(will be inserted by the editor)

Dahlia Malkhi · Doug Terry

Concise Version Vectors in WinFS

Received: date / Accepted: date

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* × *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 a server, on the user's office and home PCs, on the user's laptop, and on a PDA. The system allows concurrent, optimistic updates to its objects from distributed locations without communication or centralized control. 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 these computers are connected, she wishes to synchronize object versions across replicas and be alerted to any conflicts.

This system model arises naturally within the scope of Microsoft's WinFS platform, which was designed to provide peer-to-peer weakly consistent replicated storage facilities. The model is fundamental in distributed systems, and numerous replication methods exist to support it. However, 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.

In this paper, we present a precise description and correctness proof of the replica reconciliation and conflict detection mechanism inside Microsoft's WinFS system. 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 the traditional optimistic replication scheme. The results demonstrate a substantial reduction in the storage and communication overhead associated with replica synchronization in most cases. In conditions that allow 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 their states. Moreover, they need to maintain only a single counter per object in order to determine the causal ordering of objects' versions and detect any conflicting versions arising from concurrent updates. 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) [6] are traditionally used in optimistic replication sys-

D. Malkhi
Microsoft Research Silicon Valley
E-mail: dalia@microsoft.com

Doug Terry
Microsoft Research Silicon Valley

tems in order to decide which replica has the more up-to-date contents for each object, as well as to detect conflicting versions. Per object version vectors were pioneered in the Locus distributed file system [6], and subsequently employed in various optimistic replication systems [7,11,9].

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 number of modifications performed on the object by the replica. For example, suppose that we have three replicas, A, B, and C. An object is initialized with version vector $(\langle A,0\rangle, \langle B,0\rangle, \langle C,0\rangle)$. An update to the object initiated at replica A increments A's component, and so generates version vector $(\langle A,1\rangle, \langle B,0\rangle, \langle C,0\rangle)$. Later, B may obtain the updated object from A, along with its version vector, and produce another update on the object. The newer object state receives version vector $(\langle A,1\rangle, \langle B,1\rangle, \langle C,0\rangle)$. 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 of V is greater. Due to optimism, objects on different replicas may have version vectors that 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 vector $(\langle A,1\rangle, \langle B,1\rangle, \langle C,0\rangle)$ for a stored object. Now replicas A and C produce diverging updates on the object simultaneously. These updates generate version vectors $(\langle A,2\rangle, \langle B,1\rangle, \langle C,0\rangle)$ and $(\langle A,1\rangle, \langle B,1\rangle, \langle C,1\rangle)$, respectively, which are conflicting since neither one dominates the other.

Consider a system with N objects replicated across R replicas. Further, consider the synchronization between two replicas that have differing versions for 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; [1]
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 log entries that have previously been synchronized with other replicas, they may bring the cost down close to $\widetilde{O}(q \times R)$, which is the lowest possible communication overhead with the VV scheme.

[1] 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.

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

In WinFS, the goal is to quickly synchronize heavy-volume replicas, each carrying large magnitudes of objects. In situations where communication disruptions are not the norm, the innovative PVE mechanism in WinFS reduces storage and communication costs by a considerable amount. Replicas need only exchange $\widetilde{O}(R)$ information bits to determine their differences, *i.e.* the equivalent of communicating **a single** version vector. In addition, in most cases, per object meta-information storage and communication is constant (one counter). Table 1 in Section 6 contains a summary of these complexities. In the remainder of this paper, we describe the foundations of the PVE replication protocol and compare it against the traditional VV scheme.

The contributions of this paper are three-fold. First, we give a precise and detailed formulation of the PVE replica reconciliation protocol employed in WinFS. Second, we develop a performance model capturing the cost measures of interest and quantify the performance gains of the PVE scheme compared with known methods. Third, we evaluate the PVE scheme 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. We note that the full design and architecture of the WinFS replication platform is the result of a large team effort and is beyond the scope of this paper; we refer the interested reader to [10] for a wider coverage of the WinFS architecture.

The next section gives a detailed problem statement. Section 3 provides an informal overview of the PVE scheme. Section 4 provides a precise description of the method and lays the foundation for reasoning about its correctness. Correctness proofs are provided in Section 5. Section 6 contains a performance study. Section 7 surveys related work, and Section 8 concludes.

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 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. The alternative would be database or journal systems in which the history of updates on an object is logged and applied at every replica. State-based replication saves storage and computation and is suitable for the kind of information-intensive applications for which WinFS was designed, *e.g.* a user's Outlook files, where updates may be numerous. In state-based replication systems, only the most recent version of any object needs to be sent during synchronization. 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 while requiring only a small amount of per-object overhead. With this mechanism, replicas can bring each other up-to-date and 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 $\langle \mathit{replica\ id}, \mathit{counter} \rangle$.
- predecessors: a set of preceding versions (including the current version).
- data: opaque application-specific information.

Because versions uniquely determine objects' instances, we simply refer to any particular instance by its version. Over space and time, 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.* In other words, if $w \not\prec v \ \wedge \ v \not\prec w$, 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.

2.1 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 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.

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 ⟨replica, counter⟩. 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 PV $(\langle A,1\rangle, \langle B,1\rangle, \langle C,0\rangle)$. This predecessor vector represents the latest versions known by B on object o_1.

The formal definitions capturing the 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 *predecessor vector* (PV) is an R-array of tuples of the form $(\langle X_1,i_1\rangle,...,\langle X_R,i_R\rangle)$.

A predecessor 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, predecessor 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 predecessor vector $(\langle A,3\rangle, \langle B,1\rangle, \langle C,0\rangle)$; 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 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 version vectors. The surprising benefit of 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 PVs. 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 data objects 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.
– Having determined the q relevant newer objects, A sends these objects with (only) a single version each. In addition, A needs to send (once) its *knowledge* vector.

The reader may be concerned at this point that information is lost regarding the ability to determine 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 $B.knowledge$ and $A.knowledge$ respectively. 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 presented 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, producing version $\langle B, 1\rangle$ 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:
$o_1 : [\langle A, 1\rangle; \text{PV} = (\langle A, 1\rangle, \langle B, 0\rangle, \langle C, 0\rangle)] \prec [\langle B, 2\rangle; \text{PV} = (\langle A, 1\rangle, \langle B, 2\rangle, \langle C, 0\rangle)]$
$o_2 : [\langle B, 1\rangle; \text{PV} = (\langle A, 0\rangle, \langle B, 1\rangle, \langle C, 0\rangle)] \prec [\langle A, 2\rangle; \text{PV} = (\langle A, 2\rangle, \langle B, 1\rangle, \langle C, 0\rangle)]$

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

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, $\langle B, 2\rangle$, 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 disconnected.

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, $C.\mathit{knowledge}$ will contain $(\langle A, 0\rangle, \langle B, 2\rangle, \langle C, 0\rangle)$ with an *exception* $\langle eB, 1\rangle$.[2]

Definition 3 (PVs with Exceptions)
A *predecessor 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 predecessor 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 predecessor 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 an 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: $o_1.\mathit{version} = \langle B, 2\rangle$; $o_2.\mathit{version} = \perp$; $\mathit{knowledge} = (\langle A, 0\rangle, \langle B, 2\rangle\langle eB, 1\rangle, \langle C, 0\rangle)$. Suppose that A synchronizes with C and sends it

[2] 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.

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), 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 invariants that it maintains. This protocol introduces the basic synchronization process and is simple enough that one can easily argue that it satisfies the stated correctness properties. Second, we instantiate the method with the PVE concise predecessor vectors developed for WinFS. This is an abstract version of the actual synchronization protocol used in WinFS. Its correctness is then proven in Section 5.

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 stored 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 $v \prec w$.

When a replica is first created, its *knowledge* set is empty. Local updates produce new versions that are added to the replica's knowledge, as are versions received from other replicas during synchronization.

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

When a new data object is created, its *predecessors* set contains only its own version. Each update to the object produces a new version that is added to the previous version's predecessors to get the predecessors set for the new instance of the object.

Given the above two invariants, it is 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

Figure 1 presents an asymmetric synchronization protocol that uses *knowledge* and *predecessors*. The protocol is a one-way protocol between a requesting replica and a source replica. The requestor contacts a source and obtains all the versions in the source replica's knowledge. These versions are integrated into the requestor's storage, and conflict alarms are raised where needed.

1. Requestor r sends source s its knowledge set $r.knowledge$.
2. Source 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 replica synchronization protocol using causality information.

This protocol clearly maintains the two desired invariants. However, for practical 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 and $o.predecessors$ its predecessor set. Let S be a set of versions. We denote by $S\mid_{o.name}$ the reduction of S to versions pertaining to object $o.name$ only. S is called *extrinsic* to o if $S\mid_{o.name} = o.predecessors$.

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 o, the knowledge may be used as the predecessors set for o. The main storage savings is derived from using an empty predecessors set for an object to denote (by convention) that the replica's knowledge set may be used instead. 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, which 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. 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.

Synchronization. Figure 2 below describes the full PVE synchronization protocol. 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 source s a version v with an extrinsic $v.predecessors$ set. Unlike the simplified protocol in Figure 1, 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 r's synchronization with s. Every version v sent by s has been inserted into $r.knowledge$. However, there may be some versions, *e.g.* $w \prec v$, that $r.knowledge$ does not contain. Source replica 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 $r.knowledge$. 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 $(\langle A,7\rangle\langle eA,6\rangle, \langle B,5\rangle\langle eB,4\rangle, \langle C,6\rangle)$.

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).

4.3 Properties

The following properties are maintained by our protocol, and are derived from the two invariants given

1. Requestor r sends source s its knowledge set $r.knowledge$.
2. Source 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, 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 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 then set $v.predecessors = \perp$.**

Fig. 2 Synchronization using extrinsic predecessors; modifications from the generic protocol are indicated in boldface.

in Definition 4 and Definition 5 (proofs are provided in the next section).

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

5 Correctness

This section provides a correctness proof for the protocols presented in Section 4.2.

Lemma 1 *The Knowledge Invariant of Definition 4 is maintained throughout the update generation and synchronization protocol.*

Proof Initially, $r.knowledge$ is empty, and so the invariant trivially holds. The set is updated in the following events.

– When replica r generates a new version $\langle r, n_r \rangle$, the version is inserted into $r.knowledge$. Clearly, this version is held in storage by r. Moreover, any preceding version $\langle r, n'_r \rangle$, $n'_r < n_r$, must also be stored in r, or has been obsoleted by another version on the same object. Hence, no exceptions are needed.
– During synchronization, when a new version for some object o arrives, that version is inserted into $r.knowledge$, along with any required exceptions. Clearly, since o is stored by r (or is obsoleted by some causally succeeding version of o), the knowledge invariant holds.
– At the end of synchronization, the knowledge vector $s.knowledge$ is merged into $r.knowledge$. The merging produces a vector that represents a union of $s.knowledge$ and $r.knowledge$. Since merging is done only at the end of a complete synchronization, the requestor r must already store all versions included in $s.knowledge$ or later ones. Hence, replacing $r.knowledge$ with a vector representing the union with $s.knowledge$ maintains the Knowledge Invariant.

Lemma 2 *Let o and w be two versions of an object. Let S_o be an extrinsic set to o, and S_w to w. Then the Predecessors Invariant of Definition 5 is maintained if we use S_o in place of $o.predecessors$ and S_w in place of $w.predecessors$.*

Proof If $o \prec w$, then by definition $o \in S_w$. The converse holds if $w \prec o$. If neither version precedes the other, then since the set of versions pertaining to object $o.name$ in S_o is the same as $o.predecessors$, we have that $w \notin S_o$. The converse holds for $o \notin S_w$.

Lemma 3 *The Predecessors Invariant of Definition 5 holds throughout the updates made by the above protocols.*

Proof According to Lemma 2, replacing a predecessor vector with an extrinsic set cannot falsify the Predecessors Invariant. Let r be any replica, and o an object. Initially, $o.predecessors$ is set to empty if the replica's knowledge is extrinsic to o. The relationship between $o.predecessors$ and $r.knowledge$ need to be re-examined in the following events.

– Replica r generates an update on o. In this case, first $r.knowledge$ is updated with the new version, hence if $r.knowledge$ was extrinsic to o before the update, it continues being so after it.
– During synchronization, a new version of o is received from a source s. In this case, the requestor r explicitly stores the predecessor vector that arrives with the new version. Hence, the relationship between $o.predecessors$ and $r.knowledge$ is verified.
– At the end of synchronization, the source's knowledge $s.knowledge$ is merged into $r.knowledge$. Here, the only case we must consider is that

o.predecessors was $\perp$ before this step. Then we need to prove that the merged knowledge must remain extrinsic to o after the merge.

Let w be any version of object *o.name* included in *s.knowledge*. If $w \in r.knowledge$, then by the predecessors invariant it also precedes o. Hence, the extrinsic relation holds.

In all other cases, we reach a contradiction. Specifically, if $o \prec w$, then w would have replaced o in r's storage, and it is impossible that o is still stored. There remains the case that w and o are conflicting versions. However, in this case, in step 3(d) of the protocol above in Figure 2, *o.predecessors* would not be left empty. Again, we reach an impossible state according to the protocol.

Theorem 1 *The protocols above maintain Safety, Liveness, and Nontriviality, as defined in Section 4.3 above.*

Proof (Sketch) We already argued that the simple synchronization framework in Figure 1 maintains the desired Safety, Liveness, and Nontriviality properties, given any predecessor and knowledge sets that maintain the two invariants in Definition 4 and Definition 5. Since we proved the maintenance of the invariants, our proof is done.

6 Performance

The storage overhead associated with precedence and conflict detection comprises 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 information. In the PVE scheme, faultless scenarios result in one version being maintained per object, incurring a space of $\widetilde{O}(N)$. By comparison, the VV scheme keeps $\widetilde{O}(R \times N)$ storage, i.e. one version vector per object. In fairness, the per-replica counters used to generate versions in the PVE scheme may desire a larger number of bits than those used in a conventional version vector scheme since they are incremented for all updates to any object. However, WinFS can easily deal with counters that wraparound by retiring replicas once they have generated too many updates and replacing them with new replicas with fresh counters. Thus, the basic comparison between PVE and VV schemes in reliable communication environments remains valid.

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 predecessor 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 since the set of exceptions may grow with the number of updates. These formal upper bounds are also summarized in Table 1 below. Later 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 source and a requestor need to determine which objects have versions yet unknown to the requestor. In the PVE scheme, this is done by conveying the requestor's knowledge vector to the source. The faultless overhead here is $\widetilde{O}(R)$; the upper bound is again theoretically unbounded.

Let q denote the number of object versions that the source determines it has to send to the requestor. 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 version per object. Hence, the overhead is $\widetilde{O}(q)$. In case of faults, as explained before, some objects sent during synchronization may have explicit predecessor vectors and an unbounded number of exceptions associated with them. Hence, there is no formal upper bound on the communication 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, replicas may keep a log of all the objects that received updates since the last synchronization with the requestor and send only the version vectors associated with these objects. The communication complexity will be between $\widetilde{O}(q \times R)$ and $\widetilde{O}(N \times R)$, but the storage overhead increases due to logging.

	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.*

In face of communication faults, replicas using the PVE method might accumulate over time both knowledge exceptions and object versions that require explicit predecessors. There is no simple formula that describes how frequently exceptions are 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 the PVE 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, with replica 1 serving updates to 2, replica 2 serving 3, and so on, up to replica R sending updates back to 1. This was repeated 100 times. We expect that other communication patterns, such as randomly selecting synchronization partners, would yield similar simulation results. A failure-probability variable $pfail$ controlled the chances of a communication disruption within every pairwise synchronization. The disruption occurred at the end of the synchronization procedure, thus potentially causing the maximal number of exceptions. We measured the resulting average communication and storage overhead. These are depicted in Figure 3 for two cases, 100 and 1000 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 (without logging) 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. For quite moderate size systems (1000 objects), the cut-off point is beyond a 90 percent communication disruption rate.

7 Related Work

In weakly consistent replicated databases and file systems, conflicts are generally defined as concurrent updates to an object or file. In other words, conflicts arise when two clients independently update the same file at different replicas. More semantic definitions of update conflicts that take into account the needs of particular applications have been supported in replicated databases systems like Bayou [14]. Even though WinFS provides a richer data model than a conventional file system, we adopted the same notion of conflicts as in previous replicated file systems but devised a new scheme for detecting when conflicts occur.

In a client-server architecture where a hub-and-spoke or star topology is used for replication, conflict detection is relatively easy. For example, in the Coda system [8], servers maintain a version number for each file. A client records, for each file that it locally caches, the version number that the file had when it was retrieved from the server. When the client reconciles its local updates with the server after a period of disconnection, the client checks the current version of each file on the server. If the servers version for a file differs from the version on which the client based its update, then a conflict has occurred. This simple form of optimistic concurrency control works because the server is a central authority for each file.

Version vectors were devised for systems in which replicas reconcile with each other in a peer-to-peer fashion. Locus was the first system to use version vectors to detect concurrent updates to files [6], although Fischer and Michael proposed a similar data structure for resolving insert/delete ambiguities in replicated dictionaries [4]. Locus stored a version vector with each replica of each file. A files version vector included an entry for each site on which a copy of the file was replicated; entries in the version vector indicated the number of updates made to the file by each site. Two copies of a file are determined to be in conflict if their associated version vectors are incompatible, meaning that one version vector does not dominate the other. Follow-on systems to Locus, such as Ficus, Rumor, and Roam, utilize this same technique to detect conflicting file updates [7,11].

It has been shown that version vectors or related data structures, like vector clocks, are necessary to detect the causal ordering of events in a distributed system [3,13]. Concerns about the unbounded size of version vectors have caused some researchers to propose compact representations [1,2,5,15] or techniques for pruning entries that are no longer needed, such as entries that are globally known by all replicas [11,12]. These techniques could be adopted for use in WinFS, though they are less necessary since, as described in earlier sections, the PVE scheme maintains a single version vector for an entire replica rather than one for each file.

8 Conculsions

In optimistically replicated systems, metadata must be maintained for detecting conflicts caused by concurrent updates to data objects. The overhead for storing and communicating such metadata can be prohibitive. The traditional technique of using per-object version vectors simply does not scale to systems with thousands of replicas and large numbers of objects. WinFS, a new structured storage platform developed at Microsoft, was designed to support information management applications, such as electronic mail and calendar applications, with potentially millions of fine-grained data objects. WinFS allows objects to be replicated across machines running Windows, and thus must to scale from a handful of replicas in a home to thousands of replicas within a global corporation. In this paper, we have shown that the WinFS design meets these scalability demands by requiring only a single version vector per machine along with simple versions for each object. We provide the first proof that this unusually small amount of metadata is sufficient to detect concurrent writes to any data object.

An analytical comparison of the PVE scheme of WinFS to the conventional version vector scheme showed a factor of 50 improvement for scenarios with 50 replicas. These results were confirmed through simulation. As the number of replicas increases and the number of data objects increases, the benefits of the PVE design becomes even more pronounced. However, when running over unreliable networks, PVE synchronization sessions can be disrupted before their full completion, and the metadata maintained by PVE can grow over time due to holes in a replicas knowledge. In theory, PVE overheads can exceed the overhead of per-object version vectors. Our simulation studies show that this is only a concern, even for a small number of objects (100) and a modest number of replicas (50), when the percentage of failed synchronization sessions exceeds 40%, an unacceptably high unreliability in practice. For systems of 1000 objects, PVE overheads are strictly less even if 95% of synchronization sessions terminate prematurely.

In the future, we plan to evaluate the PVE design with real workloads gathered from the emerging WinFS applications. These studies should shed further light on the practical benefits of the new conflict detection scheme developed for WinFS.

Acknowledgements

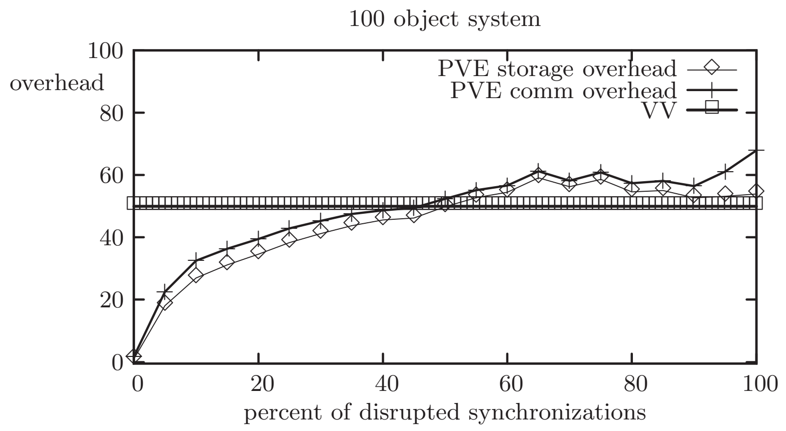
The protocol described in this paper was designed by the Microsoft WinFS product team, which included one of the authors (Doug Terry). We especially acknowledge Irena Hudis and Lev Novik for pushing the idea of concise version vectors. Harry Li, Yuan Yu, and Leslie Lamport helped with the formal specification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

References

1. J. B. Almeida, P. S. Almeida, and C. Bacquero. Bounded version vectors. In *Proceedings International Symposium on Distributed Computing (DISC)*, pages 102–116, 2004.
2. A. Arora, S.S. Kulkarni, and M. Demirbas. Resettable vector clocks. In *19th Symposium on Principles of Distributed Computing (PODC)*, 2000.
3. C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *In Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.
4. M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1982.
5. Y.-W. Huang and P. Yu. Lightweight version vectors for pervasive computing devices. In *Proceedings IEEE International Workshops on Parallel Processing*, pages 43–48, 2000.
6. 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.
7. 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.
8. J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
9. 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.
10. 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.
11. D. H. Ratner. *Roam: A Scalable Replication System for Mobile and Distributed Computing*. PhD thesis, 1998. UCLA Technical report UCLA-CSD-970044.
12. Y. Saito. Unilateral version vector pruning using loosely synchronized clocks. Technical Report Technical Report HPL-2002, HP.
13. R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.
14. 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.
15. F. Torres-Rojas and M. Ahamad. Plausible clocks: constant size logical clocks for distributed systems. *Distributed Computing*, 12(4):179–196, 1999.

D. Malkhi Dahlia Malkhi is a Principal Researcher in the Microsoft Research Silicon Valley lab. She received her Ph.D., M.Sc. and B.Sc. degrees in 1994, 1988, 1985, respectively, from the Hebrew University of Jerusalem, Israel. During the years 1995-1999 she was a member of the Secure Systems Research Department at AT&T Labs-Research in Florham Park, New Jersey. Her research interests include all areas of distributed systems.

D. Terry Doug Terry is a Principal Researcher in the Microsoft Research Silicon Valley lab. His research focuses on the design and implementation of novel distributed systems and addresses issues such as information management, fault-tolerance, and mobility. He currently is serving as Chair of ACM's Special Interest Group on Operating Systems (SIGOPS). Prior to joining Microsoft, Doug was the co-founder and CTO of Cogenia, Chief Scientist of the Computer Science Laboratory at Xerox PARC, and an Adjunct Professor in the Computer Science Division at U. C. Berkeley, where he regularly teaches a graduate course on distributed systems. Doug has a Ph.D. in Computer Science from U. C. Berkeley.



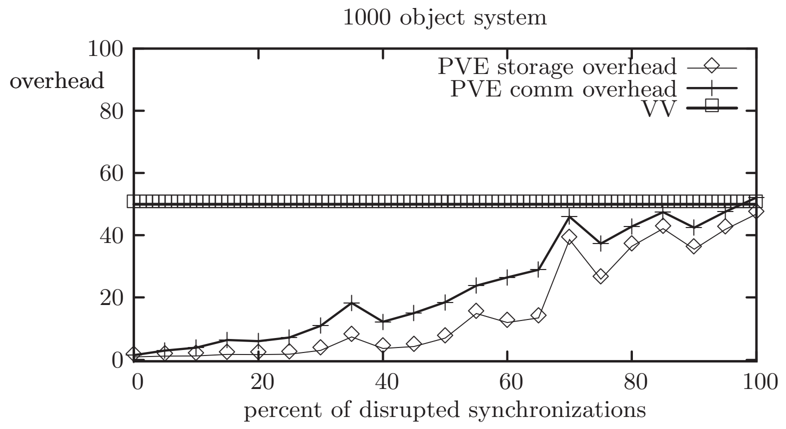


Fig. 3 Per-object storage and communication overheads for varying communication failure frequency with $N = 100$ objects (top) and $N = 1000$ objects (bottom).