P2P Replica Synchronization with Vector Sets

Dahlia Malkhi
Microsoft Research Silicon Valley

Lev Novik
Microsoft Corporation

Chris Purcell
Cambridge University

dalia@microsoft.com

ABSTRACT

This paper describes an enhanced replica synchronization mechanism built in Microsoft's WinFS replica management system.

The system reconciles autonomously-operating replicas in a completely peer-to-peer manner, without employing a central master or locking. The main challenge is for two replicas to exchange meta-information efficiently about (potentially numerous) data objects in order to discover what updates they are missing, and detect conflicts.

The paper introduces a novel bundling mechanisms called VS, that groups together multiple objects and represents their state in a single version-vector. VS provides improved storage and communication overheads over previously known optimistic replication schemes, in the following sense. Under normal, low-fault situations, it maintains and communicates as little as a single version vector in order to represent precedence ordering of the entire set of data objects. Moreover, under settings of severe communication disruptions, VS degenerates to no worse than a single vector per object. This dramatically improves the complexities described in a preliminary write-up of the WinFS replication scheme.

The VS mechanism has potentially wide applicability as a mechanism for compactly handling synchronization of arbitrarily overlapping groups of objects.

1. INTRODUCTION

Consider the following problem. A set of data *objects* is loosely replicated over a group of replicas. Replicas are allowed to introduce updates on objects independently. Occasionally, some pair of replicas communicate in order to synchronize the states of all the objects they store. They need to convey to each other the latest state of each object, determine which objects need to be shipped over, and alert to any conflicting, independent simultaneous updates. As the full set of objects may potentially be very large, capturing this causality information precisely in a concise form is a challenging task.

This problem arises at the core of the replica reconciliation and conflict detection mechanism inside Microsoft's WinFS system. WinFS supports weakly consistent file replication among autonomous hosts, e.g., laptops, mobile PDAs, and PCs. WinFS encompasses a visionary end-user experience that allows users to hop on a plane, be at home, at the office, or in an ad-hoc network relationship with others; yet, all the time, the user experience remains unchanged, as she may transparently access her files. To this end, disconnected replicas are allows access to files both for reads and for updates. Periodically, and whenever communication is facilitated, WinFS reconciles replicas in a completely peer to peer fashion, with no central coordinator or master. Although this fundamental optimistic replication paradigm is well known in distributed computing, the applications that are targeted by WinFS mandate taking scale more seriously than ever before. In particular, e-mail repositories, log files, digital libraries, and application databases can easily reach millions of objects. Hence, communicating even a single bit per object (*e.g.*, a 'dirty' bit) in order to be able synchronize replicas might simply be too costly.

This paper presents a precise description and correctness proof of an enhanced replica synchronization scheme deployed in WinFS, which we name *Vector Sets* (VS). We produce a systematic study of the performance gains of VS and provide a comparison with previous weakly-consistent replication schemes. The results demonstrate a substantial reduction in the storage and communication overhead associated with replica synchronization in the following sense.

Under normal, low-fault situations, VS maintains and communicates as little as a single version vector in order to represent precedence ordering of the entire set of data objects. This is in sharp contrast to the prevailing method for almost two decades: Associating a *version vector* per object [1]. A version vector contains a count of updates from each replica. For example, a vector $\langle (A, 1), (B, 3), (C, 0), (D, 5) \rangle$ indicates one update by A, three by B, none by C, and five by D. It is easy to determine between two version vectors whether one precedes the other. The overheads incurred by the version vectors method are as follows. With N objects, R replicas, and q recent updates, storing a version vector per object costs each replica $N \times R$. Communicating its knowledge during synchronization costs a replica at least $q \times R$ and at worst $N \times R$. An additional communication overhead of $q \times R$ results from communicating one version vector with each object that is being updated. VS brings these costs down to N, R and q, respectively. Table 1 summarizes the various costs incurred with VS and version vectors.

Under settings of severe communication disruptions, performance may gradually degrade. However, it is provably never worse than the performance of version vectors (see Table 1). In such settings, VS dramatically improves the complexities described in a preliminary writeup of the WinFS replication scheme [4], which was named PVE. More specifically, PVE incurs an unbounded price when synchronization between parties is disrupted: An uncompleted synchronization must either be aborted, or incurs a management cost that could grow beyond a per-object version vector.

In summary, VS combines the good properties of version vectors and PVE. Specifically, under all circumstances, VS is at least as good as the version vector scheme. During normal, low-fault settings, VS is as good or slightly improves on the costs of PVE.

We first present in Section 3 a simple version of VS, and call it SVS (*Simple Vector Sets*). SVS serves to point to the core idea behind our improvement. It has the same upper and lower bound complexities as VS, but its average case behavior is sub-optimal.

Briefly, SVS works as follows. A single vector counts updates made by the replicas on all data objects. For example, $\langle (A,1),(B,3),(C,0),(D,5)\rangle$ indicates one update by A on some date object, three by B on some object(s), none by C on any object, and five by D on some object(s). Consider a replica, say A, whose objects are represented by the vector above. Suppose that A tries to synchronize with another replica, say C, whose knowledge vector is $\langle (A,1),(B,5),(C,2),(D,5)\rangle$. Then A needs to obtain from C any updates among $\{(C,1),(C,2),(B,4),(B,5)\}$ which were not overridden. Specifically, presume that object w is updated first in $(C,1)$ and then by $(B,5)$. Object x is updated in $(B,4)$ followed by $(C,2)$. Then C only needs to send over to A objects w,x in their most updated versions, namely, $(B,5)$, $(C,2)$, respectively.

An interesting challenge occurs if the communication is disrupted after object w ($(B,5)$) is received, but before object x ($(C,2)$) is. Note that the knowledge vector of C cannot be combined at A. We resolve this problem in SVS as follows. After the disrupted synchronization with C, A continues holding an unchanged knowledge vector. In addition, for any individual object that has been updated, e.g., w in the example scenario above, A maintains an independent version vector. Individual version vectors are updated independently during synchronization and data updates. At any point where an individual version vector becomes dominated by the global knowledge vector, it may be erased.

SVS clearly has a worst-case cost no greater than version vectors, since in the extreme, every object holds an independent vector. But in most normal situation, it will perform much better.

Moreover, we can further optimize average-case storage and communication costs. In our full VS scheme, presented in Section 4, we introduce the notion of *Vector Sets*. These are sets of objects that share a single version vector. The Initially, each replica has one set only, the set of all objects, and they are represented by a single version vector. When a synchronization is interrupted, this set splits into two subsets (updated and non-updated), and subsequent interrupted synchronization procedures may cause additional splits. Merging of sets is made possible by having future synchronization procedures run to completion. Compared with SVS, the full VS scheme stores fewer individual vectors by aggregating sets of objects. For example, as a result of a single interrupted synchronization that passed q objects, SVS stores $q+1$ vectors (one global, and one for each updated object), whereas VS stores only two vectors (one for the updated subset, and one for the non-updated subset).

Contribution. Our contributions are as follows. We present a method for aggregating causality information using the novel notion of Vector Sets (VS). When deployed for peer-based synchronization, VS provides improved storage and communication overheads over previously known optimistic replication schemes. VS is deployed in the WinFS storage project at Microsoft.

In addition to its good performance, the VS mechanism has potentially wide applicability as a mechanism for compactly handling synchronization of arbitrarily overlapping groups of objects.

2. PRELIMINARIES

2.1 Problem Statement

The system consists of a collection of data objects, potentially numerous. Each object might be quite small, *e.g.* a mail entry or even a status word. Objects are replicated on a set of replicas. Each replica may locally introduce updates to any object without any concurrency control. These updates create a partial ordering of object versions, where updates that sequentially follow one another are causally related, but unrelated updates are *conflicting*.

Our focus is on distributed systems in which updates overwrite previous versions. In such state-based replication systems, only the most recent version of any object needs to be sent during synchronization.

Our goal is to provide a replica reconciliation and conflict detection mechanism. The mechanism should provide two communicating replicas with the means to detect precedence ordering on object versions that they hold and detect any conflicts while requiring only a small amount of per-object overhead. With this mechanism, replicas can bring each other up-to-date and report conflicts.

	Version vectors	PVE	VS
store lower bound	$\widetilde{O}(N \times R)$	$\widetilde{O}(N + R)$	$\widetilde{O}(N + R)$
store upper bound	$\widetilde{O}(N \times R)$	unbounded	$\widetilde{O}(N \times R)$
comm lower bound	$\widetilde{O}(q \times R)$	$\widetilde{O}(q + R)$	$\widetilde{O}(q + R)$
comm upper bound	$\widetilde{O}(N \times R)$	unbounded	$\widetilde{O}(N \times R)$

Table 1: *Lower and Upper Bounds Comparison of VS, PVE and version-vectors.*

More precisely, we now describe objects, versions, and causality. An object is identified uniquely by its name. We identify objects by letters, e.g., W, X, Y, Z. Every object instance, in addition to its name, has a version. An object's version is a pair $\langle \textit{replica id}, \textit{counter} \rangle$. When we need to explicitly associate a version with an object we denote this by *object-id.version*.

There is a partial, causal ordering among different versions of the same object. When a replica A updates an object X it sets the version of X to a new version v. The set $\mathcal{W}$ of versions of X that are previously known by replicas A *causally precedes* version v. In notation, $\mathcal{W} \prec v$. For every version $w \in \mathcal{W}$, we likewise say that w *causally precedes* v; in notation, $w \prec v$. Causality is transitive.

Since the system permits concurrent updates, the causality relation is only a partial order, i.e. multiple versions of the same object might follow any single version. When two versions do not follow one another, they are *conflicting.* In other words, if $x.w \not\prec x.v \;\wedge\; x.v \not\prec x.w$, then v and w are said to conflict.

It is desirable to detect and resolve conflicts, either automatically (when application-specific conflict resolution code is available) or by alerting users who can resolve conflicts manually. In either case, a resolution of conflicting versions is a version that causally follows both. For example, here is a conflict and its resolution: $v_0 \prec v \prec w \;;\; v \not\prec u \;;\; u \not\prec v \;;\; v_0 \prec u \prec w$.

New versions override previous ones, and so replicas are generally only interested in the most recent version available; versions that causally precede it are obsolete and carry no valuable information. This simple rule is complicated by the fact that multiple conflicting versions may exist; in this case, replicas are interested in all concurrent versions until they are resolved.

2.2 Synchronization Framework

Synchronization occurs between a pair of peer replicas, a *requestor* and a *source.* Any replica may initiate a synchronization, so in particular, reciprocal synchronization may occur in the reverse direction. The interaction consists of four steps.

1. The requestor r contacts a source and sends a *knowledge* description, encompassing all of the latest object versions r has in store.
2. The source s responds by sending the latest version of every object o which is not known to r.
3. For every object o the requestor receives, it integrates o into store, and raises a conflict alarm where needed.
4. For every new object version r has stored, it updates its repository with the new version. The new version, as well as any version that directly or indirectly precedes it, is updated in r's knowledge, even though it may not have been explicitly added to store.

2.3 Performance Measures

This paper is concerned with mechanisms that facilitate synchronization of different replicas. The challenge is to bring the storage and communication costs associated with replica reconciliation (significantly) down. More precisely, we focus on two performance measures:

Storage is the total number of overhead bits stored in order to preserve version ordering.

Communication is (i) the total number of bits communicated between two replicas in order to determine which updates are known to one replica but not the other, and (ii) any overhead data that is transferred along with objects' states in order to determine precedence/conflicts.

2.4 Versions and Version-Vectors

DEF. 2.1 (PER-REPLICA COUNTER). *Let X be a replica. The* versions *generated by replica X across all objects in the collection are the ordered sequence* $\{\langle X, i \rangle\}_{i=1,2,...}$.

DEF. 2.2 (VERSION VECTORS). *Let $X_1, ..., X_R$ be the set of replicas. A* version vector *is an R-array of tuples of the form* $(\langle X_1, i_1 \rangle, ..., \langle X_R, i_R \rangle)$.

A version vector $(\langle X_1, i_1 \rangle, ..., \langle X_R, i_R \rangle)$ dominates *another vector* $(\langle X_1, j_1 \rangle, \ldots, \langle X_R, j_R \rangle)$ *if $i_k \geq j_k$ for $k = 1..R$, and it* strictly dominates *if $i_\ell > j_\ell$ for some $1 \leq \ell \leq R$.*

By a natural overload of notation, we say that a vector $(\langle X_1, i_1 \rangle, \ldots, \langle X_R, i_R \rangle)$ *dominates a version $\langle X_k, j_k \rangle$ if $i_k \geq j_k$; strict domination follows accordingly with strong inequality.*

3. SIMPLE VECTOR SETS (SVS)

Versions and Predecessor Vectors. The Simple Vector Sets scheme uses the per-replica counter defined in Definition 2.1, which enumerates updates generated by the replica on all objects. Hence, a replica X_r maintains a local counter

c. When replica X_r updates an object o, it increments the local counter and sets o's version to $\langle X_r, c\rangle$.

The predecessors vector of an object o is a vector that dominates all the versions that causally precede $o.version$ (including $o.version$). It may (and usually does) dominate versions of other objects, but no other versions of o.

Knowledge and Predecessors. A replica X_r maintains in $X_r.knowledge$ a version vector representing versions it knows of. An object's individual *predecessors* vector contains information about preceding versions on that object. When an individual *predecessors* vector can be replaced by $X_r.knowledge$, it is set to empty ($\perp$). Empty *predecessors* implicitly imply that $X_r.knowledge$ contains all (and no more) than the object's preceding versions. We will see below the precise conditions under which $o.predecessors$ can be dropped.

Generating a New Version. When a replica X_r generates a new update on an object o, the new version $\langle X_r, c\rangle$ is inserted into $X_r.knowledge$ right away. Then, if $o.predecessor$ is $\perp$, nothing needs to be done to it. Implicitly, this means that the versions dominated by $X_r.knowledge$ causally precede the new version. If $o.predecessors$ is not empty, then the new version is inserted to $o.predecessors$ as well. Indeed, all the versions that were dominated by the previous $o.predecessors$ causally precede the new update.

Synchronization. Figure 1 below describes the SVS synchronization protocol.

When a requestor X_r obtains a new object version from X_s, it first keeps an individual *predecessor* vector for this object. Later, it may omit this vector if it is dominated by $X_r.knowledge$.

In addition to this, $X_s.knowledge$ needs to be merged with $X_r.knowledge$ at the end of a complete synchronization. The reason for this is that there may be some versions that X_s knows about that are overridden by later versions. For example, X_s may have version $\langle B, 5\rangle$ of an object o, overriding a previous version $\langle C, 6\rangle$. Hence, X_s only sends $\langle B, 5\rangle$ over to X_r during synchronization, and never sends $\langle C, 6\rangle$ explicitly to X_r. The goal of the merging is to produce a vector that represents a union of all the versions included in $X_r.knowledge$ and $X_s.knowledge$, and replace $X_r.knowledge$ with it. For example, merging $X_s.knowledge = (\langle A, 3\rangle, \langle B, 5\rangle, \langle C, 6\rangle)$ into $X_r.knowledge = (\langle A, 7\rangle, \langle B, 3\rangle, \langle C, 1\rangle)$ yields $(\langle A, 7\rangle, \langle B, 5\rangle, \langle C, 6\rangle)$.

Conflicts. When conflicting versions of the same object exist at replica X, the base knowledge $X.knowledge$ can be set to dominate both of them. However, in order to correctly detect further conflicts, individual *predecessors* need to be stored for each version. We exemplify this with a simple scenario. Suppose that replica X has the following two versions of a single object. Version $v_1 = \langle B, 5\rangle$ has predecessors $(\langle A, 1\rangle, \langle B, 5\rangle, \langle C, 2\rangle)$, and version $v_2 = \langle C, 3\rangle$ has predecessors $(\langle A, 1\rangle, \langle B, 4\rangle, \langle C, 3\rangle)$. These are conflicting versions. Replica X can set $X.knowledge$ to $(\langle A, 1\rangle, \langle B, 5\rangle, \langle C, 3\rangle)$. Then, during synchronization, if X requests updates from X_s, $X.knowledge$ suffices to describe the versions known to X, i.e., X_s does not need to send over to X any version dominated by $X.knowledge$.

Now consider the reverse situation, where X is prompted for updates by another replica, X_r, and X sends the versions $\langle B, 5\rangle$ and $\langle C, 3\rangle$. Then neither version must override the other, and hence, X must send the original, individual predecessors vector along with each version.

1. Requestor X_r sends source X_s its knowledge set $X_r.knowledge$, and a set IS of pairs $(w.name, w.predecessors)$, such that $w.predecessors \neq \perp$.
2. Source X_s responds with the following:
   (a) It sends $X_s.knowledge$.
   (b) It sends every object o (containing its name, version, predecessors vector, and data) for which $o.version \not\leq r.knowledge$, and there does not exist $w \in IS$ with $w.name = o.name$ and $o.version \leq w.predecessors$.
3. For every version o received from X_s, requestor X_r does the following:
   (a) Loop through all objects w in store, such that $w.name = o.name$:
   if $o.version \leq w.predecessors$ or ($w.predecessors = \perp$ and $o.version \leq r.knowledge$) then ignore o and stop loop;
   else if $w.version \leq o.predecessors$ or ($o.predecessors = \perp$ and $w.version \leq s.knowledge$) then delete w;
   (else o and w conflict.)
   (b) store o.
   (c) If $o.predecessors = \perp$, then set $o.predecessors = s.knowledge$.
   (d) For every object w in store, such that $w.name = o.name$ (these must be conflicting versions), link w to o and mark them conflicting.
4. Merge $X_s.knowledge$ into $X_r.knowledge$.
5. (Lazily) go through versions v such that (i) $v.predecessors \neq \perp$, (ii) v is not marked as conflicted with another version, and (iii) $X_r.knowledge$ dominates v: Set $v.predecessors = \perp$. As an additional minor optimization, for every version $v = \langle s, n_s\rangle$ such that $X_r.knowledge$ has $\langle s, n_s - 1\rangle$ at the X_s-position, update $X_r.knowledge$ to contain $\langle s, n_s\rangle$.

Figure 1: SVS Synchronization.

4. VECTOR SETS (VS)

This section describes the full Vector Sets (VS) scheme. The Vector Sets scheme strives to pack together sets of object versions for which there exists a single version vector that dominates all the versions that precede or include them.

Versions. The Vector Sets scheme uses the same per-replica counter as in Section 3 above. Hence, a replica X_r maintains a local counter c. When replica X_r updates an object o, it increments the local counter and sets o's version to $\langle X_r, c\rangle$.

Vector Sets and Predecessor Vectors. A replica X_r maintains a partition of the set of objects, $\{S_1, ..., S_c\}$. Each subset S contains a description $S.list$ of the objects in the set (e.g., a list of names, or a range of names). In order to keep costs down, object sets that can be enumerated concisely are preferred, e.g., consecutive ranges of objects. Each object set has an associated *predecessors* vector. $S.predecessors$ dominates all the versions that causally precede the versions in S, but not any other versions of objects in S. It may (and usually does) dominate versions of objects not included in S, that are therefore not in any ordering relations with the versions in S.

Conflicts. Conflicting versions are handled as in the case of SVS, i.e., by storing individual predecessor vectors with every conflicting version, until they are resolved.

Inserting a New Version. When a new version $v = \langle X_s, j_s\rangle$ is stored at X_r, every subset S that has $S.predecessors$ with $\langle X_s, j_s - 1\rangle$ at the X_s-position is updated to $\langle X_s, j_s\rangle$.

Synchronization. Figure 2 below describes the VS synchronization protocol.

A requestor X_r has in store subsets $\{R_1, ..., R_k\}$. It tries to obtain any newer version X_s has in store. X_s may have a different partition of the objects, say $\{S_1, ..., S_m\}$. This could be an explicit partition at X_s, or an implicit partition created by a disrupted synchronization.

Let S_j be a set of objects for which synchronization with X_s has completed, and $S_j.predecessors$ is its associated vector. Then merging S_j's predecessors into storage at X_r is done as follows. For every subset R at X_r such that $R \cap S_j \neq \emptyset$, split R into $R_1 = R \cap S_j$ and $R_2 = R \setminus S_j$ (note that, R_2 may be empty, in which case ignore it). Merge $S_j.predecessors$ into $R_1.predecessors$.

Merging subsets. Periodically, a replica X_r can merge two subsets S' and S'' if they have identical predecessor vectors. This would be the case, for example, after synchronizing with a replica that has both S' and S'' contained in a larger set.

4.1 VS Performance

For best performance, the VS scheme should strive to maintain concise representations of partitions. It achieves this as follows. Let there be an enumeration of all the objects (say, 'A' through 'Z'). During synchronization, the replicas exchange missing versions in order of the objects enumeration ('A', 'B', 'C', etc.). A partition that results from disrupted synchronization can always be described concisely.

1. Requestor X_r sends source X_s a list of object-sets descriptions $\{\langle R_1.list, R_1.predecessors\rangle, ..., \langle R_k.list, R_k.predecessors\rangle\}$.
2. Source X_s responds with the following, for every subset S it stores:
   (a) It sends $\langle S.list, S.predecessor\rangle$.
   (b) It sends every object $o \in S$ (including name, version, predecessors vector, and data) such that $o.name \in R.list$, for which $o.version \not\leq R.predecessors$. If there are multiple (conflicting) versions of o, it sends all versions in one, unbreakable message.
3. For every version o received from X_s in the context of an object set S, requestor X_r does the following:
   (a) Loop through all objects w in store, $w \in R_j$, such that $w.name = o.name$:
   if $o.version \leq R_j.predecessors$) then ignore o and stop loop;
   else if $w.version \leq S.predecessors$) then delete w;
   (else o and w conflict.)
   (b) store o.
   (c) For every object w in store, such that $w.name = o.name$ (these must be conflicting versions), link w to o and mark them as conflicting.
4. Let $\mathcal{S} = \{S_1, S_2, ..., S_k, T\}$ be object sets for which X_s sent complete information, where $T \subseteq S_{k+1}$ is the completed (sub)set of objects from the last set X_s attempted to send. For every object-set $S \in \mathcal{S}$ obtained, and for every R_i, split into $R'_i = R_i \cap S$ and $R''_i = R_i \cap S$. If R'_i is not empty, merge $S.predecessors$ into $R_i.predecessors$ and save in $R'_i.predecessors$. If R''_i is not empty, maintain it as a separate set and initiate $R''_i.predecessors$ to $R_i.predecessors$.
5. (Periodically) go through subsets R_i, R_j and if $R_i.predecessors$ and $R_j.predecessors$ are identical, merge them into one subset.

Figure 2: VS Synchronization.

Suppose that the last object for which an update is received is 'X'. Then all relevant updates on objects 'A' through 'X' have been received. That is, although some object, say 'B', had no new update during a particular synchronization, the receiver concludes, by observing the update on object 'X', that object 'B' is up-to-date. Hence, this creates a partition at the receiver into two ranges. 'A' though 'X' are up-to-date, and 'Y' through 'Z' are not. This property continues to hold as partitions split and merge.

Consider two replicas X_r and X_s performing a synchronization. X_s has q objects with updates that X_r is missing. Let X_r have k partitions in its store, X_s have m partitions. First, X_r sends X_s a description of its knowledge. This comprises of a collection of pairs $(R.list, R.predecessors)$, each describing the versions in one partition, R. Each partition description contains one version vector, and two values that bound the range of objects in the partition. In total, X_r sends X_s $\widetilde{O}(k \times R)$ bits. In response, X_s sends q objects, and up to m object set descriptors of the partitions that contain them. The total communication complexity here is $\widetilde{O}(q + m \times R)$ bits. Both X_r and X_s send at most $\widetilde{O}(N \times R)$ overhead bits.

In storage, a replica stores with each object its version, and with each partition an object set descriptor and a version vector.

5. CORRECTNESS

The following properties are maintained by our SVS and VS protocols.

Safety: Every conflicting version received by a requestor is detected.

Non-triviality: Only true conflicts are alerted.

Liveness: At the end of a complete execution of a synchronization procedure, for all objects, the requestor X_r stores versions that are identical, or that causally follow, the versions stored by source X_s .

In order to prove them, we make use of a key invariant which is stated and proven in the following lemma.

LEMMA 5.1. *For every vector set S, object $o \in S$, and version o.v, S.predecessors dominates v if and only if $v \prec$ o.version.*

PROOF. The initial vector set created by each replica contains all objects empty of updates and a zeroed *predecessors* vector. The invariant clearly holds then.

We now prove that the lemma continues to hold through all the transitions that a replica set goes through.

Consider a transition that occurs when a replica r generates a new version $o.v$. Replica r updates the vector set S that contains o, and merges the new version $o.v = o.version$ into $S.predecessors$. Version $o.v$ implicitly dominates preceding versions by r, possibly on different objects o'. Therefore, for any version $o'.w$ which is (implicitly) dominated by $o.v$, we must show that the lemma is not violated by merging $o.v$ into $S.predecessors$. This is indeed the case: If $o' \in S.list$, then since r itself knows every update it generated on any object, $o'.w$ must already be included in S.

The next transition to consider occurs during synchronization. A replica r integrates a vector set R it receives with vector set S it previously stored. This results in splitting S into two vector sets, $S_1 = S \cap R$ and $S_2 = S \setminus R$. The set S_2 has $S_2.predecessors$ inherited from $S.predecessors$, and contains a subset of S's objects. Hence, the predecessors invariant continues to hold for it. (S_2 might be empty, in which case the discussion is vacuous.) We now consider the situation regarding S_1. Let o be an object in the subset S_1. S_1 contains the most updated version(s) of o found in either R or S. Multiple versions of o exist in case of conflicts. A version v of o which has been incorporated in S_1 from R is dominated by $R.predecessors$. Hence, $S_1.predecessors$ dominates it as well. Likewise, a version v which is incorporated from S is dominated by $S.predecessors$. In this case, too, $S_1.predecessors$ dominates v. Finally, to prove that the converse hold, consider some version of o satisfying $o.w \prec S_1.predecessors$. Then either $o.w \prec S.predecessors$ or $o.w \prec R.predecessors$. The proof is identical in either case. In the former (latter) case, $o.w$ is dominated by a version of o stored in S (R), say, $o.v$. Therefore, S_1 must store either $o.v$, or $o.v'$ where $o.v \prec o.v'$. Hence, $o.v'$ dominates $o.w$. Note, however, that in case of conflicts, there might be some version $o.t$ in S_1 that does not dominate $o.w$. This is the reason that it is crucial to link all conflicting versions of an object and send in one message during synchronization, so that $o.w \prec S_1.predecessors$ indeed guarantees that $o.v'$ exists in S_1.

Finally, another transition is incurred by merging vector sets. The pre-requisite for merging is that the predecessor vectors of two sets are identical. Clearly, merging in this case does not violate the invariant holding separately for each set. □

THEOREM 5.2. *The protocols above maintain Safety, Liveness, and Non-triviality, as defined above.*

PROOF. These properties easily follow from the fact (proven in Lemma 5.1) that vector sets predecessors accurately reflect the object versions that they contains.

More specifically, let X_r be a replica that requests updates from a source replica X_s. In order to prove Safety, suppose that X_r obtains a version $o.v$ that conflicts with a version $o.v'$ it previously stored. Denote by R the vector set at X_r that contains o. Denote by S the vector set that was received from X_s that contains $o.v$. By Lemma 5.1, $o.v' \prec R.predecessors$ and $o.v' \not\prec S.predecessors$, and similarly, $o.v \prec S.predecessors$, $o.v \not\prec R.predecessors$. Looking at the synchronization procedure, we see that an alert must be generated in this case.

Non-triviality holds for similar reasons. Suppose that X_r obtains a version $o.v$ that causally succeeds a version $o.v'$ it previously stored. Denote by S the vector set that was received from X_s that contains $o.v$. By Lemma 5.1, $o.v \prec S.predecessors$, hence $o.v' \prec o.v \prec S.predecessors$. The synchronization procedure therefore discards $o.v'$ in this case, and does not alert conflicts.

Finally, Liveness holds as follows. Let $o.v$ be a version that X_s stores. Denote by R the vector set at X_r that contains o. Denote by S the vector set that was received from X_s that contains $o.v$. If $o.v \prec R.predecessors$, then by Lemma 5.1, X_r stores either $o.v$ or a version of o that causally follows o. Liveness holds in this case. If, on the other hand, $o.v \not\prec R.predecessors$, then X_s must send $o.v$ during synchronization. By the synchronization code, X_r integrates $o.v$ into storage, and again, liveness holds. □

6. RELATED WORK

Weakly consistent, non-transparent replication is supported in several known systems. The common goal and vision in these is to alleviate the costs incurred by strongly consistent replication paradigms, and provide non-stop availability in weakly connected settings by allowing *read-anywhere/write-anywhere*.

Locus [8] has pioneered the vision that optimistic replication is useful when conflicts are rare, are quickly detected and are easily resolved. It has also introduced *version vectors* [1] as a compact logical clock that captures partial, causal ordering between updates and facilitates conflict detection. Locus stores a version vector with each data object, and communicates this information in order to detect conflicts. Follow-on systems to Locus, such as Ficus, Rumor, and Roam, utilize this same technique to detect conflicting file updates [2, 6]. The focus of this work is to reduce the number of vectors that are stored and are exchanged for replica reconciliation. Coda [3] uses version vectors similar to LOCUS to support optimistic replication, although in Coda they are used in a client-server context, rather than peer-based replication.

In Bayou [7], every replica stores a log of all updates to the data known to it, as well as ordering information among the updates. When two replicas synchronize, they exchange any updates unknown to each other and detect conflicts. The main emphasis in Bayou centers around a mechanism for automatic semantic-based conflict detection and resolution. Additional effort is dedicated in Bayou to form eventual agreement on a total order in which updates are applied from the log. Bayou clients do not require to wait for conflict resolution or for update ordering to converge. A client's update is submitted to one of the replicas, and is considered *tentative*. It may be applied locally and the result can be read by clients right away. However, a tentative update may be rolled back later and re-executed at a different position in the total order. Bayou uses a primary base commit protocol in order to stabilize the order of updates. Overall, the issues of semantic conflict resolution, as well as total ordering of updates, which are central to Bayou, are orthogonal to the topic of discussion in this paper. However, an aspect of the Bayou storage management is of relevance. The storage infrastructure supports three logical partitions of the log, a tentative part, a *committed* or *stable* part, and a garbage-collected *omitted* part. Bayou introduced the idea of compactly representing each part using one vector (called the *O vector*, the *C vector*, and the *F vector*, respectively) containing the maximal update counter from each replica. This works in Bayou because each of the three log parts contains a complete prefix, and hence, it is possible to represent it using the maximal counter from each replica. The compact vector is used for detecting missing updates during replica synchronization, but not for conflict resolution. The idea of representing a set of version using compact vectors forms the basis of our vector sets method.

Our work reflects the replication management scheme of Microsoft's WinFS system. A description of the full WinFS architecture and design is provided in [5]. A preliminary write-up of the WinFS replication management has been provided by Malkhi et al. in [4]. The previous write-up concentrates on a bundling mechanism called PVE (*predecessors vectors with exceptions*), that groups together **all** objects and represents their state in one knowledge version vector. The current paper focuses on VS, which enhances PVE in a number of important ways. As in PVE, VS uses a single vector counts updates made by the replicas on all data objects. VS differs from PVE in its handling of disrupted synchronization. PVE handles disrupted synchronization by introducing *exception versions*, whose count is unbounded. This is the source of the theoretically unbounded overhead in PVE. Thus, whereas PVE has an unbounded worst case complexity, VS has a worst-case cost no greater than version vectors, since in the extreme, every object holds one independent vector. This dramatically improves the complexity under settings of difficult connectivity. Furthermore, in most normal situation, SV will perform as well as PVE, which is much better. We note that efficiency comes at no added complication. To the contrary, VS is simple and elegant, and avoids many of the complexities associated with exception versions. Finally, the VS mechanism has potentially wide applicability as a mechanism for compactly handling synchronization of arbitrarily overlapping groups of objects.

Acknowledgements
The protocol described in this paper closely reflects the replication method designed and used by the Microsoft WinFS product team, which included one of the authors (Lev Novik). Yunxin Wu is a member of WinFS who contributed greatly to this work. We are thankful to Doug Terry and Rama Ramasubramanian for helpful discussions.

7. REFERENCES

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

[2] T. W. Page (Jr.), R. G.. Guy, J. S. Heidemann, D. H. Ratner, P. L. Reiher, A. Goel, G. H. Kuenning, and G. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software – Practice and Experience*, 11(1), December 1997.

[3] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst.*, 10(1):3–25, 1992.

[4] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *The 19th Intl. Symposium on Distributed Computing (DISC)*, pages 339–353, September 2005.

[5] L. Novik, I. Hudis, D. B. Terry, S. Anand, V. J. Jhaveri, A. Shah, and Y. Wu. Peer-to-peer replication in winfs. Technical Report MSR-TR-2006-78, Microsoft, June 2006.

[6] D. H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing.* PhD thesis, 1998. UCLA Technical report UCLA-CSD-970044.

[7] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings 15th Symposium on Operating Systems Principles (SOSP)*, pages 172–183, December 1995.

[8] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The locus distributed operating system. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, pages 49–70, New York, NY, USA, 1983. ACM Press.