Concise Version Vectors in WinFS

Dahlia Malkhi[1] and Doug Terry[2]

[1] Microsoft Research Silicon Valley and
The Hebrew University of Jerusalem, Israel
[2] Microsoft Research Silicon Valley

Abstract. Conflicts naturally arise in optimistically replicated systems. The common way to detect update conflicts is via version vectors, whose storage and communication overhead are *number of replicas* $\times$ *number of objects*. These costs may be prohibitive for large systems.
This paper presents *predecessor vectors with exceptions* (PVEs), a novel optimistic replication technique developed for Microsoft's WinFS system. The paper contains a systematic study of PVE's performance gains over traditional schemes. The results demonstrate a dramatic reduction of storage and communication overhead in normal scenarios, during which communication disruptions are infrequent. Moreover, they identify a cross-over threshold in communication failure-rate, beyond which PVEs loses efficiency compared with traditional schemes.

1 Introduction

Consider an information system, such as an e-mail client, that is composed of multiple data objects, holding folders, files and tags. Data may be replicated in multiple sites. For example, a user's mailbox may reside at the server, on the user's office and home workstations, and on a PDA. The system allows concurrent, optimistic updates to its objects from distributed locations, without communication or centralized control. So for example, the user might hop on the plane with a copy of her mailbox on a laptop and edit various parts of it while disconnected; she may introduce changes on a PDA, and so on. At some point, when connecting between these components, she wishes to synchronize versions across replicas, and be alerted to any conflicts generated.

This problem model arises naturally within the scope of Microsoft's WinFS project, whose aim is to provide peer-to-peer weakly consistent replicated storage facilities. The problem model is fundamental in distributed systems, and numerous replication methods exist to tackle it. However, the applications that are aimed for by the WinFS team mandate taking scale more seriously than ever before. In particular, e-mail repositories, log files, and 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.

In this paper, we present a precise description and correctness proof of the replica reconciliation and conflict detection mechanism inside Microsoft's WinFS. We name the scheme *predecessor vectors with exceptions* (PVE). We

produce a systematic study of the performance gains of PVE, and provide a comparison with traditional optimistic replication scheme. The results demonstrate a substantial reduction in storage and communication overhead associated with replica synchronization, in most normal cases. In conditions that allow (most) synchronizations to complete without communication breaks, a pair of replicas needs only communicate a constant number of bits per replica in order to detect discrepancies in replicas' states. Moreover, they need to maintain only a single counter per object in order to determine versions ordering and alert to any conflict. Our study also demonstrates the "cut-off" point in the communication fault-rate, beyond which the PVE technique becomes less attractive than the alternatives.

In order to understand the efficiency leap offered by the PVE scheme, let us review the most well known alternative. Version Vectors (VVs) [1] are traditionally used in optimistic replication systems in order to find which replica has more updated object states, as well as to detect conflicting versions. Per object version vectors were pioneered in Locus [1], and subsequently employed in various optimistic replication systems, *e.g.*, [2,4,3].

The version vector for a data object is an array of size R, where R is the number of replicas in the system. Each replica has a pair $\langle \text{replica}, \text{counter} \rangle$ in the vector, indicating the latest counter value introduced on the object by the replica. For example, suppose that we have three replicas, A, B, and C. An object is initialized with VV $(\langle A, 0\rangle, \langle B, 0\rangle, \langle C, 0\rangle)$. An update to the object initiated at replica A increments A's component, and so generates version $(\langle A, 1\rangle, \langle B, 0\rangle, \langle C, 0\rangle)$. Later, B may obtain the new version from A and store it, and produce another update on the object. The newer object state receives version $(\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 0\rangle)$. And so on.

A version vector V *dominates* another vector W if every component of V is no less than W; V *strictly dominates* W, if it dominates W and one component is greater. Due to optimism, there may be objects on different replicas whose version vectors are incomparable by the domination relation; this corresponds to conflicting versions, indicating that simultaneous updates were introduced to the object at different replicas. For example, continuing the scenario above, suppose that all replicas have version $(\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 0\rangle)$ in store. Now proceed to have diverging updates on the object simultaneously by A and C. These generate VVs $(\langle A, 2\rangle, \langle B, 1\rangle, \langle C, 0\rangle)$ and $(\langle A, 1\rangle, \langle B, 1\rangle, \langle C, 1\rangle)$, respectively, which are conflicting as neither one dominates the other.

Consider a system with N objects replicated across R replicas. Further, consider the synchronization between two replicas whose views of the object space differs in q objects. The VV scheme is designed for synchronizing replicas object by object, and incurs the following costs.

1. Store a version-vector per object, incurring a storage overhead of $\widetilde{O}(N \times R)$ bits space; [3]

[3] For simplicity of notation, the notation $\widetilde{O}(\cdot)$ indicates the same complexity order as $O(\cdot)$ up to logarithmic factors of N and R, which may be required to code any single value in our settings.

2. Communicate information that allows the two replicas to determine which objects one should send the other, and to detect conflicts. A naive implementation sends all N version vectors, incurring a communication overhead of $\widetilde{O}(N \times R)$. If the replicas store logs of recent updates, and maintain additional information about the position known to other replicas in the log, they may bring the cost down close to $\widetilde{O}(q \times R)$, which is the lowest possible communication overhead with the VV scheme.

These cost measures and their analysis are made more precise later in the paper. Note that even for moderate numbers of replicas R, storing $N \times R$ values is a substantial burden when N is large, and moreover, communicating between $\widetilde{O}(q \times R)$ to $\widetilde{O}(N \times R)$ overhead bits may be prohibitive.

In WinFS, the goal is to quickly synchronize heavy-volume servers, each carrying large magnitudes of objects. In situations where communication distuptions are not the norm, the innovative PVE mechanism in WinFS that reconciles replica discrepancies brings down costs a considerable amount. It needs $\widetilde{O}(R)$ information bits to determine replica's differences, *i.e.*, the equivalence of communicating **one** version vector. In addition, per object meta-information storage and communication is in most cases constant (one counter). Table 1 in Section 5 contains a summary of these complexities. In the remainder of this paper, we describe the foundations of the PVE replication protocol, and compare is against VVs.

The contributions of this paper are as follows. First, we give a precise and detailed formulation of the PVE replica reconciliation protocol employed in WinFS. We note that the full design and the architecture of the WinFS system is the result of a large team effort, and is beyond the scope of this paper. Second, we develop a performance model capturing the cost measures of interest to us, and quantify the performance gains of the PVE scheme, as compared with known methods. Third, we evaluate these measures via simulation under complex system conditions with increasing communication failures rates. This evaluation reveals a cut-off point that characterizes the benefit area of the PVE scheme over traditional version vectors.

2 Problem Statement

In this section, we begin with the precise specification of our problem. Later sections provide a rigorous treatment of the solution.

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 hosts. Each host may locally introduce updates to every 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 non-related updates exist and are *conflicting*.

Our focus is on distributed systems in which updates overwrite previous versions. The alternative would be database or journal systems, in which the history of updates on an object is stored and applied at every replica. State-based storage saves storage and computation, and is suitable for the kind of

information systems that WinFS aims for, *e.g.*, a user's Outlook files, where updates may be numerous. In state-based systems only the most recent version of any object needs to be sent. Nevertheless, it is worth noting that the method presented in this paper can work with (minor) appropriate modifications for log-based replication systems. For brevity, we omit this from discussion in this paper.

The goal is to provide a lightweight 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 in them. With this mechanism, they can bring each other up-to-date or report conflicts.

More precisely, we now describe objects, versions, and causality. An object is identified uniquely by its name. Objects are instantiated with versions, where an object instance has the following fields:

name: the unique identifier.
version: a pair $\langle replica\ id, counter \rangle$.
predecessors: a set of preceding versions (including the current version).
data: application-specific opaque information.

Because versions uniquely determine objects' instances, we simply refer to any particular instance by its version. There may be multiple versions with the same object name. We say that these are versions of the same object.

There is a partial, causal ordering among different versions of the same object. When a replica A creates an instance of an object o with version v, the set $\mathcal{W}$ of versions that are previously known by replica A on o *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 might follow any single version. When two versions do not follow one another, they are *conflicting*. I.e., if $w \not\prec v \ \wedge \ v \not\prec w$, then v and w are conflicting.

It is desirable to detect and resolve conflicts, either automatically (when application specific conflict resolution code is available) or by alerting the user and solving 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, so we 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, and we are interested in all of them until they can be resolved.

2.1 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 missing by one that the other has, and (ii) any overhead data that is transferred along with objects' states in order to determine precedence/conflicts.

3 Overview of the PVE Method

This section provides an informal overview of the PVE scheme. Later sections provide a more formal description and a proof of correctness.

The PVE scheme works as follows. An object version is a pair $\langle \text{replica}, \text{counter} \rangle$. Instead of using separate counters for distinct objects, the scheme uses one per-replica counter to enumerate the versions that the replica generates on all objects (the counter is across all objects). For example, suppose that replica A first introduces an update to object o_1, and second to o_2. The versions corresponding to o_1 and to o_2 will be $\langle A, 1 \rangle$, $\langle A, 2 \rangle$, respectively. Note that versions are **not** full vectors, as in the traditional VV scheme described in the Introduction.

Each object has, in addition to its version, a predecessor set that captures the versions that causally precede the current one. Predecessor sets are captured in PVE using version vectors, though we will show momentarily that in most cases, PVE can replace these vectors with a null pointer. In order to distinguish these vectors from the traditional version vectors, we call them *predecessor vectors*. A predecessor vector (PV) contains one version, the latest, per replica. When a replica A generates a new object version, the PV associated with the new version contains the latest versions known by A on the object from each other replica. For example, suppose we have three replicas, A, B, and C. A new object starts with a zeroed predecessor vector $(\langle A, 0 \rangle, \langle B, 0 \rangle, \langle C, 0 \rangle)$. Consider the two versions generated by replica A above on o_1 and o_2, $\langle A, 1 \rangle$ and $\langle A, 2 \rangle$, respectively. When A creates these versions, no other versions are known on either o_1 or o_2, hence the PV of $\langle A, 1 \rangle$ is $(\langle A, 1 \rangle, \langle B, 0 \rangle, \langle C, 0 \rangle)$, and the PV of $\langle A, 2 \rangle$ is $(\langle A, 2 \rangle, \langle B, 0 \rangle, \langle C, 0 \rangle)$. A (causally) subsequent update to o_1 by replica B creates version $\langle B, 1 \rangle$, with predecessors $(\langle A, 1 \rangle, \langle B, 1 \rangle, \langle C, 0 \rangle)$. This PV represents the latest versions known by B on object o_1.

The formal definition capturing the per-replica scheme with predecessor vectors scheme are given below.

Definition 1 (Per-Replica Counter).
Let X be a replica. The versions *generated by replica X on objects are the ordered sequence $\{\langle X, i \rangle\}_{i=1,2,\ldots}$.*

Definition 2 (Predecessor Vectors).
Let $X_1, ..., X_R$ be the set of replicas. A predecessors vector *(PV) is an R-array of tuples of the form $(\langle X_1, i_1 \rangle, ..., \langle X_R, i_R \rangle)$.*

A predecessors 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 predecessor 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.*

The reader should first note that despite the aggregation of multiple-object versions using one counter, the predecessor version vectors can express precedence relations between versions of the same object. For example, in the scenario above, version $\langle A, 1\rangle$ precedes $\langle B, 1\rangle$, and is indeed dominated by the PV associated with version $\langle B, 1\rangle$. Moreover, PVs do not create false conflicts. The reason is that incomparable predecessor vectors conflict only if they belong to the same object. So for example, suppose that continuing the scenario above, replica A introduces version $\langle A, 3\rangle$ to object o_1 with predecessors $(\langle A, 3\rangle, \langle B, 1\rangle, \langle C, 0\rangle)$; and simultaneously, replica C introduces version $\langle C, 1\rangle$ on o_2, with the corresponding PV $(\langle A, 2\rangle, \langle B, 0\rangle, \langle C, 1\rangle)$. These versions would be conflicting had they belonged to the same object, but are fine since they are never compared against each other.

Hence, comparing different versions for the same object is now possible as in the traditional use of version vectors. Namely, the same domination relation among predecessor vectors and versions, though these may contain replica-counters pertaining to different objects, can determine precedence and conflicts of updates to the same object.

Reducing the Overhead. So far we have not introduced any space savings over traditional VVs, though. The surprising benefit of aggregate PVs is as follows. Let $X.knowledge$ denote the component-wise maximum of the PVs of all the versions held by a replica X. The performance savings stems from the following fact: In order to represent ordering relations of **all** the versions X stores for all objects, it suffices for replica X to store only $X.knowledge$. Knowledge aggregates the predecessor vectors of all objects, and is used instead of per-object PV. More specifically, knowledge replaces PVs as follows.

- No PV is stored per object at all. The only vector stored by a replica is its aggregate *knowledge* vector.
- In order for A to determine which versions in its store are more up-to-date than B's store, B simply needs to send $B.knowledge$ to A. Using the difference between $A.knowledge$ and $B.knowledge$, A can determine which versions it should send B.
- Then, having determined the q relevant newer versions, A sends these versions with (only) a single version counter each, plus to send (once) A's *knowledge* vector.

The reader should be concerned at this point that information is lost concerning the ability to tell version precedence. We now demonstrate why this is not the case. When two replicas, A and B, wish to compare their latest versions of the same object o, say $\langle r, n_r\rangle$ and $\langle s, n_s\rangle$ respectively, they simply compare these

against $A.knowledge$, $B.knowledge$. If $A.knowledge$ dominates $\langle s, n_s \rangle$, then the version currently held by A for object o, namely $\langle r, n_r \rangle$, strictly succeeds $\langle s, n_s \rangle$. And vice versa. If none of these knowledge vectors dominates the other version, then these are conflicting versions.

Going back to the scenario built above, replica A has in store the following: $o_1.version = \langle A, 3 \rangle$; $o_2.version = \langle A, 2 \rangle$; $knowledge = (\langle A, 3 \rangle, \langle B, 1 \rangle, \langle C, 0 \rangle)$. Replica C stores the following: $o_1.version = \langle B, 1 \rangle$; $o_2.version = \langle C, 1 \rangle$; $knowledge = (\langle A, 2 \rangle, \langle B, 1 \rangle, \langle C, 1 \rangle)$. When comparing their versions for object o_1, A and C will find that A's version is more recent, and when comparing their versions of object o_2, they will find C's version to be the recent one.

The result is that storage overhead in WinFS is $\widetilde{O}(N + R)$, instead of $\widetilde{O}(N \times R)$. More dramatically, the communication overhead associated with synchronization is reduced. The communication overhead of sending $knowledge$ is $\widetilde{O}(R)$, and the total communication overhead associated with synchronizing replicas is $\widetilde{O}(q + R)$.

Dealing with Disrupted Synchronization. Synchronization among two replicas may fail to complete due to network disruption. One way of coping with this is to abort incomplete synchronization procedures; then no further complication to the above scheme is needed.

However, in reality, due to large volumes that may need to be synchronized, aborting a partially-completed synchronization may not be desirable (and in fact, may create increasingly larger and larger synchronization demands, that might become less and less likely to complete). The aggregate knowledge method above introduces a new source of difficulty due to incomplete synchronizations. Let us demonstrate this problem. When replica A receives an object's new version from another replica B, that object does not carry a specific PV. Suppose that before synchronizing with B, the highest version A stores from B on any object is $\langle B, 10 \rangle$. If B sends $\langle B, 14 \rangle$, then clearly versions $\langle B, 11 \rangle$, $\langle B, 12 \rangle$, and $\langle B, 13 \rangle$ are missing in A's knowledge, hence there are "holes".

It is tempting to try to solve this by a policy that mandates sending all versions from one replica in an order that respects their generation order. In the above scenario, send $\langle B, 11 \rangle$ before $\langle B, 14 \rangle$, unless that version has been obsoleted by another version. Then, when $\langle B, 14 \rangle$ is received, A would know that it must already reflect $\langle B, 11 \rangle$, $\langle B, 12 \rangle$, and $\langle B, 13 \rangle$.

Unfortunately, this strategy is impossible to enforce, as illustrated in the following scenario. Object o_1 receives an update from replica A, with version $\langle A, 1 \rangle$, and PV $(\langle A, 1 \rangle, \langle B, 0 \rangle, \langle C, 0 \rangle)$. Meanwhile, object o_2 is updated by B, its version is $\langle B, 1 \rangle$, with PV $(\langle A, 0 \rangle, \langle B, 1 \rangle, \langle C, 0 \rangle)$. Replica A and B synchronize and exchange their latest updates. Subsequently, object o_1 is updated at replica B with version $\langle B, 2 \rangle$ and PV $(\langle A, 1 \rangle, \langle B, 2 \rangle, \langle C, 0 \rangle)$; and object o_2 is updated at replica A with version $\langle A, 2 \rangle$ and PV $(\langle A, 2 \rangle, \langle B, 1 \rangle, \langle C, 0 \rangle)$. The orderings between all versions is as follows:

$$o_1 : [\langle A, 1 \rangle; \mathrm{PV} = (\langle A, 1 \rangle, \langle B, 0 \rangle, \langle C, 0 \rangle)] \prec [\langle B, 2 \rangle; \mathrm{PV} = (\langle A, 1 \rangle, \langle B, 2 \rangle, \langle C, 0 \rangle)]$$
$$o_2 : [\langle B, 1 \rangle; \mathrm{PV} = (\langle A, 0 \rangle, \langle B, 1 \rangle, \langle C, 0 \rangle)] \prec [\langle A, 2 \rangle; \mathrm{PV} = (\langle A, 2 \rangle, \langle B, 1 \rangle, \langle C, 0 \rangle)]$$

Then Replica B synchronizes with replica A, sending it all of its recent updates. Replica A now stores: $o_1.version = \langle B, 2\rangle$; $o_2.version = \langle A, 2\rangle$; $knowledge = (\langle A, 2\rangle, \langle B, 2\rangle, \langle C, 0\rangle)$.

Now suppose that replica C, which has been detached for a while, comes back and synchronizes with replica A. During this synchronization, only the most recent versions of objects o_1 and o_2 are sent to replica C. In this scenario, there is simply no way to prevent holes: Replica C may first obtain o_1's recent version, i.e., $\langle B, 2\rangle$, and then have its communication cut. Then version $\langle B, 1\rangle$ (which happens to belong to o_2) is missing. A similar situation occurs if replica C obtains o_2's recent version first and is then cut.

It is worth noting that although seemingly we don't care about the missing, obsoleted versions, we cannot ignore them. If the subsequent versions are lost from the system for some reason, inconsistency may result. For example, in the first case above, the missing o_2 version $\langle B, 1\rangle$ is subsumed by a later version $\langle A, 2\rangle$. However, if replica C simply includes $\langle B, 2\rangle$ in its knowledge vector, and replica A crashes such that $\langle A, 2\rangle$ is forever lost from the system, C might never obtain the latest state of o_2 from replica B.

The price paid in the PVE scheme for its substantial storage and communication reduction is the need to maintain information about such exceptions. In the above scenario, replica C will need to store *exception* information as follows. First, $C.knowledge$ will contain $(\langle A, 0\rangle, \langle B, 2\rangle, \langle C, 0\rangle)$ with an *exception* $\langle eB, 1\rangle$. [4]

Definition 3 (PVs with Exceptions).

A predecessors vector with exceptions *(PVE) is an R-array of tuples of the form* $(\langle X_1, i_1\rangle\langle eX_1, i_{j_1}\rangle\langle eX_1, i_{j_{k_1}}\rangle$, ..., $\langle X_R, i_R\rangle\langle eX_R, i_{j_R}\rangle\langle eX_1, i_{j_{k_R}}\rangle)$.

A version $\langle X_k, j_k\rangle$ *is dominated by a predecessors vector* X *with exceptions as above if* $i_k \geq j_k$, *and* j_k *is not among the exceptions in the* k*'th position in* X.

A predecessors vector with exceptions X *dominates another vector* Y *if the respective PVs without the exceptions dominate, and no exception included* X *is dominated by* Y.

Second, we require that a replica maintain explicit PV for every new version it obtains via a partial synchronization. These explicit PVs may be omitted only if the replica's knowledge dominates them. Continuing the scenario above, we demonstrate a subtle chain of events which necessitates this additional overhead.

Consider the information stored at replica C after partial synchronization: $o_1.version = \langle B, 2\rangle$; $o_2.version = \perp$; $knowledge = (\langle A, 0\rangle, \langle B, 2\rangle\langle eB, 1\rangle, \langle C, 0\rangle)$. Suppose that A synchronizes with C and sends it update $\langle A, 1\rangle$ on o_1. This update clearly does not follow $\langle B, 2\rangle$ (the current version of o_1 held by C),

[4] An alternative form of exception is to store $(\langle A, 0\rangle, \langle B, 0\rangle, \langle C, 0\rangle)$ with a 'positive exception' $\langle eB, 2\rangle$. The two alternatives result in different storage load under different scenarios, positive exceptions being preferable under long synchronization gaps. For simplicity, we use negative exceptions in the description here, although the method employed in WinFS uses positive exceptions.

but according to C's knowledge, neither is it succeeded by it – a conflict! The problem, of course, is that C's knowledge no longer dominates version $\langle A, 1\rangle$.

Only at the end of the synchronization procedure, the knowledge of the sending replica is merged with the knowledge of the receiving replica. At that point, knowledge at the receiving replica will clearly dominate all of the versions it received during synchronization, and their PV may be omitted. But if synchronization is cut in the middle, some of these PVs must be kept, until such time when the replica's knowledge again dominates them.

In our performance analysis and comparison with other methods, we take into account this cost and measure its effect. Note that, it is incurred only due to communication disruptions that prevent synchronization procedures from completing. Our simulations vary the number of such disruptions from small to aggressively high.

4 Causality-Based Replica Reconciliation

In this section, we begin to provide the formal treatment of the PVE replica reconciliation mechanism. Our approach builds the description in two steps. First, we give a generic set-oriented method for replica reconciliation, and define the properties it requires. Second, in the next section, we instantiate the method with the PVE concise predecessor vectors scheme.

The key enabler of replica synchronization is a mechanism for representing sets of versions, through which precedence ordering can be captured. To this end, replicas store the following information concerning causality. First, replica r maintains information about the entire set of versions it knows of, represented in $r.knowledge$. Second, each version v stored at replica r contains in $v.predecessors$ a representation of the entire set of causally preceding versions. More specifically, we require the maintenance of a set $r.knowledge$ per replica r, and $v.predecessors$ per version v, as follows.

Definition 4 (The Knowledge Invariant). *For every replica r, and version v, we require r to maintain a set $r.knowledge$, such that if $v \in r.knowledge$ then replica r stores version v or a version w such that $v \prec w$.*

Definition 5 (The Predecessors Invariant). *For every object instances v and w, we require r to maintain a set $w.predecessors$ such that $v \in w.predecessors$ if and only if $v \prec w$.*

Given the above two requirements, it should be possible to determine if a version is included in a replica's storage; and if one version precedes another or they conflict.

4.1 A Synchronization Framework

We now give a two-way asymmetric, conflict detection framework that uses *knowledge* and *predecessors*. The protocol is composed of a requestor that contacts a server, and obtains all the versions in the server's knowledge. These versions are integrated into the requestor's storage, and raise conflict alarms where needed.

The synchronization protocol is a one-way protocol between a requesting host and a serving host. It makes use of the conflict causality representation as follows:

1. Requestor r sends server s its knowledge set $r.knowledge$.
2. Server s responds with the following:
   (a) For every object o it stores, for which $o.version \notin r.knowledge$, it sends o
3. For every version o received by from s, requestor r does the following:
   (a) For every object w in store, such that $w.name == o.name$:
   if $o \in w.predecessors$ then ignore o and stop;
   else if $w.version \in o.predecessors$ then delete w;
   else alert conflict.
   (b) Insert $o.version$ into $r.knowledge$.
   (c) Integrate $o.predecessors$ into $r.knowledge$.
   (d) store o

Fig. 1. A generic framework for using causality information.

However, for our purposes, representing the full *knowledge* and *predecessors* sets is too costly. The challenge is to represent causality in a space-efficient manner, suitable for very large object sets, and moderate-size replica sets, while maintaining the invariants. The detailed solution follows in the next section.

4.2 Concise Version Vectors

The key to our novel conflict-detection technique is to transform the *predecessors* sets into different sets that can be represented more efficiently.

We first require the following technical definition:

Definition 6 (Extrinsic). *Let o be some object, $o.predecessors$ its predecessor set. Let S be a set of versions. We denote by $S\,|_{o.name}$ the reduction of S to versions pertaining to object $o.name$ only. S is called* extrinsic *to o if $S\,|_{o.name} == o.predecessors$.*

The surprising storage saving is derived in PVE from the following realization. For any object o, we can use an extrinsic set to o in place of *predecessors* throughout the protocol. In particular, when a replica's knowledge set is extrinsic to a predecessors set, it may be used in its place; the main storage savings is derived from using an empty set to denote (by convention) the replica's knowledge set, and avoid repeated storage of it. The following rule is the root of the PVE storage and communication savings:

Property 1. At any point in the protocol, any *predecessors* set may be replaced with an extrinsic set. By convention, an empty *predecessors* set indicates the replica's *knowledge* set.

We are now ready to introduce the PVE novel conflict detection scheme, which considerably reduces the size of representations of predecessor versions in normal cases.

Versions and Predecessor Vectors. The scheme uses the per-replica counter defined in Definition 1, that enumerates updates generated by the replica on all objects. Hence, a replica r maintains a local counter c. When replica r generates a version on an object o, it increments the local counter and creates version $\langle r, c\rangle$ on object o. Predecessors are represented using the PVEs as defined in Definition 3.

Knowledge. A replica r maintains in $r.knowledge$ a PVE representing all the versions it knows of. Inserting a new version $\langle s, n_s\rangle$ into $r.knowledge$ is done by updating the highest version seen by s to $\langle s, n_s\rangle$, and possibly inserting exceptions if there are holes between n_s and the previous highest version from s.

Object Predecessors. As already mentioned, an empty ($\perp$) *predecessors* set is used whenever $r.knowledge$ is extrinsic to an object's predecessors. In all other cases, *predecessor* contains a PVE, describing the set of causally preceding versions on the object.

Generating a New Version. When a replica r generates a new update on an object o, the new version $\langle r, c\rangle$ is inserted into $r.knowledge$ right away. Then, if $o.predecessor$ is $\perp$, nothing needs to be done to it. Implicitly, this means that the versions dominated by $r.knowledge$ causally precede the new version. If $o.predecessors$ is not empty, then the new version is inserted to $o.predecessors$ without exceptions. Implicitly this means that the set of versions that were dominated by the previous $o.predecessors$ causally precede the new update.

Merging Knowledge. During synchronization, the knowledge vector of a sender s is merged into the knowledge vector of the receiver r. The goal of the merging is to produce a vector that represents a union of all the versions included in $r.knowledge$ and $s.knowledge$, and replace $r.knowledge$ with it. For example, merging $s.knowledge = (\langle A, 3\rangle, \langle B, 5\rangle\langle eB, 4\rangle, \langle C, 6\rangle)$ into $r.knowledge = (\langle A, 7\rangle\langle eA, 6\rangle, \langle B, 3\rangle\langle eB, 2\rangle, \langle C, 1\rangle)$ yields $(\langle A, 7\rangle\langle eA, 6\rangle, \langle B, 5\rangle\langle eB, 4\rangle, \langle C, 6\rangle)$.

Synchronization. Space saving using empty predecessors requires caution in maintaining the extrinsic nature of predecessor sets throughout the synchronization protocol.

First, suppose that a requestor r receives from a server s a version v with an extrinsic $v.predecessors$ set. It is incorrect to merge $v.predecessors$ into the $r.knowledge$ set right away, since $v.predecessors$ may contain versions of objects different from v that r does not have. Hence, only v itself can be inserted into $r.knowledge$.

Second, consider the state of $r.knowledge$ at the end of its synchronization with s. Every version v sent by s has been inserted into $r.knowledge$. However, there may be some versions, *e.g.*, $w \prec v$, which $r.knowledge$ does not contain. s does not explicitly send w, because it is included in $v.predecessors$. But since predecessor sets are not merged into $r.knowledge$, it may be left not containing w. To address this, at the end of an uninterrupted synchronization with s, the requestor r merges $s.knowledge$ into $r.knowledge$.

Third, should synchronization ever be disrupted in the middle, a requestor r may be left with $r.knowledge$ lacking some versions. This happens if a version v

was incorporated into $r.knowledge$, but some preceding version $w \prec v$ has not been merged in.

As a consequence, in a future synchronization request, say with s', r may (inefficiently) receive w from s'. Hence, r checks if it can discard w by testing whether w is contained in $v.predecessors$ (and if yes, r also inserts w into $r.knowledge$ for efficiency). Figure 2 below describes the full PVE synchronization protocol.

1. Requestor r sends server s its knowledge set $r.knowledge$.
2. Server s responds with the following:
   (a) **It sends** $s.knowledge$**.**
   (b) For every object o it stores, for which $o.version \notin r.knowledge$, it sends o. If $s.knowledge$ is not extrinsic to $o.predecessors$, s sends $o.predecessors$ (otherwise, leave $o.predecessors$ empty).
3. For every version o received from s, requestor r does the following:
   (a) For every object w in store, such that $w.name == o.name$:
   if $o.version \in w.predecessors$ **or** $w.predecessors == \perp$ **and** $o.version \in r.knowledge$ then ignore o and stop;
   else if $w.version \in o.predecessors$ **or** $o.predecessors == \perp$ **and** $w.version \in s.knowledge$ then delete w;
   else alert conflict.
   (b) store o
   (c) **If** $o.predecessors == \perp$**, then unless** $r.knowledge$ **is extrinsic to** $s.knowledge$ **set** $o.predecessors = s.knowledge$**.**
   (d) **For every object** w **in store, such that** $w.name == o.name$ **(these must be conflicting versions), if** $w.predecessors = \perp$ **then set** $w.predecessors = r.knowledge$**.**
   (e) Insert $o.version$ into $r.knowledge$.
4. **Merge** $s.knowledge$ **into** $r.knowledge$**.**
5. **(Lazily) go through versions** v **such that** $v.predecessors \neq \perp$**, and if** $r.knowledge$ **is extrinsic to** $v.predecessors$ **then set** $v.predecessors = \perp$**.**

Fig. 2. Using extrinsic predecessors; modifications from the generic framework indicated in boldface.

4.3 Properties

The following properties are easily derived from the two invariants given in Definition 4 and Definition 5. In the full paper, we provide proof that the protocol above maintains these invariants.

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 r stores versions that are identical, or that causally follow, the versions stored by server s .

5 Performance

Storage overhead associated with precedence and conflict detection comprises of two components. The per replica *knowledge* vector contains aggregate information about all known versions at the replica. In typical, faultless scenarios, the PVE scheme requires $\widetilde{O}(R)$ space per replica for the *knowledge* representation. By comparison, the VV scheme has no aggregate information on a replica's knowledge.

Additional storage overhead stems from precedence vectors. In our scheme, in faultless scenarios there is one version counter per object, incurring a space of $\widetilde{O}(N)$. By comparison, in faultless scenarios, the VV scheme keeps $\widetilde{O}(R \times N)$ storage, i.e., one version vector per object.

The fault-free (lower-bound) storage overhead for PVE and VV are summarized in Table 1.

When failures occur, the overhead of VV remains unchanged, but the PVE scheme may gradually suffer increasing storage overheads. There are two sources of additional complexity. The first is the need to keep exceptions in the knowledge, the second is the explicit version vectors (and their corresponding exceptions) kept for versions which the replica's knowledge does not dominate. In theory, neither of these components has any strict upper bound. These formal upper bounds are also summarize in Table 1 below. Below we provide simulation results that demonstrate storage growth in the PVE scheme relative to failure rates.

The communication overhead associated with synchronization also has two parts. First, a sender and a receiver need to determine which objects have versions yet unknown to the receiver. In the PVE scheme, this is done by conveying the receiver's knowledge vector to the sender. The faultless overhead here is $\widetilde{O}(R)$; the upper bound is again theoretically unbounded.

Denote from now on the number of object versions that the sender determines it has to send to the receiver by q. The second component of the communication overhead is the extra precedence information associated with these q objects. In faultless runs of the PVE scheme, this information consists of one counter (the version counter) per object. Hence, the overhead is $\widetilde{O}(q)$. In case of faults, as before the knowledge vector might contain an unbounded number of exceptions, and additionally, some objects may have explicit version vectors (and their exceptions) associated with them. Hence, there is no formal upper bound on the synchronization overhead. Here again, our simulation studies relate this complexity with the fault rate.

As for the VV scheme, the only way to convey knowledge of the latest versions held by a replica is by explicitly listing all of them, which requires $\widetilde{O}(N \times R)$ bits. Therefore, in realistic deployments of VV, the server may keep a log of version vectors of all the objects that received updates since the last synchronization with the requestor, and sends only the VVs associated with these objects. The complexity will be between $\widetilde{O}(q \times R)$ and $\widetilde{O}(N \times R)$.

In face of communication faults, replicas using the PVE method might accumulate over time both knowledge exceptions, and versions that require explicit

	Version vectors	PVE
storage l.b.	$\widetilde{O}(N \times R)$	$\widetilde{O}(N + R)$
storage u.b	$\widetilde{O}(N \times R)$	unbounded
comm l.b.	$\widetilde{O}(q \times R)$	$\widetilde{O}(q + R)$
comm u.b.	$\widetilde{O}(N \times R)$	unbounded

Table 1. *Lower and Upper Bounds Comparison of PVE with the version-vector scheme.*

predecessors. There is no simple formula that describes how frequently are exceptions accrued, as this depends on a variety of parameters and exact causal ordering.

In order to evaluate the effect of communication disruptions on storage in our scheme, we conducted several simple simulations. We ran $R = 50$ replicas, generating version updates to objects at random. The number of objects varied between $N = 100$ and $N = 1000$. Every 100 total updates, a synchronization round was carried out in a round-robin manner: Replica 1 served updates to 2, replica 2 served 3, and so on, up to replica R sending updates back to 1. This was repeated 100 times. A failure-probability variable $pfail$ controlled the chances of a communication disruption within every pairwise synchronization. We measured the resulting average communication and storage overhead. These are depicted in Figure 3 for three cases, 100, 1000 and 10000 objects. We normalize the overhead to per-object overhead. For reference, the per object storage overhead in standard VVs is exactly $R = 50$. The best achievable communication overhead with VVs is also $R = 50$, and is depicted for reference.

The figure clearly indicates a tradeoff in the PVE scheme. When communication disruptions are reasonably low, PVE storage and communication overhead is substantially reduced compared with the VV scheme, even for a relatively small number of objects. As failure rate increases, the number of exceptions in aggregate vector rises, and the total storage used for knowledge and for predecessor sets increases. The point at which the per-object amortized overhead passes that of a single VV depends on the number of objects, and for quite moderate size systems (1K objects) the cut-off point is beyond 90 percent communication disruption rate.

Acknowledgements

The protocol described in this paper was designed by the the Microsoft WinFS product team, including Doug Terry. We especially acknowledge Irena Hudis and Lev Novik for pushing the idea of concise version vectors. Harry Li, Yuan Yu, and Leslie Lamport helped with the formal specification of the replication protocol and the proof of its correctness.

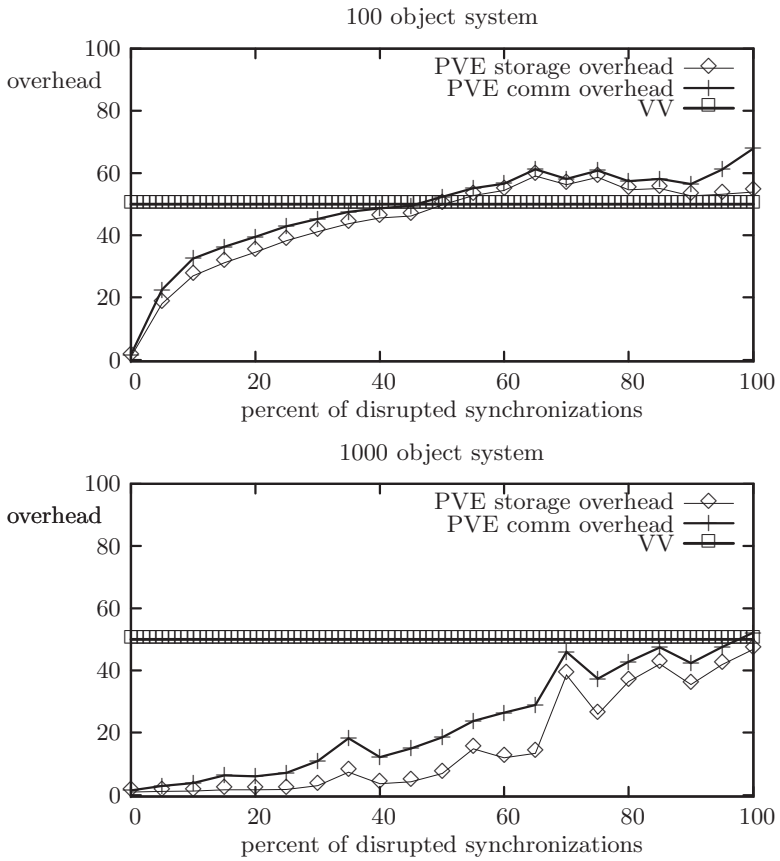

Fig. 3. Per-object storage and communication overheads with varying communication failure frequency, with $N = 100$ objects (top) $N = 1000$ objects (bottom).

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. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992.
4. D. H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, 1998. UCLA Technical report UCLA-CSD-970044.