Agreeing to Agree: Conflict Resolution for Optimistically Replicated Data

Michael B. Greenwald
Bell Labs, Lucent Technologies

Sanjeev Khanna
University of Pennsylvania

Keshav Kunal
University of Pennsylvania

Benjamin C. Pierce
University of Pennsylvania

Alan Schmitt
INRIA

Technical Report MS-CIS-06-10
Department of Computer and Information Science
University of Pennsylvania

November 1, 2006

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 in order to make progress, depending instead on a notion of *causal history* of events. If a replica in such a system 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. [23, 21]), 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 [23, 16], 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 or at multiple sites.

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* [23]. 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 [23].

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

	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)
$R^a \to R^c$ and $R^b \to R^c$	1	x (1,0,0)	1	x (0,1,0)	2	x (1,1,2)
$R^a \to R^b$	1	x (1,0,0)	2	x (1,2,0)	2	x (1,1,2)

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

R^c all independently set their value to x at (local) time 1. Although 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 (all with value x) 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,

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

Figure 2: A case in which vector clocks "forget" a resolution event.

all would be well. 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. The clock on R^a will be (2,1,0), while the clock on R^c will be (1,1,1).)

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 the particular representation of causal history in terms of vector clocks, but a fundamental limitation of the conventional notion of causal history itself: the system stalls because causal histories do not recognize 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 what we want is for the system to prefer one *causally unrelated* value to another.

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 [11, 25], 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 [12]. 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 Section 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 Section 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 (containing its current state) 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. Section 4 shows that the restriction to reciprocal communication is necessary: with completely unrestricted asymmetric communication, no sparse representation that operates in bounded space can implement the specification described in Section 2 correctly. Section 5 discusses related work.

2 An Agreeable Reconciliation Framework

A reconciliation framework has three choices of action 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, or that an event may be better-than a causally unrelated event.

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 (e.g. [3]) should apply.) The variables α, β, etc. range over indices of replicas. For simplicity, we focus on the case where each replica holds just 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, etc. (and v^α_+, v^α_-, etc., for earlier and later events at the same replica).

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 $G^\alpha_\equiv$, 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 $G^\alpha_\equiv$ 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 $y \in cone(x)$. 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^a_i and v^b_j, 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^b_j (by creating an event v^a_{i+1} with the same value as v^b_j and declaring v^a_{i+1} to be in an equivalence class E with v^b_j). R^b tries to reconcile the conflict by similarly adopting the value of v^a_i, by putting v^b_{j+1} in an equivalence class E' with it. E takes E' into account, because v^a_i is in the cone of v^a_{i+1}; similarly, E' takes E into account because v^b_j is in the cone of v^b_{j+1}. 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 $y \in cone(x)$. 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 considering a component, there are two kinds of latest events in it: those which are taken into account by events in another component, and those which are not. The former are clearly not candidates for solving a conflict, the latter are the maximal events defined above. These maximal events may however be superseded by other events in the component but, as components consist of cycles of classes, the converse is also true. Hence these intra-component relationships do not matter when defining maximal events.

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 Section 1. Figure 3 shows the evolution of the history graphs in the example from Figure 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

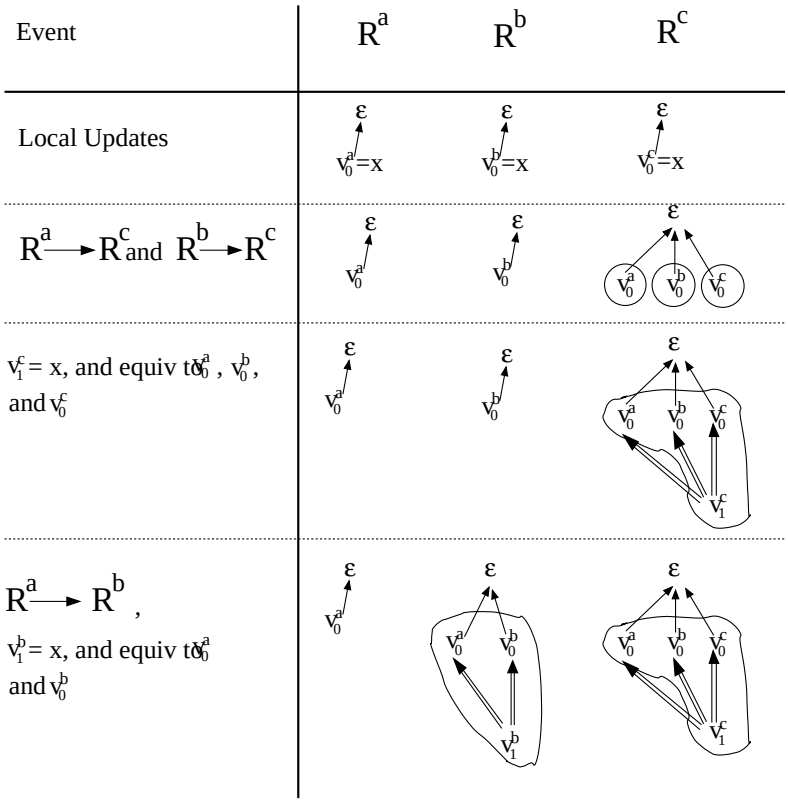

Figure 3: Example from Figure 1, using history graphs. Single line arrows represent "took into account through dominance", and double-lined arrows represent "took into account through agreement". The circled events are in the same class. It is easy to see that no conflicts or cycles will be caused by the union of any subset of these graphs.

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 [6].) However, to ensure that our implementation in Section 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 message of its own to R^β. (Indeed, in Appendix **??** 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 $\mathit{Current}^\alpha = v_{\mathit{init}}$ 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 equivalence 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. If $v_{i-1}^\alpha \notin W$, it is added to the set. 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. If $v_{i-1}^\alpha \notin W$, an edge $v_i^\alpha \longrightarrow v_{i-1}^\alpha$ is added to the graph G^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 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. To this end, 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 Section 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.

An important lesson from the intractability result mentioned in the previous section is that 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. Reciprocal communication enables us to track such "critical" events with bounded space.

An event v is *globally 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 globally closed is *globally open*. If v is globally closed, then any replica that hears about v will simultaneously hear about a successor of v. It follows from this that a globally closed event can never be a latest event at any replica (hence also not a maximal one), and the following fact holds:

3.1 Fact: Once an event is globally closed, it stays globally closed forever. No edges can be created to or from a globally closed event at any replica at any time in the future.

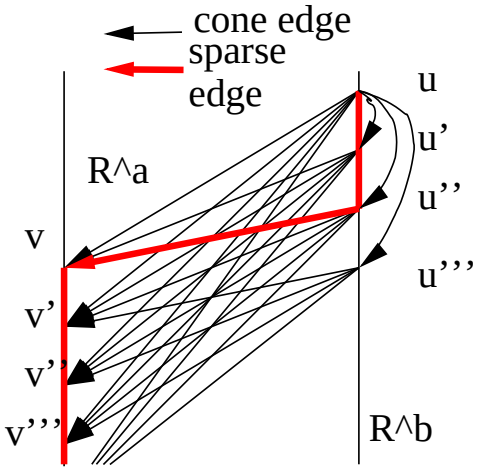

Figure 4: The black edges shown are the edges from the events $u, u', \ldots$ on R^b to the events $v, v', \ldots$ on R^a. The red, thick, edge is the only edge from u to v in *sparse-cone*(u). It's the edge from the earliest predecessor of u that has an edge to the latest event on R^a that has any edges from R^b.

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

We maintain a data structure O^α (described in Section 3.1) 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 *considered closed at* R^α. An event that is not considered closed at a given replica is *considered open at* that replica.

When there is no room for ambiguity we will use the phrase "closed" to refer to globally closed events.

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) as shown in the next Lemma.

3.2 Lemma: For any element w, it can be determined whether or not $w \in$ *cone*(v) using *sparse-cone*(v).

Proof: There are two cases — either $w \in$ *cone*(v) or $w \notin$ *cone*(v). If $w \in$ *cone*(v), then consider the latest successor of w, say w_+, such that $w_+ \in$ *cone*(v). By definition, $w_+ \in$ *sparse-cone*(v), and thus we can conclude that $w \in$ *cone*(v) from the evidence in *sparse-cone*(v). On the other hand, if $w \notin$ *cone*(v), then for any successor of w, say w_+, it also holds that $w_+ \notin$ *cone*(v). Since *sparse-cone*(v) is a subgraph of *cone*(v), neither w nor any successor of w belong to *sparse-cone*(v) and hence we can infer $w \notin$ *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 Section 3.1 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. Finally, in Section 3.2, we prove that the representation is correct in the sense that it will report the same maximal events (and equivalence classes) as the specification in Section 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 $H^\alpha_\equiv(t)$. 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 $H^\alpha_\equiv(t)$, summarizes the information contained in $G^\alpha_\equiv(t)$ as follows. Let $v \rightsquigarrow w$ denote the existence of a path from an event v to event w in a graph $G^\alpha_\equiv$. For each event v_i^β considered open at $R^\alpha(t)$, $H^\alpha_\equiv(t)$ 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 $G^\alpha_\equiv(t)$. (Even though the information contained in $G^\alpha_\equiv(t)$ can be derived from $G^\alpha(t)$, we need to explicitly maintain the graph $H^\alpha_\equiv(t)$ 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 $G^\alpha_\equiv(t)$ such that (i) $v_j^\gamma \rightsquigarrow v_i^\beta$ in $G^\alpha_\equiv(t)$, (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 $G^\alpha_\equiv(t)$, we include in $H^\alpha_\equiv(t)$ 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 $H^\alpha_\equiv$ merely indicates the existence of a path $u \rightsquigarrow v \in G^\alpha_\equiv$ but not whether its edges are $\longrightarrow$ or $\Longrightarrow$ or a mixture of the two.

3.3 Definition: [Sparse Representation] The *sparse representation* at a replica R^α at time t is a 4-tuple $\mathcal{S}^\alpha(t) = \langle \mathrm{O}^\alpha(t), H^\alpha(t), H^\alpha_\equiv(t), \mathcal{C}^\alpha(t) \rangle$, 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 Section 3.1),
- $H^\alpha(t)$ is the sparse graph derived from $G^\alpha(t)$,
- $H^\alpha_\equiv(t)$ is the sparse graph derived from $G^\alpha_\equiv(t)$, 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 $\mathcal{S}^\alpha$.

3.1 Incremental Maintenance of the Sparse Representation

The incremental maintenance of the sparse graphs in $S^\alpha(t)$ depends critically on the component data structure O^α that is used to mark events as open or closed. We now describe O^α in detail and present an algorithm for maintaining it incrementally.

For a replica R^α, we use the term *local* to qualify events created at that replica, that is, an event v_i^α. We use *non-local* to qualify events which were created at other replicas, that is, an event v_j^β where $\beta \neq \alpha$. We use n to denote the number of replicas. Let m_l denote an upper bound on the number of local events that are considered open at a replica at any given moment. We can group the non-local events considered open at a replica based on the replica they were created at. Let m_{nl} denote an upper bound on the number of events created at any replica R^β that are considered open

at a replica R^α. Hence the total number of events considered open at a replica can be bounded by $m_l + (n-1)m_{nl}$. Claim 3.1.4 will show that these two upper bounds, m_l and m_{nl} are precisely equal. Henceforth, we will use m for the common upper bound for ease of notation. Claim 3.1.3 will prove that this common bound is in fact $O(n^2)$, from which it follows that the total number of events considered open by any replica at any given moment is $O(n^3)$.

Let $l^\alpha_\beta(t)$ represent the index of the latest event from R^β in the graph $G^\alpha(t)$. We elide t when it is clear from context. Let t^α_β denote, for any R^β that communicated with R^α, the largest value of $t' \leq t$ (in other words, the latest time before t) such that R^α received $G^\beta(t')$. In the treatment below, although the implementation maintains, sends, and receives S^α, we refer to latest events in the graph G^α instead of in the sparse representation S^α. Similarly, we refer to the transmission and reception of G^α. Our terminology has been mostly defined in G^α, and not in S^α. The two formulations are equivalent when we discuss latest events and their classes, because at all times latest events are always considered open and hence belong to S^α.

At each replica R^α and for all times t, we maintain the following structures (collectively called O^α):

- $\mathit{NonLocalSent}^\alpha$, a table of $n.m$ entries storing sets of replicas. For each $\beta \neq \alpha$ and for each event v^β_i considered open at R^α, the entry[1] $\mathit{NonLocalSent}^\alpha[\beta, i](t)$ is the set of replicas that heard from R^α while v^β_i was the latest event from R^β in the graph G^α. (Slightly more formally, $\mathit{NonLocalSent}^\alpha[\beta, i](t)$ is the set of all R^γ, s.t. $\exists t', l^\alpha_\beta(t') = i$ and R^α sent the graph $G^\alpha(t')$ to R^γ.)

- $\mathit{LocalKnownToBeSent}^\alpha$, a table of m entries storing sets of replicas. For each local event v^α_i considered open at R^α, the entry $\mathit{LocalKnownToBeSent}^\alpha[i](t)$ is a set of all replicas that have received v^α_i as a latest value, that R^α is aware of. ($\mathit{LocalKnownToBeSent}^\alpha[i](t)$ is the set of all R^β such that there exist a t' and γ s.t. $t' < t^\alpha_\gamma$, and $l^\gamma_\alpha(t') = v^\alpha_i$, and R^γ sent $G(t')$ to R^β.)

- $\mathit{LastHeardBack}^\alpha$, an $n \times n$ table indexed by pairs of replicas and storing event indices. At time t, the entry $\mathit{LastHeardBack}^\alpha[\beta, \gamma](t)$ records R^α's knowledge of the index of the latest event from R^γ in the graph $G^\beta(t')$, for some $t' \leq t$. In other words, $\mathit{LastHeardBack}^\alpha[\beta, \gamma](t) = l^\beta_\gamma(t^\alpha_\beta)$.

- Open^α, a set containing all the events (from all replicas) that are considered open at R^α.

R^α updates the components of O^α as follows.

Before R^α sends $\mathcal{S}^\alpha$ to another replica R^β, it sets $\mathit{LastHeardBack}^\alpha[\alpha, \alpha] = l^\alpha_\alpha$ and adds $v^\alpha_{l^\alpha_\alpha}$ to the set Open^α. After sending $\mathcal{S}^\alpha$, it adds R^β to the set $\mathit{LocalKnownToBeSent}^\alpha[l^\alpha_\alpha]$ and, for all γ, adds R^β to $\mathit{NonLocalSent}^\alpha[\gamma, l^\alpha_\gamma]$.

Whenever R^α receives a transmission of $\mathcal{S}^\beta$ from another replica R^β it performs the following steps in order:

1. For each local event v^α_i considered open at R^α, it sets

$$\mathit{LocalKnownToBeSent}^\alpha[i] = \mathit{LocalKnownToBeSent}^\alpha[i] \cup \mathit{NonLocalSent}^\beta[\alpha, i].$$

[1] In both *NonLocalSent* as well as *LocalKnownToBeSent*, i should be thought of as a key for lookup rather than an index into the table.

2. For every pair of replicas (γ, δ), R^α sets

$$\mathit{LastHeardBack}^\alpha[\gamma, \delta] = \max\{\mathit{LastHeardBack}^\alpha[\gamma, \delta], \mathit{LastHeardBack}^\beta[\gamma, \delta]\}.$$

3. For all $v_i^\gamma \in (\mathit{Open}^\beta - \mathit{Open}^\alpha)$, $\gamma \neq \alpha$, if $l_\gamma^\alpha < i$, it sets $\mathit{Open}^\alpha = \mathit{Open}^\alpha \cup v_i^\gamma$.

4. For all $v_i^\gamma \in (\mathit{Open}^\alpha - \mathit{Open}^\beta)$ with $\gamma \neq \alpha$, if $l_\gamma^\beta \geq i$, it sets $\mathit{Open}^\alpha = \mathit{Open}^\alpha - v_i^\gamma$.

5. For all $v_i^\alpha \in \mathit{Open}^\alpha$, if $\mathit{LastHeardBack}^\alpha(\beta, \alpha) \geq i + 1$ for every replica R^β such that $R^\beta \in \mathit{LocalKnownToBeSent}^\alpha[i]$, it sets $\mathit{Open}^\alpha = \mathit{Open}^\alpha - v_i^\alpha$.

The third step ensures that any new events that get added to G^α and are considered open at R^β, get added to set of events considered open at R^α. Earlier we defined an event to be "considered closed at a replica" when that replica marked it closed. This is accomplished at a replica R^α by removing the event from the set Open^α. For non-local events, R^α can mark it closed only after it knows that the creator replica marked it closed. This is captured in Step 4 above where the *if* condition checks whether $v_i^\gamma \in G^\beta(t')$ for some $t' < t$ (accomplished by checking $l_\gamma^\beta \geq i$). If the event is considered closed at R^β (accomplished by checking if $v_i^\gamma \in \mathit{Open}^\beta$), it must have been considered closed through a chain of communication originating from the event's creator replica, R^γ in this case, and so it is safe for R^α to consider it closed too.

A creator replica marks an event closed when it has a certificate for it to be globally closed. The last step determines when a creator replica can mark its event closed. Note that once $\mathit{LastHeardBack}^\alpha(\beta, \alpha) \geq i + 1$, R^α knows that v_i^α is no longer a latest event at R^β and also any edges created to it have been communicated to R^α. Therefore, the following claim is sufficient to show that when a creator replica marks an event closed, it is globally closed and hence $\mathrm{O}^\alpha(t)$ can be used to determine events which are considered closed correctly.

3.1.1 Claim: If v_i^α is ever a *latest event* in G^β then $R^\beta \in \mathit{LocalKnownToBeSent}^\alpha[i]$ before v_i^α is marked closed by R^α.

Proof: Assume for a contradiction that for some R^β, $R^\beta \notin \mathit{LocalKnownToBeSent}^\alpha[i]$ before v_i^α is marked closed at its creator R^α. At least one such R^β must have received a $G^\gamma(t)$ containing v_i^α from a replica $R^\gamma \neq R^\alpha$ in which the Claim holds. (v_i^α must originally be communicated from R^α. The claim must hold in any replica R^γ that directly received a G^α containing v_i^α from R^α, because R^α would have added R^γ to $\mathit{LocalKnownToBeSent}^\alpha[i]$.) By assumption v_i^α is a latest event in G^β, and therefore v_i^α was the latest event from R^α(in G^γ) at the time of transmission, and therefore $R^\beta \in \mathit{NonLocalSent}^\gamma[\alpha, i]$ by our update rules.

But when v_i^α is closed, $\mathit{LastHeardBack}^\alpha[\gamma, \alpha]$ must be at least $i + 1$, implying that R^γ communicated (perhaps transitively) with R^α after transmitting v_i^α to R^β. So R^α must have updated $\mathit{LocalKnownToBeSent}^\alpha[i]$ when it (transitively) received a communication from R^γ and added R^β to the list $\mathit{LocalKnownToBeSent}^\alpha[i]$. This must happen before v_i^α is marked closed and is hence a contradiction. □

The next lemma follows from the correctness of O^α and the definition of closed events. It is used only for analysis while arguing the correctness of $\mathcal{S}^\alpha$.

3.1.2 Lemma: If an event v is considered closed at R^α at time t, all edges added to/from it at any replica belong to $G_\equiv^\alpha(t)$ and no more edges can be added to/from it at a replica at a time $> t$. Also, if there exists an edge $(v, w) \in G_\equiv^\beta(t)$ such that $(v, w) \notin G_\equiv^\alpha(t)$, v is considered open at R^α at time t.

Next we bound the number of local events which are considered open at any replica.

3.1.3 Claim: For each replica R^α, the number of local events considered open at R^α at any time t is $O(n^2)$.

Proof: Let v_i^α be a local event considered open at replica R^α at time t. We call a pair of replicas (R^β, R^γ) a *witness pair* for v_i^α if the pair satisfies the following: (i) v_i^α was not an event at R^γ until it received some $G^\beta(t')$ from R^β, (ii) v_i^α was a latest event in $G^\gamma(t')$ for some $t' < t$ (iii) $\mathit{LastHeardBack}^\alpha[\beta,\alpha](t) \geq i+1$, and (iv) $\mathit{LastHeardBack}^\alpha[\gamma,\alpha](t) < i+1$.

Note that for every local event v_i^α considered open at time t, except the latest such event, $\mathit{LastHeardBack}^\alpha[\alpha,\alpha](t) \geq i+1$. Also, there exists at least one replica $R^\gamma \in \mathit{LocalKnownToBeSent}[i](t)$ such that $\mathit{LastHeardBack}^\alpha[\gamma,\alpha](t) < i+1$. So there exists at least one witness pair (R^β, R^γ) (if we trace the path from R^α to R^γ through which the latter heard of v_i^α, R^β is the replica that immediately precedes R^γ). If there is more than one witness pair, we can choose one arbitrarily.

We now claim that a pair (R^β, R^γ) can be chosen as a witness pair for at most 2 open events from R^α. Since there are at most $n(n-1)$ distinct replica pairs, the claim follows.

Suppose not and let $v_{i_1}^\alpha, v_{i_2}^\alpha, v_{i_3}^\alpha$ with $i_1 < i_2 < i_3$ denote the open events for which (R^β, R^γ) are witness pairs with $t_1 < t_2 < t_3$ denoting the times when the events were communicated to R^γ by R^β respectively.

Note that at time t, $\mathit{LastHeardBack}^\alpha[\beta,\alpha](t) \geq i_3+1$ and $\mathit{LastHeardBack}^\alpha[\gamma,\alpha](t) < i_1+1$ by our definition of a witness pair. Our rules for reciprocal communication ensure that between time instances t_2 and t_3, R^β must have received a direct communication from R^γ at which point the latest event from R^α in G^γ was $v_{i_2}^\alpha$, or some successor of $v_{i_2}^\alpha$. Hence when R^β receives a communication from R^γ, it sets $\mathit{LastHeardBack}^\beta[\gamma,\alpha] \geq i_2$. Now since $\mathit{LastHeardBack}^\alpha[\beta,\alpha] \geq i_3+1$, it communicated (transitively) with R^α after t_3 and hence $\mathit{LastHeardBack}^\alpha[\gamma,\alpha](t) \geq \mathit{LastHeardBack}^\beta[\gamma,\alpha](t_3) \geq i_2$ which is a contradiction because $\mathit{LastHeardBack}^\alpha[\gamma,\alpha](t) < i_1+1 \leq i_2$ for (R^β, R^γ) to be a witness pair. □

3.1.4 Claim: For each replica R^γ, for each replica R^α, the number of events created at R^α and considered open at R^γ at any time t is $O(n^2)$.

Proof: $l_\alpha^\gamma(t)$ is not just the index of the latest event from R^α *currently* in G^γ, but the index of the latest event that R^γ *ever* received up to time t. If R^γ never received a v_j^α, with $j > i$, then v_i^α must still be considered open at R^γ because it has no known successor on R^γ. It follows that any event created on R^α and considered open on R^γ was already created on R^α at the time $v_{l_\alpha^\gamma(t)}^\alpha$ was created.

It is also easy to see that if R^β sends $S^\beta(t)$ to R^γ, then any event considered closed on either R^γ or R^β will be considered closed on R^γ after processing $S^\beta(t)$. Because $v_{l_\alpha^\gamma}^\alpha$ was sent originally from R^α, it follows that every event that was considered closed on R^α at the time $v_{l_\alpha^\gamma}^\alpha$ was created is also closed on R^γ.

Consequently, the set of events created on R^α and currently considered open at R^γ is a subset of the local events considered open at R^α at the time $v_{l_\alpha^\gamma}^\alpha$ was created. By Claim 3.1.3 the number of open local events on R^α at any time was at most $O(n^2)$ and our claim follows. □

These claims provide an upper bound on the number of open events at a replica. We next show that this bound is tight.

3.1.5 Lemma: The number of local events considered open at a replica can be $\Omega(n^2)$. The total number of events considered open at a replica can be $\Omega(n^3)$.

Proof: Consider $n/2+1$ replicas, with a special replica R^a and $n/2$ replicas named $R^{b_1} \ldots R^{b_{n/2}}$. We will first show that there can be $\Omega(n^2)$ events considered open at R^a. For events which are created and considered open at R^a, we will "blame" i events on a replica R^{b_i}, there by getting the desired bound.

In the first round, R^a creates an event and sends v_1^a to $R^{b_{n/2}}$, does an update, sends v_2^a to $R^{b_{n/2-1}}$, and so on till it sends $v_{n/2}^a$ to R^{b_1}. Next, the replicas $R^{b_1} \ldots R^{b_{n/2-1}}$ send their graphs to $R^{b_{n/2}}$ and R^a. $R^{b_{n/2}}$ does not communicate with anyone from this time instance (neither sending its own graph, nor receiving any other graph) and hence these $n/2$ events will remain open.

Likewise, in the i^{th} round, replicas R^a and $R^{b_1} \ldots R^{b_{i-1}}$ conspire together to send $n/2+1-i$ new events created at R^a to $R^{b_{n/2+1-i}}$, thus "blaming" these open events on $R^{b_{n/2+1-i}}$. Replica $R^{b_{n/2+1-i}}$ stays disconnected from everyone from this instance.

So, after $n/2$ rounds, there are $\sum_{i=1}^{n/2} = \Omega(n^2)$ events considered open at R^a.

If we consider a set of $n/2$ replicas $R^{a_1} \ldots R^{a_{n/2}}$ such that they simultaneously create new events and send their graphs to replicas in the other set, we will have $\Omega(n^2)$ events considered open at each replica R^{a_i}. We have to be slightly careful in setting up the communication pattern to ensure this given our reciprocality constraints. A replica R^{b_j} waits till it receives graphs from all replicas $R^{a_1} \ldots R^{a_{n/2}}$ before sending its graph to a replica $R^{b'_j}$.

Now if all replicas $R^{a_2} \ldots R^{a_{n/2}}$ send their graphs to R^{a_1}, the total number of events (local as well as non-local) considered open at R^{a_1} is $\Omega(n^3)$. □

It is easy to see that all events considered open in the proof are also globally open. This leads to the following corolllary:

3.1.6 Corollary: The number of local events at a replica which are globally open can be $\Omega(n^2)$. The number of events which belongs to a replica's graph and are globally open can be $\Omega(n^3)$.

By inspection of the structures in O^α, we see that the size of the tables in O^α is $O(mn^2) + O(mn) + O(n^2) + O(mn)$. By Claims 3.1.3 and 3.1.4, we know that the upper bound m is $O(n^2)$, which leads us to the following claim:

3.1.7 Claim: The size of tables at each replica to maintain the list of events considered open at that replica is $O(n^4)$.

Having established a bound on the number of events considered open at each replica as well as the data structure to maintain them, we can prove a bound on the size of sparse representation S^α at a replica R^α.

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

Proof: We will separately bound the space needed by each component in the representation. We know from Claim 3.1.7 that the size of the tables in O^α is is $O(n^4)$. The size of the sparse cone of any element is $O(n)$ as it contains $O(1)$ entries per replica. Since at any time t, R^α has at most one latest element for each replica R^β, the size of $H^\alpha(t)$ is $O(n^2)$. The size of $H^\alpha_{\equiv}$ is $O(n^4)$ as it may need $O(n)$ events per open event, and it uses $O(1)$ space per event.

Finally, consider the (equivalence) class of any open event v. For any replica R^β, we have the property that if v_i^β and v_j^β belong to this class then so does v_k^β for any $i > k > j$. Thus all events from R^β that are in the class of v can be compactly described by simply storing the earliest and the latest events from R^β known to be equivalent to v. Hence each equivalence class takes $O(n)$ space and the total space taken by $\mathcal{C}^\alpha(t)$ is $O(n^4)$. Adding up the sizes of all three components, the theorem follows.

□

3.2 Correctness of the Sparse Representation

Our next goal is to show that if $S^\beta(t')$ is maintained correctly at all replicas R^β for all times $t' < t$, then at any replica R^α we can derive $S^\alpha(t)$.

Claim 3.1.1 already shows that $O^\alpha(t)$ can be maintained correctly . We will rely on this claim to show it to be true for $H^\alpha(t)$, $H^\alpha_\equiv(t)$ and $\mathcal{C}^\alpha(t)$ as well.

3.2.1 Lemma: If for all β, we have correctly computed $H^\beta(t')$ at each replica R^β and at each time $t' < t$, then we can correctly compute $H^\alpha(t)$ at any replica $R^\alpha(t)$.

Proof: At a given time t, $H^\alpha(t)$ is (re)computed only when either R^α receives $\mathcal{S}^\beta(t-1)$ from some replica R^β, or when R^α injects a new local event.

If a user sent $\mathcal{S}^\beta(t-1)$ to R^α from some replica R^β at time t, then $H^\alpha(t)$ can be computed by taking the union of the sparse cones of latest elements from the union of $H^\alpha(t-1)$ and $H^\beta(t-1)$.

Alternatively, suppose the user action at time t is to inject a new local event, say v_l^α, with edges to a subset W of maximal elements as well as to element v_{l-1}^α. Each maximal element is necessarily a latest element, therefore $H^\alpha(t-1)$ contains sparse cones of all elements in $W \cup \{v_{l-1}^\alpha\}$. It is easy to verify that *sparse-cone*(v_l^α) can be derived from the sparse cones of elements in $W \cup \{v_{l-1}^\alpha\}$. □

We did not yet give a detailed description of how R^α maintains the graph $H^\alpha_\equiv$. It proceeds in two stages. The first stage, which we denote by t_u, is the update stage when R^α updates its graph $H^\alpha_\equiv$. The second stage, which we denote by t_s, is the sparsifying stage when R^α deletes unneeded edges and vertices from the graph.

During stage t_u, R^α either receives a graph $H^\beta_\equiv$ or updates the local event by adding new edges. If the former, R^α sets $H^\alpha_\equiv(t_u) = (H^\alpha_\equiv(t-1) \cup H^\beta_\equiv(t-1))$. If the latter, (updating the local event), the new event along with its (new) edges are added to $H^\alpha_\equiv(t-1)$. Edges are only added to latest events, and latest events are open and hence are contained in $H^\alpha_\equiv(t-1)$. In either case, for every pair of events w and w_- in the graph (where w_- is a predecessor of w) we add the edge (w, w_-), if it does not already exist. We then recompute the transitive closure of the entire graph. We refer to this graph as $H^\alpha_\equiv(t_u)$.

In the sparsification stage, based on the set of events considered open, the graph $H^\alpha_\equiv(t_u)$ is suitably sparsified to retain the relevant edges and vertices to obtain the graph $H^\alpha_\equiv(t)$

For a graph $H^\beta_\equiv(t')$, a pair of events (v, w) is called a valid *query pair* if w is considered open at $R^\beta(t')$ and $v \in G^\beta_\equiv(t')$.

3.2.2 Claim: For every valid query pair (v, w) in $H^\alpha_\equiv(t)$, $H^\alpha_\equiv(t)$ can be used to determine whether a path from v to w is in $G^\alpha_\equiv(t)$.

Proof: Let v_- be the earliest predecessor of v which has a path to w in $G^{\alpha}_{\equiv}(t)$. By assumption w is an event that is considered open at R^{α}. It follows from our definition of $H^{\alpha}_{\equiv}(t)$ that the edge (v_-, w) belongs to $H^{\alpha}_{\equiv}(t)$. We can determine if v has a path to w, because the path (v, v_-) always exists. □

We can further show that $H^{\alpha}_{\equiv}(t_u)$ (the augmented $H^{\alpha}_{\equiv}(t-1)$ before sparsification eventually yields $H^{\alpha}_{\equiv}(t)$) has as much information.

3.2.3 Claim: For every query pair (v, w) that is valid for $H^{\alpha}_{\equiv}(t)$, $H^{\alpha}_{\equiv}(t_u)$ can be used to determine correctly whether a path from v to w exists in $G^{\alpha}_{\equiv}(t)$ if and only if $H^{\alpha}_{\equiv}(t)$ can be used to do so.

The claim follows from the fact that $H^{\alpha}_{\equiv}(t) \subseteq H^{\alpha}_{\equiv}(t_u)$ and that, by Claim 3.1.1, $O^{\alpha}(t)$ maintains the list of open events correctly.

3.2.4 Lemma: If for all β, we have correctly computed $H^{\beta}_{\equiv}(t')$ at each replica R^{β} and at each time $t' < t$, then we can correctly compute $H^{\alpha}_{\equiv}(t)$ at any replica $R^{\alpha}(t)$.

Proof: By the definition of $H^{\alpha}_{\equiv}(t)$, for each event v considered open at R^{α}, $H^{\alpha}_{\equiv}(t)$ must contain the earliest event w^{γ} from R^{γ} such that $w^{\gamma} \rightsquigarrow v \in G^{\alpha}_{\equiv}(t)$. In other words, for each v considered open at R^{α}, and for each R^{γ}, $H^{\alpha}_{\equiv}(t)$ contains the query pair (w^{γ}, v) with the earliest w^{γ}.

Claim 3.1.1 shows that we correctly maintain the set of events considered open, so sparsification preserves correctness. Therefore the earliest incorrectly maintained $H^{\alpha}_{\equiv}(t)$ must occur during update, for some $H^{\alpha}_{\equiv}(t_u)$.

Suppose the claim is not true at time t at replica R^{α}. This implies there exists at least one event z that has a path to an event y considered open in $G^{\alpha}_{\equiv}(t)$ and no predecessor of z has a path to y in $G^{\alpha}_{\equiv}(t)$ but the edge $(z, y) \notin H^{\alpha}_{\equiv}(t)$.

We fix (z, y) by considering all query pairs (p, q) that are valid for $H^{\alpha}_{\equiv}(t)$ such that $H^{\alpha}_{\equiv}(t)$ cannot determine that a path from p to q exists, even though a path $p \rightsquigarrow q \in G^{\alpha}_{\equiv}(t)$. Among all such pairs, let (z, y) denote the pair with the shortest path in the graph $G^{\alpha}_{\equiv}(t)$. (Pick one arbitrarily if more than one pair have the same shortest path).

Based on Claim 3.2.3, we will show that there exists a different pair (p, q) valid for $H^{\alpha}_{\equiv}(t)$ such that $p \rightsquigarrow q$ is shorter than $z \rightsquigarrow y$ for which $H^{\alpha}_{\equiv}(t_u)$ fails to answer correctly if $H^{\alpha}_{\equiv}(t_u)$ fails to answer correctly for (z, y), thereby contradicting our assumption that (z, y) has the shortest path.

We assume from the inductive hypothesis that for all β and time $t' < t$, $H^{\beta}_{\equiv}(t')$ can be used to correctly determine the existence of a path from p to q in $G^{\beta}_{\equiv}(t')$ for a query pair valid for $H^{\beta}_{\equiv}(t')$.

Fix a shortest path $z \rightsquigarrow y \in G^{\alpha}_{\equiv}(t)$. Let $z \rightsquigarrow x$ be the largest subpath of $z \rightsquigarrow y$ in $G^{\alpha}_{\equiv}(t-1)$. (More formally, let x be an event on $z \rightsquigarrow y \in G^{\alpha}_{\equiv}(t)$, such that if $z \rightsquigarrow x' \in G^{\alpha}_{\equiv}(t-1)$, then x' is an event on $z \rightsquigarrow x$.) Consider the following cases:

1. $x = y$. Clearly y is also considered open at $R^{\alpha}(t-1)$ implying that (z, y) is a valid query pair for $H^{\alpha}_{\equiv}(t-1)$. Hence $H^{\alpha}_{\equiv}(t-1)$ and thus $H^{\alpha}_{\equiv}(t_u) \supseteq H^{\alpha}_{\equiv}(t-1)$ answers "yes" for the query pair, which contradicts our assumption that (z, y) is *not* in $H^{\alpha}_{\equiv}(t)$.

2. $y \neq x \neq z$. If x is open at $R^{\alpha}(t-1)$ then (z, x) is a valid query pair for $H^{\alpha}_{\equiv}(t-1)$. Hence $H^{\alpha}_{\equiv}(t-1)$ and thus $H^{\alpha}_{\equiv}(t_u) \supseteq H^{\alpha}_{\equiv}(t-1)$ answer "yes" for this query pair. Since (x, y) is a valid query pair for $H^{\alpha}_{\equiv}(t)$, we have a shorter contradiction for the pair else $H^{\alpha}_{\equiv}(t_u)$ would answer "yes" for (z, y).

Otherwise, if x is considered closed at $R^\alpha(t-1)$, then we know from Lemma 3.1.2 that no more edges can be added to or from x, contradicting the existence of the path $z \rightsquigarrow y$ at time t.

3. $x = z$ or $z \notin G^\alpha_{\equiv}(t-1)$. There are two cases to consider, depending on the user action at time t.

   First, suppose the action at time t at replica R^α was updating the local event. Let w be the next event on the path. The only possibility is that the edge (z, w) was created at time t and hence $(z, w) \in H^\alpha_{\equiv}(t_u)$. Since (w, y) is a valid query pair for $H^\alpha_{\equiv}(t)$, we have a shorter contradiction for (w, y).

   Second, suppose the action at time t was receiving a graph from replica R^β. If the entire path $z \rightsquigarrow y \in G^\beta_{\equiv}(t-1)$, (z, y) is a valid query pair for $H^\beta_{\equiv}(t-1)$ as y has to be considered open at $R^\beta(t-1)$ or else y would be considered closed at $R^\alpha(t)$. Hence $H^\beta_{\equiv}(t-1)$ and thus $H^\alpha_{\equiv}(t_u) \supseteq H^\beta_{\equiv}(t-1)$ answer "yes" for the query pair which is a contradiction.

   Otherwise, let (w, v) be the edge closest to z on the path $z \rightsquigarrow y$ such that $(w, v) \notin G^\beta_{\equiv}(t-1)$. Note that there is at least one such edge by our assumption and $w \neq z$ because the first edge on the path has to belong to $G^\beta_{\equiv}(t-1)$ since it does not belong to $G^\alpha_{\equiv}(t-1)$. Now w should be considered open at $R^\beta(t-1)$ or else using Lemma 3.1.2, the edge $(w, v) \in G^\beta_{\equiv}(t-1)$. Hence (z, w) is a valid query pair for $H^\beta_{\equiv}(t-1)$ implying both $H^\beta_{\equiv}(t-1)$ and $H^\alpha_{\equiv}(t_u)$ answer "yes" for it. Since (w, y) is a valid query pair for $H^\alpha_{\equiv}(t)$, we have a shorter contradiction. □

3.2.5 Lemma: If for all β, we have correctly computed $\mathcal{C}^\beta(t')$ at each replica R^β and at each time $t' < t$, then we can correctly compute $\mathcal{C}^\alpha(t)$ at any replica $R^\alpha(t)$.

Proof: Suppose the user action at replica R^α at time t is injecting a new local event, say v^α_l. If there are no equivalence edges incident from v^α_l, then $\mathcal{C}^\alpha(t)$ is $\mathcal{C}^\alpha(t-1)$ along with a new equivalence class containing the event v^α_l. Otherwise, since equivalence edges can be created only to other latest events (which are always considered open), we know their equivalence classes by the inductive assumption. We merge classes of each of these events into a single new class along with the event v^α_l. The other classes in $\mathcal{C}^\alpha(t-1)$ remain unchanged.

Suppose the user action at replica R^α is receiving $S^\beta(t-1)$ from another replica R^β. The only interesting case is when there exists E in $G^\alpha(t-1)$ and $E' \in G^\beta(t-1)$ such that $E \cap E' \neq \emptyset$. In this case, the graph $G^\alpha(t)$ contains a new merged class $E \cup E'$. If each of E and E' contains an event considered open at $R^\alpha(t-1)$ and $R^\beta(t-1)$ respectively, then by inductive assumption, we know all elements in E and E' in $\mathcal{C}^\alpha(t-1)$ and $\mathcal{C}^\beta(t-1)$, respectively. Hence we can compute $E \cup E'$ in $C^\alpha(t)$. If one of them, say E, consists only of events considered closed at $R^\alpha(t-1)$, then by Lemma 3.1.2 any element with an equivalence path to an element in E must already be in E. So $E' \subseteq E$ and both classes will be discarded from $\mathcal{C}^\alpha(t)$. Thus we can correctly determine $\mathcal{C}^\alpha(t)$. □

3.3 Maximal Classes can be Computed from the Sparse Representation

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.

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

In order to prove this theorem, we need to establish a few preliminaries. Recall that in G^α, for two classes E and E', we say that $E > E'$ if E and E' belong to different components and there exist events $x \in E$ and $y \in E'$ with $y \in \mathit{cone}(x)$. A class E is a *maximal class* if it contains a latest event and there is no class E' with $E' > E$.

We first show that in the sparse representation, we can correctly determine if two classes, each containing a latest event, belong to the same component or not.

3.3.2 Lemma: Two classes E and E', each containing a latest event, belong to the same component in $G^\alpha(t)$ if and only if they belong to the same component in $\mathcal{S}^\alpha(t)$.

Proof: Let v and w denote a latest event from each class. It suffices to show that a path $v \rightsquigarrow w \in G^\alpha_{\equiv}(t)$ if and only if it can be inferred in $\mathcal{S}^\alpha(t)$. Let v_- be the earliest predecessor of v such that a path $v_- \rightsquigarrow w \in G^\alpha_{\equiv}(t)$. Since w is a latest event and hence considered open, it follows from the correctness of $H^\alpha_{\equiv}(t)$ that $v_- \rightsquigarrow w \in G^\alpha_{\equiv}(t)$ if and only if $(v_-, w) \in H^\alpha_{\equiv}(t)$. Since $v \rightsquigarrow v_-$ always exists, the existence of a path $v \rightsquigarrow w \in G^\alpha_{\equiv}(t)$ is correctly inferred in $\mathcal{S}^\alpha(t)$. □

Our algorithm for determining maximal classes in the sparse representation works as follows:

Let $\mathcal{C}^\alpha_l \subseteq \mathcal{C}^\alpha$ be the set of classes containing a latest event. Since each latest event must be considered open, $\mathcal{C}^\alpha$ and hence $\mathcal{C}^\alpha_l$ contains all classes in G^α containing a latest event.

For two classes $E, E' \in \mathcal{C}^\alpha_l$, we say that $E' >_s E$ if the two conditions are met:

- E and E' are in distinct components in $\mathcal{S}^\alpha$, and
- there exist $x \in E'$ and $y \in E$ such that x is a latest event and $y \in \mathit{cone}(x)$.

Note that *sparse-cone*$(x) \subseteq H^\alpha$ as x is a latest event, and that $y \in \mathit{cone}(x)$ can be inferred from *sparse-cone*(x).

We say that a class $E \in \mathcal{C}^\alpha_l$ is *maximal* in $\mathcal{S}^\alpha(t)$ if there does not exist any $E' \in \mathcal{C}^\alpha_l$ for which $E' >_s E$. We next show that a class containing a latest event is maximal in $G^\alpha(t)$ if and only if it is maximal in $\mathcal{S}^\alpha(t)$.

3.3.3 Lemma: For a class E containing a latest event, there exists a class E' in $G^\alpha(t)$ such that $E' > E$ if and only if there exists a class E_s in $\mathcal{S}^\alpha(t)$ such that $E_s >_s E$.

Proof: Suppose $E' > E$. Then there exist $x \in E'$ and $y \in E$ such that $y \in \mathit{cone}(x)$ and the classes E, E' belong to different components.

For any replica R^β, let v^β_* denote the latest event from R^β that is known at R^α at time t. If $x = v^\beta_i$, let $x_* = v^\beta_*$. Let E_s be the class containing x_*. Since E and E_s contain latest events, both these classes belong to $C^\alpha_l(t)$. We now show that $E_s >_s E$, that is, both conditions above are met.

Assume, for a contradiction, that the first condition is violated, that is, E_s and E are determined to be in the same component in $\mathcal{S}^\alpha(t)$. By Lemma 3.3.2, E_s and E belong to the same component in $\mathcal{S}^\alpha(t)$ if and only if they also belong to the same component in $G^\alpha_{\equiv}(t)$. Since E_s and E belong to the same component in $\mathcal{S}^\alpha(t)$ and hence in $G^\alpha_{\equiv}(t)$, there is a path from $y \in E$ to $x_* \in E_s$ in $G^\alpha_{\equiv}(t)$. This implies that there is a path y to x in $G^\alpha_{\equiv}(t)$ since $x \in \mathit{cone}(x_*)$. But then E' and E are in the same component in $G^\alpha_{\equiv}(t)$, a contradiction. Hence E_s and E belong to different components.

For the second condition, $y \in \mathit{cone}(x)$ implies $y \in \mathit{cone}(x_*)$. Since x_* is a latest event, *sparse-cone*(x_*) is contained in $H^\alpha(t)$. By Lemma 3.2, *sparse-cone*(x_*) suffices to determine that $y \in \mathit{cone}(x_*)$. Hence, we get $E_s >_s E$.

For the converse, suppose $E_s >_s E$ in $\mathcal{S}^\alpha(t)$. This implies the two classes do not belong to the same component and there exist $x \in E_s$ and $y \in E$ such that x is a latest event and $y \in \mathit{cone}(x)$ can be inferred from $\mathit{sparse\text{-}cone}(x)$. But E_s belongs to $G^\alpha(t)$, and by Lemma 3.3.2, E and E_s belong to distinct components. Also, $\mathit{sparse\text{-}cone}(x)$ is a subgraph of $\mathit{cone}(x)$. It follows that $E_s > E$ in $G^\alpha(t)$. Setting $E' = E_s$ completes the proof. □

Proof of Theorem 3.3.1: In both $G^\alpha(t)$ and $\mathcal{S}^\alpha(t)$, only a class E that contains a latest events can be a maximal class. By Lemma 3.3.3, for any such class E, if there exists a witness for non-maximality of E in $G^\alpha(t)$, then there also exists a witness in $\mathcal{S}^\alpha(t)$, and vice versa. The theorem follows. □

4 Intractability with Unrestricted Communication

We show in this section that, with completely unrestricted asymmetric communication, no sparse representation that operates in bounded space can implement the specification described in Section 2 correctly. The intuition behind this result is that old conflict resolutions may matter much later if another replica made choices implicating these old events. As a consequence, old conflict resolutions must be preserved until every replica has acknowledged them, thus requiring unbounded storage in the general case.

We first give an example where omitting an edge between two events leads to a scenario where a latest event is inferred to be non-maximal, though it is maximal. This example provides the key insight for the more general impossibility proof at the end of the section.

Consider six replicas named $R^a \ldots R^f$. We will consider blocks of time where each block consists of several time instances. For each block of time $t = j$, the following actions take place sequentially:

- R^a updates itself to v_j^a and then sends his state to replicas $R^c \ldots R^f$.
- R^b updates itself to v_j^b and then sends his state to replicas $R^c \ldots R^f$.
- R^c uses the following actions to, essentially, declare that v_j^a dominates v_j^b and create a path from v_j^a to v_j^b in the graph $G^c_{\equiv}$:
  - updates itself to v_{2j}^c and creates $v_{2j}^c \longrightarrow v_j^b$
  - updates itself to v_{2j+1}^c and creates $v_{2j+1}^c \Longrightarrow v_j^a$
- R^d does the opposite of R^c, essentially declaring that v_j^b dominates v_j^a and creating a path from v_j^b to v_j^a in the graph $G^d_{\equiv}$:
  - updates itself to v_{2j}^d and creates $v_{2j}^d \longrightarrow v_j^a$
  - updates itself to v_{2j+1}^d and creates $v_{2j+1}^d \Longrightarrow v_j^b$

This sequence of actions goes on till $t = k$ for some suitably large k and replica R^c can not store the path from v_j^a to v_j^b for all j. Let i be the index for which it omits the path from v_i^a to v_i^b (the path consists of three edges and omitting one of the two edges other than the default $v_{2i+1}^c \longrightarrow v_{2i}^c$ suffices to omit the path). Note that replica R^c has not communicated with any other replica so far and so no other replica knows of a path between v_i^a to v_i^b.

The replicas R^e and R^f do the following actions in order for the time block $t = i$.

- R^e does the following:
  1. creates an event v^e
  2. sends its state to R^f
  3. creates an event v^e_+ and an edge $v^e_+ \Longrightarrow v^b_i$
- R^f does the following:
  1. creates an event v^f and an edge $v^f \Longrightarrow v^a_i$
  2. creates an event v^f_+ and an edge $v^f_+ \Longrightarrow v^e$

They sit idle (except for receiving the graphs from R^a and R^b) for all other time blocks. Note that they never communicate with any other replica and so their actions are independent of other replicas.

At time $t = k$, all replicas communicate their state to a new replica, say R^g. This replica has been sitting idle until now, so at this instant its state is simply the entire record of actions of all other replicas. The latest events created by replicas R^a to R^f are v^a_k, v^b_k, v^c_{2k+1}, v^d_{2k+1}, v^e_+, and v^f_+, respectively.

Consider first what R^g would see if R^c had not dropped an edge. The events $v^a_k, v^b_k, v^c_{2k+1}, v^d_{2k+1}$ belong to one component, C_0; the (unique) cycle[2] $v^a_k \overset{\equiv}{\Longleftrightarrow} v^c_{2k+1} \longrightarrow v^c_{2k} \longrightarrow v^b_k \overset{\equiv}{\Longleftrightarrow} v^d_{2k+1} \longrightarrow v^d_{2k} \longrightarrow v^a_k$ is a witness to this fact. There are four classes in component C_0:

- $E^0_1 = \{v^a_k \overset{\equiv}{\Longleftrightarrow} v^c_{2k+1}\}$
- $E^0_2 = \{v^c_{2k}\}$
- $E^0_3 = \{v^b_k \overset{\equiv}{\Longleftrightarrow} v^d_{2k+1}\}$
- $E^0_4 = \{v^d_{2k}\}$

The events v^e_+, v^f_+ belong to another component, C_1; the cycle $v^e_+ \longrightarrow v^e \overset{\equiv}{\Longleftrightarrow} v^f_+ \longrightarrow v^f \overset{\equiv}{\Longleftrightarrow} v^a_i \overset{\equiv}{\Longleftrightarrow} v^c_{2i+1} \longrightarrow v^c_{2i} \longrightarrow v^b_i \overset{\equiv}{\Longleftrightarrow} v^e_+$ is a witness to this fact. There are five classes in component C_1:

- $E^1_1 = \{v^f_+ \overset{\equiv}{\Longleftrightarrow} v^e\}$
- $E^1_2 = \{v^e_+ \overset{\equiv}{\Longleftrightarrow} v^b_i \overset{\equiv}{\Longleftrightarrow} v^d_{2i+1}\}$
- $E^1_3 = \{v^f \overset{\equiv}{\Longleftrightarrow} v^a_i \overset{\equiv}{\Longleftrightarrow} v^c_{2i+1}\}$
- $E^1_4 = \{v^c_{2i}\}$
- $E^1_5 = \{v^d_{2i}\}$

Class E^1_1 contains a latest event (v^f_+) and is not dominated by any class of component C_0, hence it is a maximal class and v^f_+ is a maximal event. E^1_1 is the only maximal class of component C_1.

Now consider the situation where R_c does omit the path between v^a_i and v^b_i and so R^g does not know about it. Now C_1 splits into five different components, one for each equivalence class.

[2]We use $x \overset{\equiv}{\Longleftrightarrow} y$ to denote the bi-directional equivalence edge from x to y in $G^k_{\equiv}$

Two of them contain latest events and may qualify as maximal classes: $E_1^1 = \{v_+^f \stackrel{\equiv}{\Longleftrightarrow} v^e\}$ and $E_2^1 = \{v_+^e \stackrel{\equiv}{\Longleftrightarrow} v_i^b \stackrel{\equiv}{\Longleftrightarrow} v_{2i+1}^d\}$. However $E_2^1 > E_1^1$ because $v^e \in cone(v_+^e)$ and they belong to two different components, and $E_3^0 > E_2^1$ because $v_i^b \in cone(v_k^b)$. Thus, the only maximal classes are in C_0, and the set of maximal events does not contain v_+^f—the dropped edge changes what is reported to the user by R^g.

The above example is generalized to prove the following theorem:

4.1 Theorem: No representation with bounded storage space at each replica can decide if a latest event is maximal or not.

Proof: Suppose, for a contradiction, that B bits of storage at each replica are sufficient to determine which events are maximal. Let us consider a variation on the example above where, at each time step $j < k$, we choose at replica R^c whether v_j^a dominates v_j^b, yielding 2^k different action sequences for R^c. Note that B bits can encode at most 2^B possible action sequences and so by choosing $k \geq B + 1$, we can be sure that there exist two distinct action sequences that cannot be distinguished by the representation at R^c using only B bits. Suppose the two sequences differ on the existence of a path between v_l^a and v_l^b—that is, in one of them, $S_c^{yes}(l)$, there exists a path from v_l^a and v_l^b and in the other, $S_c^{no}(l)$, there does not. Consider now the time i when replicas R^e and R^f perform the action described above, creating events v^e, v_+^e, v^f, and v_+^f: there are k choices for i, yielding k action sequences $S_{e,f}(i)$. The action sequence at the other replicas are the same as above.

We can specify a "possible world" by the action sequences at each replica. There are $2^k * k$ possible worlds: every action sequence at R^c combined with every action sequence $S_{e,f}(i)$ and the single action sequence at other replicas. We now focus on two specific possible worlds. The first one consists of action sequence $S_c^{yes}(l)$ at R^c combined with $S_{e,f}(l)$—the sequence where R^e and R^f create their events at time l. We call this first possible world I^{yes}. The second possible world consists of action sequence $S_c^{no}(l)$ at R^c also combined with $S_{e,f}(l)$. We call this second possible world I^{no}.

Consider now the data received by replica R^g at time k: it is identical, since the only difference in the action sequences is the creation of a path between v_l^a and v_l^b at R^c, which, by hypothesis, is not stored. However, as described in the example above, in possible world I^{yes} the event v_+^f is maximal, whereas in possible world I^{no} it is not. Hence there is not enough information to determine which events are maximal, a contradiction. □

5 Related Work

Both theoretical underpinnings and efficient implementation strategies for version vectors [23] and vector clocks [10, 22] have received a great deal of attention in the literature and have been used in many systems (e.g. Coda [19, 33, 20], Ficus [13], and Bengal [9]); numerous extensions and refinements have also been studied—see [4] for a recent survey. We conjecture that some of these ideas can be applied to improve the efficiency of our sparse representation. In that vein, generalizations of recent work on the use of PVEs in WinFS [21] may be applicable to our sparse representation, and useful in reducing the per-object overhead, when we extend our work to replicas with a large number of objects. 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 [26], 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 [2]. The *version histories* used in the Reconcile file synchronizer [15] and the Clique peer-to-peer filesystem [29], as well as Kang's *hash histories* [16], all represent the causal history of the system directly—storing and transmitting (hashes of) complete histories of updates — rather than deducing causal ordering from reduced representations such as clock vectors. They treat identical file contents as agreement, and keep a version history of each file (storing a SHA1 hash of the file contents after each synch operation). Agreement is therefore possible even between past versions of two different files, and can be used to determine whether two different files may be considered causally related, and hence not in conflict. Identical contents can also be used to find the latest common ancestor of two files. This can reduce communication costs during conflict resolution, by only transmitting the modifications to the common ancestor rather than the entire file. 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 Section 2 directly, bypassing the sparse representation.

Care must be taken to avoid equating two equivalent instances of an object, when the object can take on the same value more than once. Consider a file initially containing v. Editing produces a version with contents v'. Later, on another replica, the file is edited to deliberately undo the changes, producing a version containing v again. If one replica, R^a hears only of the partial history v, v', and the another R^b hears of the complete v, v', v, then one should not equate the v on R^a with the latest v on R^b — else the system will converge to v', losing the final edit. [16] reduces the likelihood of this happening by using both the file contents and a "generation number" for equality tests.

Parker's original paper on version vectors [23] argues against implicit agreement based solely on equality of value. Consider an object representing an account balance: if it were originally \$1000 and was reduced to \$900 on two different nodes, perhaps reconciliation should combine the differences and set it to \$800?) Application-level knowledge is required to explicitly mark events as in agreement, even if their values are equal.

For applications where equality can safely imply agreement, though, the optimization appears to have significant benefit. Kang [16] used simulation over trace data from `sourceforge.net` to show that "coincident equalities" (independent updates or reconciliations that resulted in identical contents) appear to be common. The paper showed that recognizing equality as agreement substantially sped up convergence. Further, they showed that the fact that version vectors did not remember "agreement events" noticeably delayed convergence: updates of one replica out of a set of replicas with equal values occured in the trace data. The update conflicted with other replicas in the set, rather than superseding it. Convergence was delayed until lengthy periods of inactivity.

Almeida, Baquero, and Fonte [3] do not use local agreement events, as we do, but rather represent reconciliation in a manner that is fundamentally quite similar to a version vector approach that would take the pointwise maximum for agreement events. However, in a restriction reminiscent

of our reciprocality requirement, they only consider synchronous synchronization (both replicas synchronize at the same time).

Many refinements to version vectors and vector clocks have addressed performance issues that may be relevant to our work. Almeida et al. [3] address the problem of efficiently representing version vectors when replicas can leave and join the system. Almeida, and Baquero[1] show that version vectors can be represented using only a bounded amount of information per replica, when they are only being used to check dominance or conflict between *current* versions of data in a distributed system. Other work [24, 28, **?**] has focused on improving the way vector clock schemes scale with the number of replicas.

Matrix clocks [32, 35] generalize vector clocks by explicitly representing clock information about other processes's views of the system's execution. Matrix clocks, too, have been optimized [7, 30, 14] and generalized [18]. We leave for (interesting) future work the question of whether agreement events such as the ones we are proposing could be generalized along similar lines.

Reconciliation protocols for optimistically replicated data can be divided into two general categories [31]: *state transfer* and *operation transfer* protocols. We have concentrated on state-based protocols in this work. However, a number of systems (e.g., Bayou [8, 34], IceCube [17, 27], and Ceri's work [5]) 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] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. Bounded version vectors. In *DISC*, pages 102–116, 2004.

[2] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Panasync: dependency tracking among file copies. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 7–12. ACM Press, 2000.

[3] Paulo Sergio Almeida, Carlos Baquero, and Victor Fonte. Version stamps – decentralized version vectors. In *Proceedings of 22nd IEEE International Conference on Distributed Computing Systems (ICDCS '02)*, 2002.

[4] Roberto Baldoni and Michel Raynal. A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002. http://dsonline.computer.org/0202/features/ bal.htm.

[5] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, and Pierangela Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3):225–246, 1995.

[6] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of PODC'87*, August 1987.

[7] Lúcia M. A. Drummond and Valmir C. Barbosa. On reducing the complexity of matrix clocks. *Parallel Computing*, 29(7):895–905, 2003.

[8] W. Keith Edwards, Elizabeth D. Mynatt, Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, and Marvin M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *ACM Symposium on User Interface Software and Technology (UIST), Banff, Alberta*, pages 119–128, October 1997.

[9] Todd Ekenstam, Charles Matheny, Peter L. Reiher, and Gerald J. Popek. The Bengal database replication system. *Distributed and Parallel Databases*, 9(3):187–210, 2001.

[10] Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, Aug 1991.

[11] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Schema-directed data synchronization. Technical Report MS-CIS-05-02, University of Pennsylvania, March 2005. Supersedes MS-CIS-03-42.

[12] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 2006. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.

[13] 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 *Proceedings of the ER Workshop on Mobile Data Access*, pages 254–265, 1998.

[14] A. Heddaya, M. Hsu, and W. Weihl. Two phase gossip: managing distributed event histories. *Information Science*, 49(1-3):35–57, 1989.

[15] John H. Howard. Reconcile user's guide. Technical Report TR99-14, Mitsubishi Electronics Research Lab, 1999.

[16] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *23rd IEEE International Conference on Distributed Computing Systems (ICDCS '03)*, 2003.

[17] Anne-Marie Kermarrec, Antony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of diverging replicas. In *ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC), Newport, Rhode Island*, pages 210–218, August 2001.

[18] Ajay D. Kshemkalyani. The power of logical clock abstractions. *Distrib. Comput.*, 17(2):131–150, 2004.

[19] Puneet Kumar. Coping with conflicts in an optimistically replicated file system. In *1990 Workshop on the Management of Replicated Data*, pages 60–64, Houston, TX, Nov 1990.

[20] Puneet Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the annual USENIX 1995 Winter Technical Conference*, pages 95–106, January 1995. New Orleans, LA.

[21] Dahlia Malkhi and Douglas B. Terry. Concise version vectors in WinFS. In Pierre Fraigniaud, editor, *Proceedings of the 19th International Conference on Distributed Computing, DISC 2005*, volume 3724 of *Lecture Notes in Computer Science*, pages 339–353. Springer-Verlag, September 26-29 2005.

[22] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et. al., editor, *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel & Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V., 1989.

[23] D. S. Parker, Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Software Eng. (USA)*, SE-9(3):240–247, 1983.

[24] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, October 1997.

[25] Benjamin C. Pierce et al. Harmony: A synchronization framework for heterogeneous tree-structured data, 2006. `http://www.seas.upenn.edu/~harmony/`.

[26] Benjamin C. Pierce and Jérôme Vouillon. 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.

[27] Nuno Preguia, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, May 2002.

[28] D. Ratner, Peter Reiher, and Gerald Popek. Dynamic version vector maintenance. Technical Report CSD-970022, University of California, Los Angeles, June 1997.

[29] Bruno Richard, Donal Mac Nioclais, and Denis Chalon. Clique: a transparent, peer-to-peer collaborative file sharing system. In *International Conference on Mobile Data Management (MDM), Melbourne, Australia*, January 2003.

[30] Frederic Ruget. Cheaper matrix clocks, July 1994. CS/TR-94-63, Chorus Systems, Montigny le Bx, France.

[31] Yasushi Saito and Marc Shapiro. Replication: Optimistic approaches. Technical Report HPL-2002-33, HP Laboratories Palo Alto, Feb. 8 2002.

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

[33] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[34] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 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*, pages 172–183, 1995.

[35] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Principles of Distributed Computing*, pages 233–242, 1984.