0)	0	ϵ (0,0,0)
Local updates	1	x (1,0,0)	1	x (0,1,0)	1	x (0,0,1)
$R^a \rightarrow R^c$ and $R^b \rightarrow R^c$	1	x (1,0,0)	1	x (0,1,0)	2	x (1,1,2)
$R^a \rightarrow R^b$	1	x (1,0,0)	2	x (1,2,0)	2	x (1,1,2)

Fig. 1. A case in which vector clocks never converge, although all replicas hold the correct value.

all replicas "agree" in the sense that they are holding the same value, the system will only stabilize if every replica communicates its state (perhaps indirectly) to a single site, that site creates a new update event (with a vector clock that is greater than the pointwise maximum of all 3), and this new event gets communicated back to all the other sites before anything else happens. In Figure 1, the replicas do successfully transfer their state to R^c, which creates an event that could stabilize the system. Unfortunately, R^a also sends its state to R^b. In response to this badly timed message, R^b creates an event that resolves the conflict between R^a and R^b but conflicts with the agreement event generated at R^c. Neither R^c nor R^b's state now dominates the other's, and the system cannot converge until the new conflict between R^c and R^b is repaired.

A natural idea for improving matters is to allow a reconciling site to introduce an *agreement event* that somehow "merges" two causally unrelated updates instead of dominating them. Then if R^c declares that the update events at replicas R^a, R^b, and R^c are all equivalent, and later R^b declares that the events at replicas R^a and R^b are equivalent, the two reconciliations will not conflict.

Agreement events raise issues, however, that cannot be modeled naturally by causal histories. It may appear that agreements that may be helpful in the example above might be implemented by simply having the reconciling site *not* increment its local timestamp after taking the pointwise max of its vector clock with that of the other conflicting replicas; then two reconciliations at different hosts would not conflict. (In the example above, R^c would set its clock to $(1, 1, 1)$ and R^b would later set its to $(1, 1, 0)$—i.e., the reconciled state from R^c would dominate the "partially reconciled" state from R^b.)

However, this scheme is still not satisfactory: if any new updates happen before the reconciliation event(s) propagate completely through the system, spurious conflicts will still be created. Figure 2 shows what can happen. The three replicas, R^a, R^b, and R^c, again begin by all taking on the value x. Later, R^a sends a message to R^b, which reconciles the conflict between their (identical) values by merging R^a's vector clock with its own, yielding (1,1,0). Later, R^b sends a message to R^c, which similarly recognizes that their conflicting values are equal and updates its local clock to (1,1,1). If, at this point, R^c were to send its state to R^a and R^b before anything else happened, all would be well.

	Replica R^a		Replica R^b		Replica R^c	
Event	Local time	value (vc)	Lcl time	value (vc)	Lcl time	value (vc)
Initial state	0	ϵ (0,0,0)	0	ϵ (0,0,0)	0	ϵ (0,0,0)
Local updates	1	x (1,0,0)	1	x (0,1,0)	1	x (0,0,1)
$R^a \to R^b$	1	x (1,0,0)	1	x (1,1,0)	1	x (0,0,1)
$R^b \to R^c$	1	x (1,0,0)	1	x (1,1,0)	1	x (1,1,1)
R^a updated	2	y (2,0,0)	1	x (1,1,0)	1	x (1,1,1)

Fig. 2. A case in which vector clocks “forget” a resolution event.

However, suppose instead that R^a locally updates its value to y. This update clearly supersedes the first update of x on R^a; also, since the value of x on R^b has been reconciled with the old x on R^a, the new update of y at R^a should also supersede the x on R^b, and similarly on R^c. However, at this point the system is totally stalled, although it is clear (to an omniscient observer) that all replicas should converge to y. No sequence of messages will ever reconcile R^a with either R^b or R^c. (Note that the value on R^c is not in the causal history of y, even if *both* the sender and receiver update their local clocks after communication.)

In a similar vein, vector clocks and standard causal histories provide no way of reconciling a conflict by simply declaring that one of the conflicting events is *better* than the others. For example, suppose replicas R^a and R^b are independently updated with conflicting values and each communicates its value to some large set of other nodes before anybody notices the conflict. If the user performing the reconciliation decides that R^a's value is actually preferable to R^b's, they would like to be able to declare this to the system so that, with no further intervention, every host that hears about both updates will choose R^a's value. Moreover, if, in the meantime, some host that heard about R^a's update has made yet a further update, this new value should also automatically be preferred over R^b's.

These shortcomings are not an artifact of a vector clock representation; the system stalls because causal histories do not remember equivalences between events. If R^c declares that the values at R^a, R^b and R^c are equivalent, and R^a simultaneously decides that the value y is preferable to its current value x, then we want the system to prefer one causally unrelated value to another. There is no way to put the value at R^c into the causal history of R^a. (We will see later that attempting to simply add equivalence edges can causes cycles in the causal history graph. Those cycles, in turn, can give rise to paradoxical behavior.)

Such scenarios become more likely as the frequency of updates (and hence conflicts and reconciliations) increases, relative to the speed with which information propagates between nodes. Thus, in systems where conflicts are rare, or where nodes are tightly coupled and communicate frequently, vector clock solutions are likely to be satisfactory; on the other hand, in systems where conflicts are more frequent and/or communication more intermittent, more sophisticated solutions, such as the one we propose here, may perform significantly better.

(We explain in the next section how our proposal, which combines agreement and dominance declarations, smoothly handles the examples in Figures 1 and 2.)

Harmony: A Motivating Application. Our interest in conflict resolution algorithms originates in our work on Harmony [4,5], a generic "data synchronizer," capable of reconciling data from heterogeneous, off-the-shelf applications that were developed without synchronization in mind. For example, Harmony can be used to synchronize collections of bookmarks from several different browsers (Explorer, Safari, Mozilla, or OmniWeb), or to keep appointments in MacOS X iCal or Gnome Evolution up-to-date with our appointments in Palm Datebook or Unix ical formats. The current Harmony prototype is able to synchronize only pairs of replicas, with pairwise reconciliation triggered by explicit user synchronization attempts such as putting a PDA into a cradle (perhaps attached to a disconnected laptop). This scheme extends fairly smoothly from pairs to small collections of replicas by iterated pairwise synchronization, but becomes awkward as the set of replicas grows. The work in this paper was inspired by the goal of extending Harmony to handle large numbers of replicas.

Several features of Harmony conspire to make conflicts likely to appear relatively frequently. First, because of its loose coupling with the applications whose data it reconciles, Harmony is a state-based reconciliation system [6]. Unlike operation-based systems, where the system keeps a log of all operations and may be able to resolve conflicts by merging the operation logs on two replicas, state-based systems cannot, in general, merge updates that modified the same atomic values. Second, Harmony reconciles updates between systems such as PDAs that may operate disconnected for long periods of time. Third, we have observed that, even with small numbers of replicas, it often happens that identical updates are entered at different nodes—particularly when the same user owns multiple devices.

Our Results. Since causal histories are not able to satisfactorily handle reconciliation in systems such as Harmony, we develop in this work a new reconciliation framework offering notions of both dominance and agreement, allowing users to resolve conflicts by explicitly specifying the prior events they want to take into account. In §2 we specify this framework precisely by defining legal sequences of local updates, dominance and agreement events, and communications between replicas and showing how to calculate, at each replica, which events will be reported as "maximal" and which as "conflicting."

Our main contribution, in §3, is an algorithm implementing our specification under the assumption that communication is "reciprocal"—after one replica has sent its current state to another, it will wait for a message from the other before sending its own state to that replica again. This algorithm has the property that the information stored at any replica and the sizes of the messages sent between replicas are bounded, in the worst case, by a polynomial function ($O(n^4)$, to be precise) of the number of replicas in the system. §4 discusses related work. Omitted proofs can be found in an accompanying technical report, available on the Harmony home page [5].

2 An Agreeable Reconciliation Framework

A reconciliation framework has three choices when comparing the same object on two different replicas. It can decide that the two objects have *equivalent* values, and do nothing. It can decide that one value is *better than* the other, and modify one replica. Or, it can decide that the two objects are *in conflict* and require external reconciliation. Our goal is to design a consistency maintenance mechanism that can reduce the number of objects that the system decides are in conflict, with less user intervention than conventional causal histories.

The key to achieving this is recognizing "agreement events" as first class citizens. A reconciliation system based on causal history, implements the better-than relation through causal order: u is better-than v if v is in the causal history of u, they are equivalent only if they are identical, and in conflict if u and v are causally unrelated. In our framework it is no longer the case that the simple fact of a node knowing about an event implies that a new update event at that node is better than that prior event — instead we offer a richer 'better-than" relation (defined formally at the end of this section). The user may declare that two or more updates *agree*, or that an update *dominates* another update, or leave two updates unrelated. The system remembers these declarations, so that, if an update u is better-than another update v then u is also better-than all updates equivalent to v, even if they are not in the (conventional) causal history of u or v. Rather than basing our notion of better-than simply on a "knows about" relation (i.e., causal order), we now require users to specify whether the new update u "took v into account" (defined formally below) and, if so, whether through agreement or domination. Agreement events introduce the possibility that two distinct events can be considered equivalent.

This seemingly small shift raises a rather subtle new issue. By introducing "equivalence" we allow the possibility of cycles in the graph of the took-into-account relation. Consider a scenario where two conflicting values x and y were both known about by two different replicas. One decided that y was better than x; the other decided that x was better than y. When the replicas communicate with each other, they discover a cyclical took-into-account relation. Such cycles represent a new sort of conflict—a situation in which users at two or more replicas have given the system conflicting guidance about how to repair a previous conflict! How should we treat such cycles of taking-into-account? In general, there may be multiple distinct values in the cycle, so we cannot pick a single value from the cycle that the system should converge to. The question, then, is not how the values in the cycle relate to each other, but how other values relate to the cycle—i.e., how we can resolve this conflict and allow the replicas to converge by finding or creating values that are not taken into account by others. We address this issue with the notion of *dominance* defined later in this section.

Preliminaries. We assume a fixed set of n replicas, called R^a, R^b, etc. (The development extends straightforwardly to a dynamically changing set of replicas. The main challenge is discovering when information about replicas that have left the system can be garbage collected; standard techniques used in vector clock systems should apply.) The variables α, β, etc. range over indices of replicas. For simplicity, we focus on the case where each replica holds a single, atomic value.

External actions (by the user or a program acting on the user's behalf) that change the value at some replica are represented as *events*, written v_i^α, where α is the replica where the event occurred and i is a local sequence number that distinguishes events on replica α.

An event is a *predecessor* of all local events that occur after it—that is, v_i^α is a predecessor of all v_j^α with $j > i$; similarly, v_i^α is a *successor* of all events v_j^α with $j < i$. We use v_{i+}^α and v_{i-}^α as variables ranging over successor and predecessor events of v_i^α. When the location or precise local sequence number of an event are not important, we lighten notation by dropping super- and/or subscripts, writing events as just v, v^α, v_+^α, v_-^α, etc.

Our specification uses a structure called a *history graph* (or just *graph*) to represent the state of knowledge at a particular replica at a particular moment in the whole system's evolution. A history graph is a directed graph whose vertices are events and whose edges represent "took into account" relations between events. There are two kinds of edges: an edge $v \longrightarrow w$, pronounced "*v takes w into account through dominance*," represents the fact that event v was created taking w into account and dominating it, while an edge $v \Longrightarrow w$, pronounced "*v takes w into account through agreement*," represents the fact that v and w were declared in *agreement* by the creator of v. (Note that we are not necessarily requiring that v and w have the same *value* in order to be declared in agreement; typically they will, but it may sometimes be useful to resolve a conflict between different values by declaring that either one is acceptable and there is no need for every replica to converge to the same one.) We use G^α to denote the history graph for replica R^α. The set of events in G^α at any given moment is the set of events in the standard causal history of R^α (in contrast, the set of edges in G^α may be only a subset of the set of edges representing causal order).

The set of events and edges reachable in a graph G from an event v, including v itself, is called the *cone* of v, written *cone*(v). This set represents the events v transitively took into account when it was created. We will maintain the invariant that edges originating at an event can be created only at its time of creation, so that the set of events reachable from v will not change over time; moreover, because entire history graphs are exchanged when replicas communicate (at the level of the specification, though of course not in the implementation we describe later), any graph G that contains v will also include *cone*(v); for this reason, we do not bother annotating *cone*(v) with G.

Another important invariant property is *equivalence*. We first define $G_\equiv^\alpha$, the graph obtained from G^α by symmetrizing its $\Longrightarrow$ edges, adding an edge $v \Longrightarrow u$ for each existing edge $u \Longrightarrow v$. Two events u and v are now said to be *equivalent* in G^α if there is a path from u to v in $G_\equiv^\alpha$ consisting only of $\Longrightarrow$ edges. Because replicas exchange whole history graphs, if two events become equivalent at some point in time in the history graph at some replica R^α, they will remain equivalent at all replicas that ever hear (transitively) from R^α. We refer to the partitions induced by this equivalence as *equivalence classes*, or just *classes*.

For a pair of classes E and E', we say E takes E' into account if there exist events $x \in E$ and $y \in E'$ with $y \in \mathit{cone}(x)$. We noted above that there can be cycles in the took-into-account relation: two distinct equivalence classes may each contain an event that has an event from the other in its cone. For example, suppose that the latest (conflicting) values in replicas R^a and R^b are v_i^a and v_j^b, respectively, and that G^a and G^b both contain the complete system history. R^a tries to reconcile the conflict by *adopting* the value of v_j^b (by creating an event v_{i+1}^a with the same value as v_j^b and declaring v_{i+1}^a to be in an equivalence class E with v_j^b). R^b tries to reconcile the conflict by similarly adopting the value of v_i^a, by putting v_{j+1}^b in an equivalence class E' with it. E takes E' into account, because v_i^a is in the cone of v_{i+1}^a; similarly, E' takes E into account because v_j^b is in the cone of v_{j+1}^b. We call such situations *reconciliation conflicts*, since they arise when users at different replicas make different decisions about which of a set of conflicting events should be preferred.

In general, a class can belong to multiple cycles—i.e., it can be involved simultaneously in multiple reconciliation conflicts. To arrive at a clear notion of "better-than", we will define a *dominance* relation. We consider strongly connected components of the graph $G^\alpha_\equiv$ (i.e., sets of events such that there is some path from every event in the set to every other event in the set), which we refer to simply as *components*. Every pair of classes in a component belongs to some cycle denoting a reconciliation conflict, and so intra-component "took into account" relations between events cannot be used to determine dominance.

Now, a class E is said to *dominate* a class E', written $E > E'$, if E and E' belong to different components and there exist events $x \in E$ and $y \in E'$ with $y \in \mathit{cone}(x)$. Note that $E > E'$ implies $E' \not> E$ because of the assumption that the two are in different components.

We say that an event $v_i^\beta \in G^\alpha$ is *latest* if no successor event v_{i+}^β belongs to G^α. We are particularly interested in events belonging to classes that are not dominated by other classes and, among these, in the ones that are latest: if the entire system is going to converge to a single value (or set of equivalent values), such events are the only possible candidates. Formally, we say that a class E is a *maximal class* if it contains a latest event and there is no class E' with $E' > E$. An event v is a *maximal event* if it is a latest event in a maximal class.

When can a replica R^α conclude that there is no conflict between the values in G^α? Based on our definition of dominance, it is easy to see that, if all maximal events belong to the same (maximal) class E, we can be sure that the events in E took every event in G^α into account and that no other events took them into account, implying that there is no conflict between these events (at least according to the present local state of knowledge) and that these events are "better than" all other events. Rule 3 in the specification below guarantees that R^α will then adopt an event from E.

Let us see how our model applies to the examples we discussed in §1. The initial values at the replicas are represented by v^a, v^b and v^c respectively. For the example in Figure 1, after receiving state updates from R^a and R^b, R^c joins v^a, v^b, and v^c into an equivalence class by creating a new event v_+^c and adding

$\Longrightarrow$ edges from v^c_+ to them. Independently, R^b, after receiving R^a's state, makes v^a, v^b and v^b_+ into an equivalence class. Fortunately, these new events v^c_+ and v^b_+ do not conflict, and anyone who later hears of both can calculate that v^a, v^b, v^c, v^b_+, and v^c_+ all belong to the same equivalence class, so that any new event dominating any of them will also dominate all the others. Similarly, in the scenario in Figure 2, R^b makes v^a, v^b, and v^b_+ equivalent and later R^c adds v^c to this equivalence class (via a new event v^c_+ with $\Longrightarrow$ edges to v^c, v^a, v^b, and v^b_+). Independently, R^a adds a new event v^a_+ (with value y), dominating v^a. Henceforth, regardless of the order of messages from R^a and R^c, any replica that learns of both v^a_+ and v^c_+ can see that v^a_+ dominates all the values from the other replicas.

Continuing the example, it is possible that, for some time, some other replica R^d may hear only from R^a and R^b(before R^b creates the event v^b_+) but not R^c and therefore believe that events v^a_+ and v_b are in conflict. Once it hears from R^c as well, the apparent conflict will disappear. But if, in the meantime, the user at R^d decides to repair the apparent conflict by declaring that v^b dominates v^a_+ (by creating an event v^d dominating v^a_+ and then another event v^d_+ in agreement with both v^b and v^d), then a reconciliation conflict will be created, requiring one more user intervention to eliminate.

We have now presented all the basic concepts on which our reconciliation scheme is based. It remains to specify exactly what state is maintained at each replica and how this state changes as various actions are performed. These actions are of two sorts: local actions by the user, and gossiping between replicas, in which one replica periodically passes its state to another, which updates its picture of the world and later sends the combined state along to yet other replicas. We will not be precise in this paper about exactly how replicas determine when and with whom to communicate—we simply treat communication as a non-deterministic transmission of state from one replica to another. (We have in mind a practical implementation based on a gossip architecture such as [7].) However, to ensure that our implementation in §3 can work in bounded space, we need to make one restriction on the pattern of communication: after a replica R^α has sent its state to a particular neighbor R^β, it should wait until it receives an update message from R^β before sending another of its own. (Indeed, in the accompanying technical report we prove that, with unrestricted asymmetric communication, no representation that operates in bounded space can implement the specification correctly.) This *reciprocality* of communication bounds the number of possible open events on each replica. To guarantee reciprocality, each replica maintains a boolean flag $\mathit{CanSend}(\beta)$ for each replica R^β, initially set to true. It is reset to false each time R^α sends a communication to R^β and reset to true each time R^α receives a communication from R^β.(This definition places a somewhat unrealistic constraint on the communication substrate: it assumes that messages are not lost and are not reordered in transit. We believe that this constraint can probably be relaxed, but we do not have a proof yet.)

Specification. The state of the entire system at any moment comprises the following information: a history graph G^α for each replica R^α, a reciprocity predicate

$\mathit{CanSend}^\alpha$ for each replica R^α, and a current event $\mathit{Current}^\alpha \in G^\alpha$ for each replica R^α. The initial state of the system has all history graphs G^α containing a single vertex v_{init} and no edges, $\mathit{CanSend}^\alpha(\beta) = \mathit{true}$ for all α and β, and $\mathit{Current}^\alpha = v_{init}$ for all α. At any given moment, a user (or user-level program) at replica R^α can query the current event at R^α, as well as the current set of maximal events in G^α and, for each of these, the other events in its class.

Each step in the system's evolution must obey one of the following rules:

1. A replica R^α may generate a new event v_i^α, where $i = 1 + \max(j \mid v_j^\alpha \in G^\alpha)$, taking into account some subset W (containing $\mathit{Current}^\alpha$) of the maximal events in G^α. The current event $\mathit{Current}^\alpha$ is set to v_i^α. A vertex v_i^α and an edge $v_i^\alpha \longrightarrow w$ for each $w \in W$ are added to the graph G^α.
2. A replica R^α may generate a new event v_i^α, where $i = 1 + \max(j \mid v_j^\alpha \in G^\alpha)$, and declare it to be in agreement with some subset W of the maximal events in G^α. A vertex v_i^α, and an edge $v_i^\alpha \Longrightarrow w$ for each $w \in W$, are added to the graph G^α. If $\mathit{Current}^\alpha \notin W$ and $\mathit{Current}^\alpha$ is a predecessor of v_i^α, an edge $v_i^\alpha \longrightarrow \mathit{Current}^\alpha$ is also added to the graph. The current event $\mathit{Current}^\alpha$ is then set to v_i^α.
   The choice of W is constrained by one technical condition: Let $E_1 \ldots E_p$ be the maximal classes containing the subset of maximal events W. This operation is allowed only if for each replica R^β, the set of events from the creating replica R^β that will now be in the new merged class, call it E, correspond to a contiguous range of indices—that is, for any $i < j < k$ if $v_i^\beta \in E$ and $v_k^\beta \in E$ then $v_j^\beta \in E$. The interpretation of this restriction is that a user is not allowed to establish agreement between two distinct events v_i^β and v_k^β created by a replica R^β unless it can do so for every event that was created by R^β in between.
3. A replica R^α may send its current state to another replica R^β, provided that $\mathit{CanSend}^\alpha(\beta) = \mathit{true}$. The history graph G^β is replaced by $G^\beta \cup G^\alpha$. A new maximal event x (if one exists) in the combined G^β is *better-than* $\mathit{Current}^\beta$ (and hence overwrites it) if $\mathit{Current}^\beta$ is not a a maximal event in G^β. The reciprocity predicates are updated with $\mathit{CanSend}^\alpha(\beta) = \mathit{false}$ and $\mathit{CanSend}^\beta(\alpha) = \mathit{true}$.

3 A Bounded-Space Implementation

We now develop an efficient implementation based on a *sparse* representation of history graphs, written S^α. The crucial property that we establish is that the size of S^α depends only on the maximum number of distinct replicas that ever communicate with R^α. For analyzing this representation, it is helpful to be able to refer to the local state at any replica at particular points in time. We introduce an imaginary *global time counter* t, which is incremented each time any action is taken by any replica—i.e., each time the whole system evolves one step by a replica taking one of the steps described in §2. The graph at replica R^α at time t is written $G^\alpha(t)$.

There are two core concepts that facilitate our polynomial-space representation of all "relevant" information contained in a history graph. The first is the notion of *open* and *closed* events, and the second is the notion of a *sparse cone* of an event v. We start by decribing these concepts and some of their properties.

Open and Closed Events. The *creator replica* of an event $v = v_i^\alpha$ is the replica R^α at which the event was created. It is clear from the specification that only a creator replica can add edges originating from v to its graph, and only at the time v is created. It can later add an $\Longrightarrow$ edge into v (in addition to the $\longrightarrow$ edge that is always added), when it creates v's immediate successor. Another replica that later hears about v can create $\Longrightarrow$ or $\longrightarrow$ edges into v as long as v is a maximal event in its local graph.

No replica R can afford to forget about an event or any edges from or into it, as long as it is possible for some replica to create edges into it, lest R be the only witness to a relevant equivalence edge. Reciprocal communication enables us to track such "critical" events with bounded space.

An event v is *closed* if, at every replica R^α, if $v \in G^\alpha$ then $v_+ \in G^\alpha$ for some successor v_+ of v; an event that is not closed is *open.* If v is closed, then any replica that hears about v will simultaneously hear about a successor of v. It follows from this that a closed event can never be a latest event at any replica (hence also not a maximal one), and that, once an event is closed, it stays closed forever. No edges can be created to or from a closed event at any replica at any time in the future.

An omniscient observer can see when an event becomes closed. But how can a replica know that an event is closed using only locally available information?

We maintain a data structure O^α at every replica R^α that can be used to certify that events are closed. The creator replica of an event v marks it closed when it knows that all other replicas who ever heard of v, have also heard of a successor to v. The other replicas mark the event closed when they hear that it has been marked closed by the event's creator replica. We say that an event that is marked closed by replica R^α is *closed at* R^α. An event that is not closed at a given replica is *considered open* at that replica.

An event can be simultaneously considered open at certain replicas and closed at others. The data structure O^α ensures that, at any time t, for each non-latest event v_i^α considered open at a replica R^α, we can identify a pair of replicas in the system, say (R^β, R^γ), such that (i) R^γ first learnt about v_i^α from R^β and (ii) R^α is certain that R^β is aware of a successor of v but it is uncertain if this is also the case for R^γ. In this case, R^α can not yet consider v_i^α closed as R^γ may possibly create an edge to the event v_i^α. We refer to such a pair as a *witness* to event v_i^α being open at time t. The reciprocal communication property allows us to ensure that each pair of replicas can serve as a witness to at most two open events from any replica. We use this fact to argue that at most $O(n^3)$ events are considered open at any replica. The data structure O^α maintains $O(n)$ information per open event and hence has size $O(n^4)$. Theorem 3.2 shows that the space complexity of S^α (which includes O^α) is also bounded by $O(n^4)$.

Sparse Cone. The *sparse cone* of an element v_l^α, written *sparse-cone*(v_l^α), can be derived from its cone in the following manner. For each $\beta \neq \alpha$, let j be the largest index, if any exists, such that $v_j^\beta \in$ *cone*(v_l^α). If such a j does exist, then add the vertex v_j^β and a directed edge (v_l^α, v_j^β) to *sparse-cone*(v_l^α).

Note that both *cone*(v) and *sparse-cone*(v) are determined at the time of v's creation and are time invariant. Also, even though *cone*(v) can be arbitrarily large, *sparse-cone*(v) is $O(n)$ in size and implicitly contains all the necessary information from *cone*(v), in the sense that, for any element w, we can determine whether or not $w \in$ *cone*(v) by examining *sparse-cone*(v).

Sparse Representation. We now describe a polynomial-space representation that summarizes the information contained in $G^\alpha(t)$ at any time t. In the accompanying technical report we show how to maintain this representation incrementally as the system evolves, calculating the compact representation at each step from the compact representation at the previous step, and prove that the representation is correct in the sense that it will report the same maximal events (and equivalence classes) as the specification in §2.

We start with the observation that the graph $G^\alpha(t)$ may be viewed as simply a union of the cones of all the elements known to replica R^α at time t. We will represent $G^\alpha(t)$ by a pair of *sparse graphs*, denoted $H^\alpha(t)$ and $H^{\underline{\alpha}}_{\equiv}(t)$. The sparse graph $H^\alpha(t)$ is defined to be simply the union of the sparse cones of latest events known at R^α at time t. It thus takes $O(n^2)$ space. The sparse graph $H^{\underline{\alpha}}_{\equiv}(t)$, summarizes the information contained in $G^{\underline{\alpha}}_{\equiv}(t)$ as follows. Let $v \rightsquigarrow w$ denote the existence of a path from an event v to event w in a graph $G^{\underline{\alpha}}_{\equiv}$. For each open event v_i^β at $R^\alpha(t)$, $H^{\underline{\alpha}}_{\equiv}(t)$ records, for every other replica R^γ, the earliest event v_j^γ from R^γ for which $v_j^\gamma \rightsquigarrow v_i^\beta$ in $G^{\underline{\alpha}}_{\equiv}(t)$. (Even though the information contained in $G^{\underline{\alpha}}_{\equiv}(t)$ can be derived from $G^\alpha(t)$, we need to explicitly maintain the graph $H^{\underline{\alpha}}_{\equiv}(t)$ since $H^\alpha(t)$ does not contain all the information in $G^\alpha(t)$.) Formally, for every pair of events v_i^β and v_j^γ in $G^{\underline{\alpha}}_{\equiv}(t)$ such that (i) $v_j^\gamma \rightsquigarrow v_i^\beta$ in $G^{\underline{\alpha}}_{\equiv}(t)$, (ii) v_i^β is considered open at $R^\alpha(t)$, and (iii) there is no $j' < j$ such that $v_{j'}^\gamma \rightsquigarrow v_i^\beta$ in $G^{\underline{\alpha}}_{\equiv}(t)$, we include in $H^{\underline{\alpha}}_{\equiv}(t)$ the events v_i^β and v_j^γ and a directed edge (v_j^γ, v_i^β). Note that an edge (u, v) in $H^{\underline{\alpha}}_{\equiv}$ merely indicates the existence of a path $u \rightsquigarrow v \in G^{\underline{\alpha}}_{\equiv}$ but not whether its edges are $\longrightarrow$ or $\Longrightarrow$ or a mixture of the two.

3.1 Definition: The *sparse representation* at a replica R^α at time t is a 4-tuple $\mathcal{S}^\alpha(t) = \langle \mathrm{O}^\alpha(t), H^\alpha(t), H^{\underline{\alpha}}_{\equiv}(t), \mathcal{C}^\alpha(t)\rangle$, where $\mathrm{O}^\alpha(t)$ is a data structure containing the set of events from each replica that are considered open at R^α as well as the tables to maintain these open events (defined in the accompanying technical report), $H^\alpha(t)$ is the sparse graph derived from $G^\alpha(t)$, $H^{\underline{\alpha}}_{\equiv}(t)$ is the sparse graph derived from $G^{\underline{\alpha}}_{\equiv}(t)$, and $\mathcal{C}^\alpha(t)$ is a collection of sets, one for each event v considered open at R^α, such that the set corresponding to v contains all events in the equivalence class of v.

Whenever replica R^α communicates to another replica R^β, it sends the tuple $\mathcal{S}^\alpha$. The next theorem bounds the size of this communication.

3.2 Theorem: At any time t, $\mathcal{S}^\alpha(t)$ takes $O(n^4)$ space, where n is the number of replicas.

We observed earlier that the number of open events at any replica can be bounded by $O(n^3)$ and the data structure $\mathrm{O}^\alpha(t)$ used to maintain them takes $O(n^4)$ space. The graph $H^\alpha(t)$ takes $O(n^2)$ space as observed above. The graph $H^\alpha_{\equiv}(t)$ needs $O(1)$ space for each open event for a total of $O(n^3)$ space. Finally, we can show that the equivalence class of each open event can be described compactly using $O(n)$ space. This gives us a bound of $O(n^4)$ space for $\mathcal{C}^\alpha(t)$.

In order to establish that a replica working with the sparse representation will have the same user-visible behavior as if it were working with the complete history graphs, it suffices to show the following.

3.3 Theorem: A class E is maximal in $G^\alpha(t)$ iff E is maximal in $\mathcal{S}^\alpha(t)$.

The proof of this theorem in the accompanying technical report crucially relies on the properties of open and closed events and sparse cones. The main idea of the proof is to establish two key properties. First, for any pair of classes E, E' in $G^\alpha(t)$ such that each contains a latest element, we can determine, using the graph $H^\alpha_{\equiv}(t)$, whether or not they belong to the same component in $G^\alpha_{\equiv}(t)$. Second, if a class E containing a latest element is dominated by another class E' in $G^\alpha(t)$, we show that the graph $H^\alpha(t)$ contains a "witness" to this fact. Since a maximal class always contains a latest element, these two properties together ensure that the set of maximal classes is the same in both $G^\alpha(t)$ and $\mathcal{S}^\alpha(t)$. Finally, we note that, since a latest event is always open, $\mathcal{C}^\alpha(t)$ contains all elements in each maximal class.

4 Related Work

Both theoretical underpinnings and efficient implementation strategies for version vectors [1] and vector clocks [8, 9] have received a great deal of attention in the literature and have been used in many systems (e.g. Coda [10–12], Ficus [13], and Bengal [14]); numerous extensions and refinements have also been studied—see [15] for a recent survey. We conjecture that some of these ideas can be applied to improve the efficiency of our sparse representation. However, we are not aware of any work in this context that explicitly addresses the main concern of our work—an explicit treatment of declarations of agreement (and dominance) between existing events.

A number of systems have used replica equality (e.g., identity of file contents) as an *implicit* indication of agreement. The user-level filesystem synchronization tool Unison [16], for example, considers two replicas of a file to be in agreement whenever their current contents are equal at the point of synchronization. This gives users an easy way to repair conflicts (decide on a reconciled value for the file, manually copy it to both replicas, and re-synchronize), as well as automatically yielding sensible default behavior when Unison is run between previously unsynchronized (but currently equal) filesystems. A similar strategy is used in Panasync [17].

Matrix clocks [18, 19] generalize vector clocks by explicitly representing clock information about other processes's views of the system's execution. We leave for future work the question of whether agreement events such as the ones we are proposing could be generalized along similar lines.

A rather different approach to conflict detection is embodied, for example, in the *hash histories* of Kang et al. [3] and the *version histories* used in the Reconcile file synchronizer [20] and the Clique peer-to-peer filesystem [21]. Rather than deducing causal ordering from reduced representations such as clock vectors, these systems represent the causal history of the system directly—storing and transmitting (hashes of) complete histories of updates. An advantage of such schemes is that their cost is proportional to the number of updates to a file rather than the number of replicas in the system, which may be advantageous in some situations. This suggests that it may be worth considering the possibility of implementing something akin to our naive specification from §2 directly, bypassing the sparse representation.

Reconciliation protocols for optimistically replicated data can be divided into two general categories [22]: *state transfer* and *operation transfer* protocols. We have concentrated on state-based protocols in this work. However, a number of systems (e.g., Bayou [23], IceCube [24], and Ceri's work [25]) reconcile the *operation histories* of replicas rather than their states. It is not clear whether agreement events in the sense we have proposed them could meaningfully be accommodated in this setting.

References

1. Parker, Jr., D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE Trans. Software Eng. (USA) **SE-9**(3) (1983) 240–247
2. Malkhi, D., Terry, D.B.: Concise version vectors in WinFS. In Fraigniaud, P., ed.: Proceedings of the 19th International Conference on Distributed Computing, DISC 2005. Volume 3724 of Lecture Notes in Computer Science., Springer-Verlag (2005) 339–353
3. Kang, B.B., Wilensky, R., Kubiatowicz, J.: The hash history approach for reconciling mutual inconsistency. In: 23rd IEEE International Conference on Distributed Computing Systems (ICDCS'03). (2003)
4. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania (2005) Supersedes MS-CIS-03-42.
5. Pierce, B.C., et al.: Harmony: A synchronization framework for heterogeneous tree-structured data (2006) `http://www.seas.upenn.edu/~harmony/`.
6. Foster, J.N., Greenwald, M.B., Kirkegaard, C., Pierce, B.C., Schmitt, A.: Exploiting schemas in data synchronization. Journal of Computer and System Sciences (2006) To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.
7. Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. In: Proceedings of PODC'87. (1987)

8. Fidge, C.: Logical time in distributed computing systems. Computer **24**(8) (1991) 28–33
9. Mattern, F.: Virtual time and global states of distributed systems. In et. al., M.C., ed.: Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms. Elsevier Science Publishers B. V. (1989) 215–226
10. Kumar, P.: Coping with conflicts in an optimistically replicated file system. In: 1990 Workshop on the Management of Replicated Data, Houston, TX (1990) 60–64
11. Satyanarayanan, M., Kistler, J.J., Kumar, P., Okasaki, M.E., Siegel, E.H., Steere, D.C.: Coda: A highly available file system for a distributed workstation environment. IEEE Transactions on Computers **39**(4) (1990) 447–459
12. Kumar, P., Satyanarayanan, M.: Flexible and safe resolution of file conflicts. In: Proceedings of the annual USENIX 1995 Winter Technical Conference. (1995) 95–106 New Orleans, LA.
13. Guy, R.G., Reiher, P.L., Ratner, D., Gunter, M., Ma, W., Popek, G.J.: Rumor: Mobile data access through optimistic peer-to-peer replication. In: Proceedings of the ER Workshop on Mobile Data Access. (1998) 254–265
14. Ekenstam, T., Matheny, C., Reiher, P.L., Popek, G.J.: The Bengal database replication system. Distributed and Parallel Databases **9**(3) (2001) 187–210
15. Baldoni, R., Raynal, M.: A practical tour of vector clock systems. IEEE Distributed Systems Online **3**(2) (2002) http://dsonline.computer.org/0202/features/ bal.htm.
16. Pierce, B.C., Vouillon, J.: What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania (2004)
17. Almeida, P.S., Baquero, C., Fonte, V.: Panasync: dependency tracking among file copies. In: EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop, ACM Press (2000) 7–12
18. Sarin, S.K., Lynch, N.A.: Discarding obsolete information in a replicated database system. IEEE Transactions onSoftware Engineering **13**(1) (1987) 39–47
19. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Principles of Distributed Computing. (1984) 233–242
20. Howard, J.H.: Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab (1999)
21. Richard, B., Nioclais, D.M., Chalon, D.: Clique: a transparent, peer-to-peer collaborative file sharing system. In: International Conference on Mobile Data Management (MDM), Melbourne, Australia. (2003)
22. Saito, Y., Shapiro, M.: Replication: Optimistic approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto (2002)
23. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado. (1995) 172–183
24. Kermarrec, A.M., Rowstron, A., Shapiro, M., Druschel, P.: The IceCube approach to the reconciliation of diverging replicas. In: ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island. (2001) 210–218
25. Ceri, S., Houtsma, M.A.W., Keller, A.M., Samarati, P.: Independent updates and incremental agreement in replicated databases. Distributed and Parallel Databases **3**(3) (1995) 225–246