

Peer-to-Peer Replication in WinFS

Lev Novik, Irena Hudis, Douglas B. Terry, Sanjay Anand,
Vivek J. Jhaveri, Ashish Shah, and Yunxin Wu

Microsoft Corporation

ABSTRACT

WinFS, Microsoft's new application storage platform, incorporates a novel peer-to-peer, knowledge-driven, state-based replication protocol. The goal is to support diverse applications requiring replication for easy sharing, high availability, and offline access. The system was designed to scale from a handful of personal computing devices in a home environment to thousands of servers that are globally distributed. Like other weakly consistent replicated systems, the WinFS replication model allows update operations to be performed on any machine without locking. Updated data items are sent between replicas in a lazy fashion via a pair-wise synchronization protocol. The technical contributions of the protocol's design include minimizing replication-specific state and efficiently propagating updates while allowing arbitrary synchronization topologies, detecting conflicting updates, supporting automatic conflict resolution, accommodating overlapping user communities, and guaranteeing eventual convergence.

1. Introduction

WinFS Sync is a replicated data service designed to support the needs of a broad class of applications. This replication protocol has been developed as part of a new storage system for Microsoft Windows called WinFS. Replication serves the dual purpose of facilitating the sharing of information across machines and increasing the availability of that information. For example, WinFS permits sales representatives to carry copies of a shared customer relationship database on their individual laptops, update data items while on the road, and later upload their updates to a server machine in the office or directly share updated items with their coworkers over a wireless network.

The WinFS replication protocol is characterized as peer-to-peer, knowledge-driven, and state-based. *Peer-to-peer* means that the protocol allows pairs of machines to synchronize updates independently, thereby sending updated items through an application-specific overlay network. *Knowledge-driven* means that sites maintain and exchange summaries of the updates they know, and sites use this information when deciding what items to send during synchronization. *State-based* means that sites send updated items from

their local databases rather than the sequence of operations that produced those items; the protocol does not rely on a write log but instead utilizes a small amount of metadata associated with each database entry. The protocol achieves robustness through its knowledge-driven design, flexibility from a peer-to-peer model, and scalability by conserving network bandwidth and maintaining minimal metadata. This paper presents the first replication system that combines these three properties to meet the following demanding design goals:

Operate on WinFS items. Any data item can be replicated on any machine running WinFS.

Support disconnected operation. Data items may be replicated on machines that have intermittent network access, such as laptops and other mobile devices.

Rely on minimal metadata. The information used to drive the replication process must be proportional to the size of the database and not grow with the update rate.

Deliver updates at most once. Once a replica receives a given version of an item, it should not be sent that version again.

Detect conflicting versions. Concurrent updates to an item must be detected and either resolved automatically or logged for manual resolution.

Scale to thousands of replicas. Communities of 5,000 read-write replicas and 100,000 read-only replicas must be supported.

Guarantee eventual convergence. All sites that share replicated data must eventually receive all updated data items and agree on the latest version of each item.

These requirements and the motivation behind them are discussed in more detail in the following section. In Section 3, we then show that none of the known replication protocols meet all of these requirements, thereby justifying our need to develop a new protocol. Section 4 presents the WinFS replication architecture and key concepts, and Section 5 describes the synchronization protocol in detail. Section 6 outlines the steps taken to validate the protocol's correctness.

Section 7 then discusses implementation and performance issues. Finally, Section 8 concludes by reviewing the protocol features that allow it to meet the stated requirements.

2. Replication requirements

The stringent and varied requirements placed on the WinFS Sync design stem from the need to support a wide class of applications and scenarios. WinFS must operate in home environments with a handful of machines that use replication to share user profiles and application settings. WinFS Sync is also targeted for large global corporations that want to replicate critical data across thousands of protected servers scattered throughout the world. Between these two extremes lies a gamut of applications that support collaboration among communities of Internet users, such as photo sharing and electronic mail applications. This section elaborates on the specific requirements previously listed.

Operate on WinFS items. First and foremost, the replication protocol was developed as an integral part of the WinFS storage platform and manipulates WinFS *items*. Items are independent data objects storing information about real-world objects, like people, places, and meetings, as well as electronic artifacts, such as e-mail messages, digital photographs, songs, music playlists, and other types of documents. Each item in WinFS consists of one or more non-overlapping *change units* and is typed by a schema describing the item's XML-like contents. A change unit represents the granularity of updates that are sent over the wire via replication as well as the basic unit of conflict detection and resolution. Change units can be as small as the individual properties of an item, such as a person's phone number or the read flag of an e-mail message, or as large as the whole item. The schema designer for a given item type faces the tradeoff between defining small change units, which results in more metadata, and large change units, which can result in increased conflicts due to false sharing. The replication protocol must deal with an extensible set of item types with their individually defined change units.

Support disconnected operation. Laptops and mobile devices represent an increasingly important platform for the Windows operating system. Such devices often rely on wireless networks for their connectivity to the rest of the computing world and regularly operate without any connectivity. Accommodating offline access to a local WinFS database was thus a critical requirement, and replication is the primary means for sharing information among occasionally connected machines.

This necessitates an update-anywhere style of replication in which replicas' contents are allowed to temporarily diverge due to independent updates. Peer-to-peer synchronization was adopted to allow replication among groups of machines that have only local connectivity, such as through an ad hoc wireless network.

Rely on minimal metadata. Despite the steady growth in disk capacity, Windows users regularly experience local file space shortages. Thus, application designers are reluctant to accept storage platforms whose metadata (data that is not directly visible to application users) is unbounded in size or is large compared to the application-managed information. This concern about unrestrained storage overhead is one of the major reasons why WinFS Sync avoided reliance on an update log (the other reason being logging's impact on write performance, as discussed in Section 7). It also dictated a design in which the metadata associated with fine-grained change units was kept to a minimum.

Deliver updates at most once. Users want control over which sites synchronize with which other sites and at what times. In some scenarios, each site directly exchanges updates with all other sites, while in other scenarios, sites are arranged in hierarchies or rings with redundant paths so that temporary site failures do not disrupt the propagation of updates. The choice of synchronization partners and schedules should have minimal impact on the bandwidth utilized to fully propagate updates to all replicas. Specifically, each updated item should be sent at most once to each replica regardless of the number of redundant paths between sites.

Detect conflicting versions. Conflicts caused by applications or users concurrently updating the same item are an inevitable consequence of an update-anywhere replication model. To avoid "lost writes" [21], customers require WinFS to detect any conflicting updates that may arise. On the other hand, updates that are causally related or that modify independent items should never be reported as a false conflict. Applications want to be informed of detected conflicts so that they can be automatically resolved via registered conflict handlers or logged for later resolution. WinFS support for conflict management can be extended by defining new conflict handlers for different types of items.

Scale to thousands of replicas. The scalability requirements faced in the design of WinFS were particularly demanding. WinFS is expected to run on millions of computers, which may participate in replication. However, sites that do not directly share

items should not affect each other. For information sharing applications, WinFS replication must scale from a couple of machines in a home or small office to thousands of machines distributed globally. The specific goal was to support 5,000 read-write replicas of any given item, which is the degree of replication in the largest Active Directory services deployed by Microsoft customers. WinFS must also support 100,000 replicas of a single-mastered database such as the Microsoft company address book, which is replicated on every corporate machine and receives thousands of updates daily. Large replication factors, in practice, arise from autonomy desires and simple configuration policies rather than from a need to tolerate thousands of failed machines. For example, each store of a large retail chain may require its own copy of the product catalog. Internet users generally want shared data to be locally accessible behind their firewalls. Even in the home environment, full replication often simplifies system management. Although the handful of machines in a home could readily share centrally managed files, this would require configuring one of the machines as a server and may lead to problems later when a teenage child takes a machine off to college.

Guarantee eventual convergence. Although support for disconnected operation dictated a protocol with weak consistency properties, the system must guarantee that all replicas eventually converge to a consistent state. That is, two sites, after synchronizing with each other, should have identical databases (for the items being replicated). When an application performs a sequence of updates, or when a conflict handler resolves a conflict between updates, the latest versions of updated items should eventually propagate to all replicas, assuming sufficient (though perhaps intermittent) connectivity between sites.

3. Related work

The computing literature over the past 30 years is replete with protocols for replicating data. While early papers focused on maintaining one-copy serializability [2][4][7], optimistic protocols [5][21] have steadily gained in prominence. Grapevine [3], Notes [10], and other systems demonstrated that weak consistency replication is acceptable for the personal information management and asynchronous collaborative applications that WinFS is intended to support. WinFS replication is based on epidemic algorithms [5], which are known to exhibit good robustness and scalability [27].

Coda pioneered the notion of disconnected operation for arbitrary file system applications running on mobile

computers [11][23]. Coda adopted a client-server model in which clients log local file updates while offline and replay the log upon reconnection, which contrasts with the WinFS peer-to-peer synchronization model that supports intermittently connected devices without an explicit disconnected mode of operation.

Locus was the first replicated file system based on a peer-to-peer model [28]. The basic Locus architecture evolved into Ficus [16] and Roam [19]. Ficus triggers update propagation through best-effort notifications and uses infrequent pairwise reconciliation to ensure convergence. Ficus's scheme of maintaining one version vector per file and fully exchanging file metadata during reconciliation would not scale for the many fine-grained change units managed by WinFS. To achieve better scalability, Roam adopted a hierarchical ward model [19]. By maintaining per-replica knowledge and per-change-unit versions, the WinFS protocol is much more efficient in bandwidth and storage consumption while using pure peer-to-peer reconciliation.

WinFS Sync's knowledge-driven design borrows from previous protocols for optimistic replication that maintained tables or vectors summarizing other replicas' knowledge [1][6][14][29]. Early papers showed that knowledge could be used to avoid insert/delete ambiguities [6], but out-of-date information yielded protocols that would routinely distribute updates multiple times. Golding's and Bayou's anti-entropy protocols solved this redundant communication problem for peer-to-peer models by having a site request changes and send its own version vector at the start of synchronization [8][18]. The WinFS synchronization protocol is similar in style to Bayou's [18]. WinFS replication achieves the same basic properties, including support for disconnected workgroups, diverse networks, arbitrary communication topologies, and dynamic replica sets, but does so without relying on a write log. The state-based approach adopted by WinFS sends fewer update messages in that only the latest version of an item needs to be sent between replicas rather than every intermediate update. By adopting an update and conflict model that treats items independently, WinFS was able to avoid much of the complexity in Bayou dealing with commit ordering and rollback of tentative writes [26].

A number of systems with optimistic replication provide automatic detection and resolution of update conflicts, including Coda [12], Ficus [20], and Bayou [26]. Such systems fully propagate conflicts to all sites and assume that application-specific conflict handlers run at each site to produce deterministic resolutions.

WinFS, on the other hand, resolves conflicts as they are detected and sends resolutions via the regular replication protocol, thereby guaranteeing convergence.

Some optimistically replicated systems have provided consistency models that are more elaborate than the eventual consistency provided by WinFS, including session guarantees [25] or bounded inconsistency [30]. These additional consistency guarantees are not supported by WinFS, but could be integrated into the system if desired by future applications.

4. System architecture

The WinFS distributed system architecture incorporates federations of machines operating as peers. WinFS supports replication communities that may overlap in both membership and shared information content. This section introduces the basic architectural features of WinFS Sync.

4.1 Communities and community folders

WinFS presents users with the notion of a *community folder*, a logical collection of items that are replicated among members of a *community*. Any user or application can create a new community and invite others to join. Each community is assigned a globally unique identifier (GUID). Communities can be advertised by a variety of means, including e-mail, name servers, web page postings, and application-specific techniques.

Each community *member* is a WinFS site that maintains a local *replica* of all of the items belonging to a community folder. Like communities, members have globally unique ids. In many information-sharing scenarios, members are viewed as humans with individual computers, but this does not always hold. For instance, a home PC and a laptop computer belonging to the same user may be different members and, hence, hold different replicas of the community folder containing that user's e-mail.

Each replica is persistently stored in a user-visible WinFS *container*, which is similar to a file directory. The local pathnames of containers storing replicas may vary across members of the community. In particular, WinFS does not provide a global name space for replicated items. On a shared PC, two users may have different replicas of the same community folder in separate personal containers.

4.2 Operations on replicated items

Members contribute items to a community folder by adding items or moving items into their local replicas. Items are removed from a community folder when they

are deleted or moved out of a replica. Different community folders may overlap in content. In other words, a given item can be in two or more community folders.

Replicated items in a community folder can be read and updated by any member of the community (assuming they have appropriate permissions, as discussed below). Applications perform operations on a replica without coordination with other sites. Thus, WinFS provides an *update-anywhere* model, also referred to as *multi-master* replication [21]. Locally updated items lazily propagate to all of the other replicas via the peer-to-peer replication protocol (which is described in detail in Section 5).

Because community members are allowed to operate independently on their local replicas, they may take actions that conflict with each other. WinFS detects two types of update conflicts: *constraint conflicts* and *concurrency conflicts*. Constraint conflicts arise when two or more operations modify items such that they violate a system or application-defined invariant. For example, WinFS prohibits two items in the same folder from having the same name. Constraint conflicts of this sort are detected via database triggers and are not further discussed in this paper. Concurrency conflicts arise when the same change unit is updated in parallel at two replicas. This type of conflict is discussed in more detail in Section 5.1.4. Concurrent updates to different change units are treated independently in WinFS and never result in concurrency conflicts.

Detected conflicts are resolved immediately by registered *conflict handlers* or logged for subsequent resolution by humans or special applications. The specific conflict handler invoked depends on the type of conflict and the type of the item for which a conflict was detected. If a suitable conflict handler does not exist locally, then the conflict is logged and the conflicting updates continue to propagate to other replicas where they may be automatically resolved.

4.3 Synchronization topologies and schedules

The synchronization *topology*, indicating which members directly exchange updates with which other members, may vary across communities. The topology is an overlay network through which updates flow to all members of a community. Some communities may be configured in a star topology where a centralized "server" acts as the distribution point for updated items. Other communities may be structured as a fully connected clique, a tree, or some hybrid.

Similarly, the *schedule*, indicating desired times for synchronization, may vary according to the needs and characteristics of community members. For example,

in a small office, desktop machines may synchronize frequently to achieve a high degree of data consistency. In a globally distributed system with high communication costs, replication may be infrequent or may be scheduled to coincide with work patterns. Occasionally disconnected laptops with local replicas may synchronize among themselves or with other members as connectivity permits.

To join a community, a WinFS machine need only contact an existing member to establish a *synchronization partnership*. Such pair-wise partnerships define the synchronization topology and schedule. New members start with an empty replica and receive community folder items via the usual peer-to-peer replication protocol. To resign from a community, a site can simply terminate its partnerships with community members. Members who leave a community retain copies of the items that existed in the community folder at the time of their departure. However, they no longer receive updates to these items and no longer send locally updated versions.

4.4 Security

The security model is simple and mostly orthogonal to the replication architecture and protocol, which are the main technical contributions of this paper. WinFS relies on the existing Windows authentication mechanism and supports access control lists (ACLs) for granting access to items. Replication introduces no new security requirements beyond those needed for remote access. The ACL for an item is not part of the data that is replicated. Each site unilaterally manages the ACLs for its local items and, therefore, can independently decide which community members it trusts. A site authenticates its partner during synchronization. When sending updates, a site should check that the requesting member has read access to the items. All items stored in a given replica must have identical read ACLs, and thus a site can simply reject synchronization requests from members that lack sufficient permission without needing to check individual items. When receiving updates via replication, a site should verify that the sender has write access to the items. A member may be able to write some items in a community folder but not others. However, to avoid confusion and preserve eventual consistency, community members should agree on which items are writeable by which members, and read-only members need to have writers as partners in the synchronization topology. Typically, all members have read and write access to all community items, though the sites that can join a community may be restricted. Applications can layer finer grained

protection on top of this basic infrastructure, for instance, using rights management services.

5. Peer-to-peer, knowledge-driven, state-based synchronization

The WinFS Sync protocol is similar in style to that developed for the Bayou system [18] in that a community member initiates the process by sending a description of its current knowledge and receives back a set of updated items that it has not yet seen. However, operating without a write log raises new challenges in how changes are tracked, how knowledge is represented and managed, and how conflicts are detected. The following section describes the metadata that is used by the replication protocol to efficiently perform these functions, and Section 5.2 then presents the protocol in detail.

5.1 Metadata

5.1.1 Item and change unit IDs

WinFS items are assigned globally unique identifiers (GUIDs) at the time of their creation. Once created, an item retains its ID throughout its lifetime. The change units comprising an item are also assigned unique identifiers derived from the item's GUID. These identifiers allow community members to refer to shared items and change units during synchronization. While items generally have human-assigned pathnames, similar to hierarchical file names in a traditional file system, such names are not used by the replication protocol. An item's name is simply a mutable property along with all of the other properties defined by the item's schema type.

5.1.2 Versions

Whenever a change unit is created or modified by a user or application, it is assigned a new *version*. The version associated with each change unit consists of the unique identifier of the member site on which the update originated along with a replica-specific counter that is incremented on each update to any data item. In examples used throughout this paper, capital letters serve as site identifiers and numerical values serve as update counters. For example, version A17 refers to the contents of a change unit that was updated on site A and was the seventeenth version produced by this site; version A18 may result from an update to the same or a different change unit.

WinFS allows applications to perform transactions that atomically update multiple change units belonging to one or more items. In such a case, each modified change unit is assigned a different version when the transaction commits. For example, a transaction that touches three change units may produce versions A17,

A18, and A19. Thus, a given version, like A17, uniquely specifies a specific change unit and its specific contents.

Deleting an item, like any other update operation, produces a new version of the item. The deleted item is not immediately removed from the local replica. Semantically, a delete operation simply sets a special bit on the item indicating that the item has been deleted and, optionally, discards the contents of the item's change units. An item with the deleted bit set is called a *tombstone*. Such items are ignored by normal database queries but propagated via synchronization to inform other sites of the item's deletion. Tombstones are garbage collected via a process described later in Section 0.

As a change unit is updated over time, it is assigned different versions while maintaining its unique identity. The sequence of versions of a change unit constitutes its replica-independent *version history*. Figure 1 depicts a sample timeline of updates to a shared customer database with three items (which have a single change unit) and their corresponding version histories.

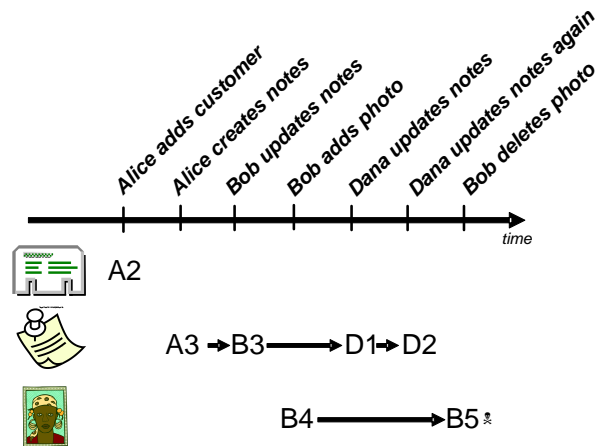


Figure 1. Sample scenario with version history.

A version history is not always a linear sequence but is more generally a directed acyclic graph in which nodes are versions and edges represent causal dependencies. When a member updates a change unit, an edge is added to the change unit's version history connecting the replica's previous stored version to its new generated version. A node may have two or more direct descendants in a version history, indicating that two or more versions were produced concurrently at different replicas. Similarly, a node may have multiple direct ancestors, indicating that its version resulted from resolving two previously conflicting versions of the change unit.

While it is instructive to think about version histories when understanding the synchronization protocol, WinFS does not explicitly maintain unbounded version histories. For change units with no unresolved conflicts, which is the common case, the only metadata in each replica is the change unit's unique identifier and latest version stored by the replica. Thus, a member that has received all of the updates depicted in Figure 1 would have a replica as shown in Figure 2. In the rare case of unresolved conflicts, multiple versions are retained along with a suitable approximation of the version history, as discussed in Section 5.1.4 below.




Unit ID	Version	Deleted?	Contents
039871	A2	no	
039903	D2	no	
046452	B5	yes	

Figure 2. Sample replica.

5.1.3 Replica knowledge

The peer-to-peer synchronization protocol is driven by per-replica *knowledge*. Conceptually, knowledge is a set of versions defined by the following invariant.

Knowledge invariant: A member's knowledge:

- (a) must include any version of a change unit that has been received by the member and stored in its replica,
- (b) may include any version that precedes a received version in its version history, and
- (c) may not include any versions that do not satisfy condition (a) or (b).

This invariant is easy for a site to maintain because it involves purely local information. Although members exchange their knowledge during synchronization, they do not retain information about the knowledge of other members since this may become quickly out of date.

Although a member's knowledge increases over time, a *version vector* [17] representation ensures that the space needed to store basic knowledge is proportional to the number of replicas rather than the number of updates. Each knowledge vector contains an entry for each member site indicating the set of contiguous versions generated by the site of which the local member is directly or indirectly aware. For example, a member that is fully aware of the events in Figure 1 would have knowledge <A3, B5, D2>, which denotes

that the member knows all versions generated by Alice at site A with counters 1 to 3, all versions from Bob at site B up to B5, and versions D1 and D2 from Dana at site D.

Version vectors need not contain entries for read-only replicas or, more generally, for sites that have not updated any items and hence have not generated any versions. Thus, members joining a community do not immediately increase the size of knowledge vectors for that community. When a member first updates a community item, the member is automatically added to its own version vector. As this updated version vector is exchanged during synchronization, the community learns of the new member along with the version that it created. Sites that leave a community cannot be dropped from version vectors since their updates may still need to be propagated among remaining members.

Since version vector entries are a fixed size, in practice, an active member may exhaust its supply of versions. When this happens, recovery is straightforward. The member can retire and simply reincarnate itself as a new member with a new unique identifier and a fresh update counter. The new member inherits the retired member's database, knowledge, and synchronization partnerships.

If all versions are received in the order they are generated by each site, then a version vector suffices to precisely represent a member's knowledge. The Bayou system, for instance, needed only simple version vectors since it guaranteed that updates were sent in log order over a reliable stream-oriented transport protocol [18]. WinFS Sync, on the other hand, avoids sending obsolete versions and was designed to work over transports that may drop or reorder messages. Thus, WinFS supports an additional type of knowledge.

Knowledge *exceptions* are versions that have been locally received but for which some earlier version generated by the same site is "missing." For example, the local member may have received version B5 without receiving version B4. A number of scenarios can lead to this situation. For instance, version B5 may have overwritten B4 (as in Figure 1), or they may be versions of different change units that were sent out of order. In general, a member's knowledge consists of a single version vector plus zero or more exceptions. When operating over a reliable network, knowledge exceptions are important but temporary, as explained in Section 5.2.4.

One member's knowledge *dominates* another member's knowledge if its set of known versions is a superset of the second member's known versions. Specifically, each site in the second member's version

vector should also be present in the dominating member's knowledge with an equal or larger counter value, and each exception in the second member's knowledge should be an exception in the first member's knowledge or included in its version vector. In general, due to the relaxed consistency model, two community members may be in a state where neither's knowledge dominates the other's. The goal of the synchronization protocol is to drive all members of a community towards common knowledge.

5.1.4 *Made-with knowledge*

When presented with a version of a change unit that it has not previously seen, a member needs to decide whether this version causally precedes the version stored in its replica and hence can be ignored, causally follows the version in its replica and hence should replace the previously stored version, or neither, thereby resulting in a concurrency conflict. Given the version histories for two versions, determining causality is simply a matter of checking whether one version is in the other's version history. Unfortunately, version histories grow indefinitely.

To obtain an approximation of a version history with bounded size, WinFS uses *made-with knowledge*. Specifically, the made-with knowledge for a version of a change unit is the knowledge that the updating member had when it generated the version. In the scenario in Figure 1, the made-with knowledge for version D2 is either <A3, B3, D1> or <A3, B4, D1> depending on whether site D had seen version B4 when D2 was produced. It is well-known that version vectors can detect concurrent updates if managed for individual files [17][24], but WinFS uses site-wide versions and manages knowledge differently than previous systems. The rest of this section reasons about the conditions placed on made-with knowledge and its relationship to per-replica knowledge.

Made-with knowledge in WinFS, like replica knowledge, is represented by a version vector and a number of knowledge exceptions but is conceptually a set of versions that satisfies the following invariant.

Made-with knowledge invariant: A change unit's made-with knowledge for version V:

- (a) must include any version that precedes version V in the change unit's version history,
- (b) may include version V itself,
- (c) may include versions of other change units, and
- (d) may not include any versions that do not satisfy condition (a), (b), or (c).

Suppose, for the moment, that a change unit's made-with knowledge is stored along with its version and sent with the change unit during replication. A site can check whether two distinct versions of a change unit, V1 and W2, have a causal relationship as follows: version W2 *overwrites* version V1 if and only if V1 is in W2's made-with knowledge. The two versions are *concurrent*, and hence *conflicting*, if neither overwrites the other [24].

Importantly, performing this causality check produces the same answer as checking versions against a complete version history. The formal proof is beyond the scope of this paper, but an informal argument is as follows. Because of clauses (a) and (b) in the made-with knowledge invariant, made-with knowledge is clearly a superset of the versions included in the change unit's version history. Because of clause (c), any versions in a change unit's made-with knowledge that are not part of its version history are versions of other change units. These extra versions cannot affect the causality check, which only considers versions belonging to the same change unit.

While made-with knowledge is sufficient for detecting conflicting versions, the overhead of storing and communicating explicit made-with knowledge for each change unit in a WinFS database would be substantial, especially for small change units. This brings us to a key and perhaps surprising observation. Consider the typical scenario where conflicting versions are automatically resolved, and thus each replica stores a single latest version of each change unit. In this case, *a member's knowledge satisfies the made-with knowledge invariant for each change unit in its local replica*. Therefore, the knowledge of the two members involved in synchronization can be used to detect concurrency conflicts without the need for extra metadata. Only in the case of unresolved conflicts does WinFS need to explicitly store made-with knowledge for the conflicting versions.

The bottom line is that made-with knowledge can serve as a suitable substitute for a complete version history, and per-replica knowledge, in most cases, can serve as a suitable substitute for made-with knowledge.

5.2 Protocol steps

Figure 3 depicts the messages exchanged during synchronization ordered from top to bottom. This protocol has been implemented on several transport layers, including TCP, SOAP, and a proprietary transport. Note that the protocol transfers updated change units in one direction only. For the communicating members to achieve mutual consistency, each site must alternate serving in turn as

the source and the target. The following subsections describe the basic protocol phases, concentrating on the semantics and processing of each message rather than the details of their representation on the network.

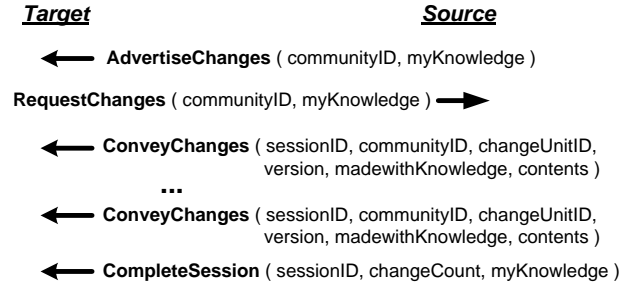


Figure 3. WinFS synchronization protocol.

5.2.1 Requesting changes

A member that has updated items locally or received updated items from others can inform another member of those changes via an **AdvertiseChanges** message. This message includes the sending member's knowledge and an indication of the associated community. If the receiving member's knowledge dominates the advertised knowledge, or if the site no longer wishes to participate in the community, then it may ignore the advertisement. Otherwise, the recipient should respond by requesting changes. Advertising changes is useful, for instance, when a user has modified items in his local replica and wants those updates distributed before disconnecting his laptop.

Typically, the protocol is initiated by the *target* member sending a **RequestChanges** message to one of its synchronization partners, the *source* member. This message includes the target's current knowledge and the global identifier for the community folder being synchronized. Upon receipt, the source queries the items in its replica of the community folder to determine which change units have versions that are not already included in the target's knowledge.

A member may concurrently request changes from multiple partners or may simultaneously serve as the source for one synchronization session and the target for another. The only negative consequence of concurrent synchronization sessions is that a member may be sent the same version by multiple partners, thereby wasting bandwidth.

5.2.2 Sending changes

In response to a **RequestChanges** message, the source returns zero or more **ConveyChanges** messages containing change units whose stored versions are unknown to the target. Conceptually, a separate **ConveyChanges** message is sent for each change unit,

though multiple messages may be packed into one network packet. Change units can be sent in any order since WinFS assumes that items are semantically independent. The one exception is that change units belonging to a single item are grouped together and processed atomically by the target.

A `ConveyChanges` message includes the contents of a change unit stored in the source's replica, as well as its unique identifier, version, deleted bit, and made-with knowledge. The message also includes a unique `sessionID` generated by the source whose purpose is discussed later in Section 5.2.4. As noted above in Section 5.1.4, members need only maintain explicit made-with knowledge in rare cases. If no explicit made-with knowledge is stored for a conveyed change unit, then the source's knowledge is sent as the change unit's made-with knowledge. Interestingly, this means that most or all of the `ConveyChanges` messages sent during a synchronization session include identical made-with knowledge, which is the same knowledge that is included in the final `CompleteSession` message. Thus, bandwidth can be saved by sending this knowledge only once.

Suppose that Alice from our scenario in Figure 1 has knowledge $\langle A3, B3 \rangle$ and requests changes from Bob, whose replica is fully up-to-date as in Figure 2. The messages exchanged during synchronization are shown in Figure 4 (though some message fields are omitted to simplify the figure).

```

RequestChanges ( <A3, B3> ) →
← ConveyChanges ( D2, <A3, B5, D2> )
← ConveyChanges ( B5, <A3, B5, D2> )
← CompleteSession ( <A3, B5, D2> )

```

Figure 4. Synchronization between sites A and B.

5.2.3 Processing changes

When the target member receives a `ConveyChanges` message, it first checks whether the enclosed version is already known. This check filters out old messages that were delayed in delivery or were received twice due to concurrent synchronizations. Next, the target uses the made-with knowledge to determine if the received version conflicts with its replica's stored version. If no conflict is detected, then the contents, version, and made-with knowledge of the received change unit replace the previous values in the target's replica.

If a concurrency conflict is detected (or a constraint conflict is detected while attempting to update the replica), the target reports the conflict to the list of

locally registered conflict handlers. A conflict handler may resolve the conflict by selecting one of the conflicting versions or returning new contents for the change unit. If resolved, the chosen contents are applied locally and assigned a new version by the target site. Thus, actions taken to resolve a conflict are treated the same as local updates, and the resolution propagates to other members via the normal synchronization protocol.

If a conflict cannot be automatically resolved, the received conflicting version is stored in a special conflict log. Essentially, the target's replica now contains two or more versions of the change unit in conflict. Only one of the conflicting versions is visible to normal database queries. All of the conflicting versions, however, are available through a special API used by applications that present conflicts to users for manual resolution. When a conflict is logged, the received made-with knowledge is explicitly stored with the conflicting change unit. Moreover, before updating its local knowledge to include the newly received conflicting version, the target stores its current knowledge as the made-with knowledge for other versions of the change unit stored in its replica (unless these versions already have explicit made-with knowledge).

If a `ConveyChanges` message includes a change unit for which there is already a logged conflict, then several outcomes are possible. The new version may include all of the previous conflicting versions in its made-with knowledge, thereby fully resolving the conflict. In this case, the target not only replaces its stored version with the new version, but also removes any logged conflicts involving the specific change unit. Alternatively, the new version may resolve some but not all of the conflicting versions, may overwrite one of these versions, or may itself conflict with all of the previous conflicts, thereby adding another version to the conflict log.

Regardless of conflicts, the target must add any received and stored versions to its knowledge. In some cases, a version can be added to the target's knowledge by simply increasing an entry in its version vector. For example, if the target's knowledge indicates that it knows all versions from site B through B3 and it receives version B4, then its knowledge vector entry for B simply becomes B4. But what if the target receives version B5 without knowing B4? This situation could arise because version B4 was sent or is about to be sent in a different `ConveyChanges` message that has not yet arrived or was lost in transmission. Version B5 may have overwritten version B4 (as in Figure 1), or B4 could be a version of an item that is

not in the community folder being synchronized, in which case B4 will never be sent. Regardless, upon receiving version B5, the target adds an exception to its knowledge indicating that it now knows this version. To ensure that the target is not sent duplicate versions, such exceptions are part of the knowledge included in its future RequestChanges messages.

In order to maintain the knowledge invariant presented in Section 5.1.3, a member must store new versions of change units and update its knowledge in a single atomic transaction. A separate transaction could be used for each conveyed change or one transaction used for the whole synchronization session. In practice, the WinFS implementation atomically updates a batch of received change units. The size of the batch is chosen to trade off the transaction overhead against the amount of lost work if the transaction aborts. If a synchronization session terminates prematurely due to a machine crash or network disconnection, updated knowledge ensures that the next session will resume after the last committed transaction.

5.2.4 Completing the session

After processing received changes, the target may be left with a number of knowledge exceptions. The CompleteSession message allows it to remove these exceptions where possible. This final message includes the unique sessionID that was included in each ConveyChanges message and a count of the number of such messages that were sent in this session. Note that the session ID and count are not needed if messages are sent over a reliable ordered transport protocol, since receiving the CompleteSession message guarantees that all previous messages were received as well. Most importantly, the CompleteSession message includes the source's knowledge as it existed at the start of the synchronization session. If the target successfully received and processed all of the ConveyChanges messages, then it is at least as knowledgeable as the source. Therefore, the target can merge the source's knowledge into its own knowledge. The target's new version vector is obtained by taking the maximum of each entry in the source and target version vectors. The source's knowledge exceptions, if any, are added to those of the target.

Once the source's knowledge is merged in, any explicitly stored made-with knowledge for non-conflicting change units can be discarded if it is dominated by the target's knowledge. Moreover, the target's knowledge can be cleaned up by removing any knowledge exceptions that are now covered by its version vector. Typically, this process removes all or most of the knowledge exceptions. Certainly, exceptions caused by items being conveyed or arriving

out of order will be removed. Importantly, exceptions arising from "holes" in the knowledge due to overwritten versions can also be removed.

For example, consider the situation where a user at site B edits a document and saves it periodically, thereby generating a sequence of overwriting versions including B4 and B5. Suppose that the target requests changes directly from site B. As discussed earlier, upon receiving version B5, the target adds an exception to its knowledge because it has not seen version B4 (and never will). However, the source's knowledge indicates that it knows all versions from itself (site B) including B4 and B5 and possibly later versions that it generated. Thus, when the target receives the CompleteSession message and merges in B's knowledge, its exception for version B5 becomes superfluous. A similar sequence of events happens when the target then serves as the source in other synchronization sessions and conveys version B5 to other members.

Exceptions remain in a target's knowledge only if its synchronization session ends before processing the CompleteSession message or if it fails to receive a ConveyChanges message (or if it synchronizes with a source that has knowledge exceptions due to a previous interrupted session). In the worst case, the target may end up with an exception for each version that it received before the session was terminated (though Section 5.3.4 below describes a technique for reducing the number of knowledge exceptions). This is only a temporary situation in that these exceptions will be cleaned up when the member next performs a complete synchronization.

An alternative design would be for the target to process all conveyed changes and merge in the source's knowledge in a single atomic transaction. This alternative would avoid the need for knowledge exceptions altogether. However, when operating over an unreliable network, two members that have many updates to exchange may have difficulty completing a synchronization session. This could lead to a "cliff" phenomenon in which the members become more divergent over time and the likelihood of successfully conveying all of their changes decreases. The approach adopted for WinFS, allowing the target to commit small batches of changes while adding knowledge exceptions, permits synchronization to make progress in the face of frequent disconnection.

5.3 Other replication issues

This section highlights issues related to synchronization that are important aspects of the WinFS system.

5.3.1 *Meta-data cleanup*

Although the tombstones arising from deleted items are small in size, they should be reclaimed in due course. A tombstone no longer serves a purpose once it is known to all members. If the replica set is well-known, techniques exist for reliably discarding tombstones [9][22]. Unfortunately, because sites can join a community simply by establishing synchronization partnerships with existing members, the membership in a community is fluid and members are likely to be unaware of the complete membership of their community. The current WinFS implementation simply discards tombstones after a fixed timeout, and each member maintains knowledge of the tombstones that were deleted, similar to Bayou's omitted vector [18]. A scheme for determining the "common knowledge" of a community has been designed for future implementation. It uses a ring structure and a token-passing protocol modeled after Lamson's sweep operation [13].

Each member of the community maintains a single pointer to some other member, forming a ring of all members. A ring has two attractive properties. First, a ring can be constructed incrementally and concurrently without any site knowing the full membership. In particular, when joining a community, a new site is spliced into the ring by an existing member without the need for external coordination. The new member simply inherits the existing member's next pointer and becomes its downstream neighbor. Second, a token circulating around the ring is guaranteed to have visited each member when it returns to the initiator. Thus, common knowledge can be acquired as follows. Periodically, a member creates a new token that includes its current knowledge and passes this token to its downstream neighbor. Upon receipt of a token, a member intersects the knowledge in the token with its own knowledge and replaces the token's knowledge with the intersection. When the token completes a pass around the ring, it contains knowledge of any versions that are known to all members of the community. This common knowledge can be passed between members via future circulating tokens or piggybacked on regular synchronization messages.

Once a version becomes common knowledge it remains so. Thus, although the common knowledge collected by a circulating token may be out of date, it still serves a useful purpose. A tombstone can be discarded when its version is included in the common knowledge. Common knowledge could also facilitate pruning inactive sites from version vectors.

For a large or poorly connected community, the time to circulate a token around the ring may be substantial.

Fortunately, this time only affects the rate at which tombstones can be discarded. The synchronization topology and schedule is unrelated to the ring structure. Therefore, a slow or long-disconnected member may cause tombstones to collect at other members but will not prevent other members from learning about updated items. If a site involuntarily retires from the community due to catastrophic failure, human intervention is needed to repair the ring since, as is well-known in the distributed computing literature, other sites cannot possibly distinguish a slow machine from a failed one.

5.3.2 *Conflict handlers*

In a world where machines are maintained by different community members running potentially different versions of the operating system, requiring consistent conflict handlers at all sites is problematic. Thus, WinFS adopted a scheme that allows sites to be configured with different conflict handlers without sacrificing eventual consistency. A site that has no locally registered conflict handlers simply logs any conflicts that it detects. In this case, the conflicting versions will continue propagating to other members and may eventually encounter a site that can resolve the conflict. For example, in a tree topology, the root site may be configured to resolve all conflicts. When a site resolves a conflict, it propagates the version produced by its local conflict handler to other community members, who may not even be aware that a conflict occurred. However, in some cases, members may independently detect and resolve a conflict, and their concurrent resolutions will, in turn, conflict. WinFS attaches additional metadata to resolution versions in order to deal with such "false" conflicts and avoid infinite chains of conflicting resolutions.

5.3.3 *Conflict resolution propagation*

The conflict resolution mechanism in WinFS deserves further elaboration. The use of application-registered conflict handlers that get invoked when conflicts are detected is similar to many other systems, but the configuration and outcome of such handlers is different. Commonly, conflicting updates are fully propagated to all sites and are resolved locally at each site. The individual sites are expected to detect and resolve conflicts in a consistent manner. This approach not only requires conflict handlers to have deterministic execution but also assumes that the same conflict handlers are installed at each site and are identically configured. In a world where machines are maintained by different community members running potentially different versions of the operating system, requiring consistent conflict handlers is problematic. Thus, WinFS adopted a scheme that accommodates

different conflict handlers without sacrificing eventual consistency.

When a WinFS site detects and resolves a conflict, it then propagates the version produced by its local conflict handler to other community members. Because WinFS only sends the latest version of each item during synchronization, the resolution is sent rather than the conflicting versions that it overwrites. The normal synchronization protocol ensures that the resolved version is eventually received by all members. Since the conflicting versions are included in the made-with knowledge of their resolved version, these versions will be added to other members' knowledge. In most cases, other members will not even be aware that a conflict occurred.

Different sites may be configured with different conflict handlers. A site that has no locally registered conflict handlers will simply log any conflicts that it detects. In this case, the conflicting versions will continue propagating to other members and may eventually encounter a site that can resolve the conflict. For example, in a tree topology, the root site may be configured to resolve all conflicts.

One consequence of this approach is that two members may independently detect and resolve a conflict. Concurrent resolutions will be assigned versions that, in turn, conflict with each other. This is not a problem as long as the conflict handlers on different sites produce identical item contents, since such "false" conflicts are ignored. However, if concurrent conflict handlers produce conflicting contents, then new mechanisms are needed for avoiding infinite chains of conflicting resolutions. The simple solution is to add a flag to each version indicating whether it was produced by an application update or a conflict handler. When conflicts are detected involving versions produced by conflict handlers, no further conflict handlers are invoked. Instead, the conflicting resolutions are either logged for human inspection or deterministically resolved using a "highest version wins" policy.

5.3.4 Overlapping synchronization communities

If numerous communities exist with overlapping content and memberships, forwarding versions of common items between communities could lead to large version vectors. Potentially, a member of a small community might end up with entries in its version vector for sites throughout the world. To avoid this scalability problem, the liaison that relays an updated change unit from one community to another "re-authors" the change by generating a new version if the original version was produced by a site that is not a

member of the destination community. To the members of the second community, it appears as though the change unit was updated by the cross-community liaison. This scheme ensures that a member's knowledge only contains versions generated by fellow community members, thereby restricting the size of its version vector to the size of the community. To avoid false conflicts from multiple members' reauthoring the same change unit, the current WinFS implementation permits only a single liaison between each pair of communities.

5.3.5 Range exceptions

As noted above, if using an unreliable transport over a lossy network, a member's knowledge could possibly contain a large number of exceptions. A new type of knowledge exception, called a *range exception*, was introduced to reduce this problem. Suppose that the conveyed changes, rather than being sent individually, are sent in batches that are organized by ranges of change unit identifiers. For example, the source target might send all change units with IDs in the range 1000-2000 as a batch along with the source's knowledge. The target atomically processes each batch and adds a range exception to its knowledge, indicating that for any change unit in the given range, it knows everything that the source knows. In other words, a range exception is more complex than ordinary knowledge exceptions since it contains a range of IDs and a version vector. However, one range exception takes the place of many individual knowledge exceptions and thus can significantly reduce the size of a member's knowledge if premature session termination is common.

5.3.6 Anti-entropy

To guard against software bugs or hardware failures that might corrupt a replica's data, the basic protocol as described in Section 5.2 is augmented with a "slow sync" or anti-entropy protocol [5] that compares two members' databases for consistency and repairs any discrepancies. This can be done on a periodic but infrequent basis or when suspicions arise.

The slow sync protocol is only run if the two members have identical knowledge, such as might occur after the members have requested changes from each other and are believed to be in a mutually consistent state. Each member computes a checksum over the entire contents of their local replica. A mismatch in checksums indicates a serious inconsistency. To recover from this condition, the members exchange and compare each item in their databases. A hierarchy of checksums could be utilized to narrow the source of the inconsistency and thereby reduce the number of items that must be exchanged, but such an optimization is

hardly warranted. While this slow sync protocol is aptly named, one hopes to rarely need it in practice.

6. Protocol validation

While the technical differences between WinFS and previous research systems are significant, there is another fundamental difference: WinFS is being incorporated into an operating system used by millions of people world-wide. To avoid serious design flaws and ensure that the WinFS replication protocol operates correctly in the myriad of scenarios in which it will be deployed, conventional testing techniques were not sufficient. A number of protocol validation activities were undertaken before and during the product development phase, including producing a formal specification, model checking, proving correctness, and simulating protocol aspects.

To gain confidence in the protocol's correctness, we produced a formal specification in TLA+, a language devised by Leslie Lamport for specifying and verifying concurrent systems [15]. The formal TLA+ specification was beneficial in that it precisely indicates the concurrent aspects of the protocol, namely which steps can be done in parallel and which need to be performed atomically. A major attraction of TLA+ is that its specifications can be run through the TLC model checker. We used the model checker to validate various protocol invariants, such as the fact that a member's local knowledge should only contain versions of items that it has already received. The model checker was also useful in debugging the TLA+ specification itself. Unfortunately, large-scale replicated systems are difficult to model check because of their large space of possible states. The model checker can only handle small models, which in this case means small numbers of replicas with small databases. One might reason that checking two-replica models is sufficient since the synchronization protocol operates pair-wise, involving exactly two members. However, some protocol anomalies, such as conflict resolution chains as described in Section 5.3.3, only occur in communities of four or more members with particular synchronization topologies.

The limitations of model checking caused us to produce a rigorous proof to validate the protocol's convergence guarantees. Specifically, we proved that the protocol (a) reliably detects concurrent updates as conflicts, (b) never reports false conflicts, and (c) ensures that each version of a change unit is eventually received by all community members or is overwritten by a later version (assuming good connectivity and uniform trust). Details of the proof are beyond the scope of this paper.

Finally, we used simulation to better understand the dynamic behavior of the protocol in diverse situations that are difficult to test, such as large numbers of replicas or unreliable networks. A correctly operating protocol is not necessarily a well-behaving protocol. For instance, while the proof indicated that replicas are guaranteed to converge to a consistent state, it could not indicate how long it takes to reach convergence for a specific number of members on a particular synchronization schedule. We developed a custom simulator that quickly allows us to collect statistics of the protocol's execution for variations in update workloads, numbers of replicas, topologies, conflict resolution policies, and network characteristics.

Although comprehensive simulation results cannot be presented due to space limitations, we include some sample results obtained from our study. Figure 5 shows the convergence time for two synchronization topologies: a fully connected clique and a star topology. In this simulation, each replica chooses one neighbor at random with which to synchronize once per round. The convergence time is the average number of synchronization rounds needed to fully propagate an update. In the clique topology, convergence time increases logarithmically as the number of replicas increases.

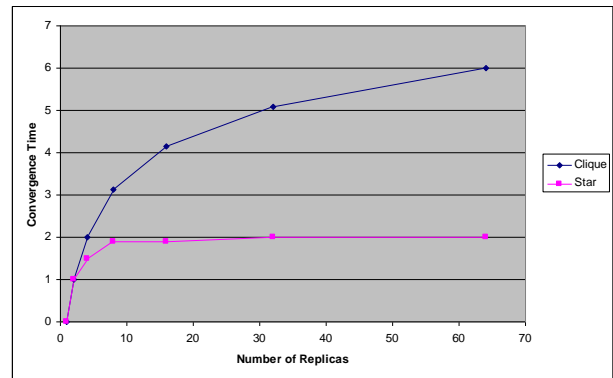


Figure 5. Convergence time vs. replication factor.

Figure 6 illustrates the effect of update rate on message traffic for an eight-replica clique. An update site is chosen at random in each round with the x-axis denoting the number of updates performed per 100 database items. The y-axis lists the total number of ConveyChanges messages sent for each update. The top (flat) line is for a workload where every operation creates a new item, and hence updates never overwrite each other. It confirms that each update is sent exactly seven times, once to each non-originating replica, independent of the rate at which items are created. This line also represents the traffic generated by

systems that send logged writes, such as Bayou. When updates are allowed to the same items, thereby overwriting previous versions, the lower line shows that the number of messages per update decreases as the update rate increases. At 20% updates, the message traffic decreases by 13% compared to the non-overwrite traffic. As the availability of replicas decreases, a similar phenomenon happens since a longer convergence time causes more overwrites. At 80% availability for 20% updates, the traffic decrease is 18%.

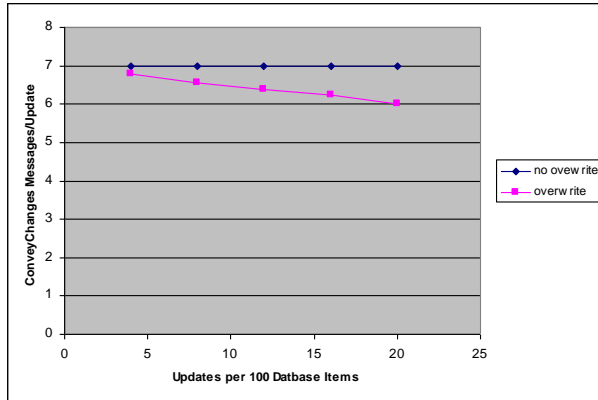


Figure 6. Traffic per update vs. update rate.

7. Implementation and performance

WinFS Sync was implemented in Microsoft Windows using SQL Server 2005. The implementation, consisting of approximately 50,000 lines of C# code, runs partly as stored procedures in SQL Server and partly as library code used by sync-aware applications.

The WinFS Sync implementation followed a key guiding principle: *the replication process should have minimal impact on the performance of local applications*. In particular, no extra database writes should be needed to update replication-specific metadata during the course of running normal application transactions. This is a key reason why the replication design avoided the use of a persistent write log. To achieve this goal, the implementation relied on features of the SQL Server database manager as described below: stored procedures, timestamps, and snapshot isolation.

Schemas for WinFS item types are compiled to produce relational database tables for storing item properties and stored procedures for modifying this data. When updating a change unit, a stored procedure assigns it a timestamp from a database-wide 64-bit counter, which is incremented automatically by the database manager whenever a new row is added to the table or an existing row is modified. The local site's

GUID coupled with the change unit's update timestamp serves as the change unit's version. This approach guarantees that new versions are generated whenever items are updated regardless of the API through which those updates are performed.

The local per-replica knowledge is stored in database tables and is updated during replication. This knowledge, however, is not explicitly updated when the database is modified by a local operation. When needed at sync time, such as when requesting changes or providing made-with and learned knowledge, the member's component of its own knowledge vector is obtained from the latest value of the database's timestamp counter. In other words, the member's knowledge of its own updates is simply the last version that it generated.

One consequence of this scheme is that new versions are generated for both updates that originate locally and those received from other members during sync. This does not affect the correctness of the protocol. The implementation maintains a table mapping local versions to the originating versions for change units updated by other sites. This versioning scheme is beneficial when enumerating changes.

When a member receives a RequestChanges message, it runs a query to determine the change units whose versions are not covered by the requestor's knowledge. For efficiency, the database maintains an index on timestamps. The source member could execute a SQL query with clauses for each site in the target member's knowledge vector, such as:

```
SELECT * FROM table
WHERE (site="A"
      AND ts>TargetKnowledge["A"])
OR (site="B"
    AND ts>TargetKnowledge["B"])
OR ...
```

Unfortunately, queries of this sort with disjunctive clauses are inefficient since the index must be accessed separately for each clause. Thus, our implementation uses a different query to enumerate changes, one utilizing the local timestamp field:

```
SELECT * FROM table
WHERE ts > TargetKnowledge["mySite"]
```

This query returns all of the change units that have been updated since the local site directly or indirectly performed synchronization with the requesting member. For communities with a moderate number of members, this query is substantially more efficient. However, it may return change units that the requestor has already received from other members. Thus, the replication code must explicitly check each returned change unit to ensure that it is not already covered by

the requestor's knowledge. Generally, the benefits of obtaining recently updated change units in one index scan more than offsets the cost of checking the query results before sending them to the requestor.

To ensure that these change enumeration queries do not interfere with concurrent application updates, the queries are run as a transaction with the *snapshot isolation* level. Essentially, the replication process is presented with a read-only snapshot of the database. SQL Server provides this snapshot efficiently using copy-on-write techniques. Using snapshot isolation not only allows the replication protocol to operate at its leisure without holding locks but also ensures that the learned knowledge reported in the CompleteSession message is consistent with the updates that were conveyed. Moreover, it permits synchronization sessions with different partners to proceed in parallel without interference, such as when multiple remote members request local changes at the same time.

Measurements of a synchronization session between two members are presented in Figure 7. In the measured configuration, each member has a WinFS store containing 50,000 total items. The community folder being synchronized contains 1,500 of these items. Each item of type "Person" consists of three change units and contains around a kilobyte of data. The members involved in synchronization reside on the same machine in order to factor out the network transmission costs; when operating over a local area network, the synchronization time is actually less due to overlapping execution by the source and target. In this scenario, the source creates 60 new items and 60 new links to add these items to the community folder. Thus, 180 change units and 60 links are sent during synchronization. The target commits changes in batches of 30.

Enumerate changes	0.73 (20%)
Apply changes	2.75 (74%)
Miscellaneous	0.24 (6%)
TOTAL	3.72 seconds

Figure 7. Performance of end-to-end synchronization.

Of the time spent enumerating changes in the source replica, about half is spent executing the query to return new change units, and the other half is spent retrieving the selected items. Most of the time elapses as the target creates new items in its database. Updating or deleting an item takes as long as creating an item. The null synchronization time, that is, the

time to synchronize when no changes need to be sent, is 0.2 seconds. The space overhead for storing replication metadata in this test database is only about 7%, even though change units are relatively small. Performance has been measured for scenarios involving many different workloads and system configurations; unfortunately, limited space in this paper prevents us from presenting additional measurements.

8. Conclusions

The WinFS replication protocol was designed to support a wide class of applications and usage scenarios. To support offline access on disconnected laptops and mobile devices, as well as to scale to large numbers of replicas, WinFS provides an update-anywhere replication model. It ensures that replicas eventually converge using a peer-to-peer, knowledge-driven, state-based synchronization protocol. Although the WinFS replication model is similar to several research prototypes, WinFS differs in its management of knowledge, its treatment of independent communities, its handling of conflicts, its collection of deletion tombstones, and most significantly, its ability to function efficiently without a write log.

Because of its reliance on only local per-replica knowledge, the WinFS replication protocol can readily tolerate failures of the network and synchronization partners. If the protocol gets interrupted for any reason, during the next synchronization it automatically picks up where it left off even if synchronizing with a different partner. The protocol is also robust against lost or reordered network traffic, and thus can operate over a variety of transport protocols.

The flexibility of the protocol allows administrators and applications to establish synchronization topologies and schedules tailored for particular sharing patterns. Many of our applications have adopted a full-mesh topology, though in some cases, a hub-and-spoke or hierarchical arrangement of community members may be desirable. Mobile devices can synchronize opportunistically when connected.

Bandwidth is used efficiently since each updated change unit is sent at most once to each replica, and obsolete versions are never sent. To initiate synchronization, a member need only send its current knowledge in the form of a version vector. Even in a community of 5,000 active members, a member's version vector occupies around 100 Kbytes. Given this compact knowledge representation, members may choose to synchronize frequently to maintain relatively strong consistency, even over low-bandwidth networks.

Storage overheads are also low and independent of the update traffic, allowing WinFS to scale to larger numbers of replicas than previous protocols, larger update rates, and larger databases with fine-grained objects. Each member maintains a single version vector for its knowledge, which is used not only to enumerate changes during synchronization but also to detect concurrent, and therefore conflicting, updates. The metadata for each change unit consists solely of a 24-byte version. This low overhead allows schema designers to define change units that are as small as a single-bit read flag on an e-mail message.

Application designers can extend the features of WinFS by defining new schemas (that is, new data types for items) and providing custom conflict handlers. WinFS Sync automatically replicates all types of items and invokes type-specific conflict handlers when conflicts are detected.

Our experience with designing the WinFS replication protocol indicates that a variety of techniques are useful for validating a system design, including both informal and formal specification, model checking, correctness proofs, and simulation. All of these techniques have different strengths and limitations, and therefore complement each other. Importantly, these activities could be performed early in the development process when the design was readily evolving. Collectively, they have improved our confidence in the correctness of the WinFS replication protocol.

9. Acknowledgements

The replication protocol described in this paper was designed in collaboration with the WinFS product team within Microsoft. Yuan Yu (MSR) and Harry Li (University of Texas) were mainly responsible for developing the TLA+ specification, with help from Leslie Lamport (MSR), and for producing the proof of convergence. Yuan Yu also wrote the TLC model checker. Rebecca Isaacs (MSR Cambridge) and Vuk Ercegovic (University of Wisconsin) developed the protocol simulator and conducted the simulations. Raghu Ramakrishnan (University of Wisconsin) helped guide the simulation studies. Thanks also go to those who reviewed drafts of this paper.

10. References

- [1] J. E. Allchin. A suite of robust algorithms for maintaining replicated data using weak consistency conditions. *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983.
- [2] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems* 9(4):596-615, December 1984.
- [3] A. D. Birrell, R. Levin M. D. Schroeder, and R. M. Needham. Grapevine: an exercise in distributed computing. *Communications of the ACM* 25(4):260-274, April 1982.
- [4] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys* 17(3):341-370, September 1985.
- [5] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. *Proceedings Symposium on Principles of Distributed Computing*, Vancouver, B. C., Canada, August 1987, pages -12.
- [6] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. *Proceedings SIGACT-SIGMOD Symposium on Principles of Database Systems*, March 1982.
- [7] D. K. Gifford. Weighted voting for replicated data, *Proceedings Seventh ACM Symposium on Operating Systems Principles*, Pacific Grove, California, December 1979, page 150-162.
- [8] R. A. Golding. A weak-consistency architecture for distributed information services. *Computing Systems* 5(4):379-405, Fall 1992.
- [9] R. G. Guy, G. J. Popek, and T. W. Page, Jr. Consistency algorithms for optimistic replication. *Proceedings First International Conference on Network Protocols*, October 1993.
- [10] L. Kalwell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif. Replicated document management in a group communication system. *Proceedings Conference on Computer-Supported Cooperative Work*, Portland, Oregon, September 1988.
- [11] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems* 10(1): 3-25, February 1992.
- [12] P. Kumar and M. Satyanarayanan. Flexible and safe resolution of file conflicts. *Proc. USENIX Winter 1995 Conference on Unix and Advanced Computing Systems*, New Orleans, LA, Jan. 1995, pages 16-20.
- [13] B. Lampson. Designing a global name service. *Proceedings ACM Symposium on Principles of Distributed Computing*, Alberta, Calgary, Canada, 1986, pages 1-10.
- [14] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transaction on Computer Systems*, 10(4):360, November 1992.
- [15] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, MA, 2003.
- [16] 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.
- [17] D. S. Parker Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* 9(3): 240-247, May 1983.
 - [18] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. *Proceedings ACM Symposium on Operating Systems Principles (SOSP)*, Saint Malo, France, October 1997, pages 288-301.
 - [19] D. Ratner, P. Reiher, G. J. Popek, and G. H. Kuenning. Replication requirements in mobile environments. *Mobile Networks and Applications* 6:525-533, 2001.
 - [20] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. *Proceedings Summer USENIX Conference*, June 1994, pages 183-195.
 - [21] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, to appear March 2005. Also available as Microsoft Technical Report MSR-TR-2003-60, September 2003.
 - [22] S. K. Sarin and N. A. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering* SE-13, No. 1 (January 1987), pp. 39-47.
 - [23] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel and D. C. Steere. Coda: a highly available file system for a distributed workstation environment. *IEEE Transactions on Computers* 39(4):447-459, 1990.
 - [24] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing* 7(3):149-174, 1994.
 - [25] D. Terry, A. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. Welch. Session guarantees for weakly consistent replicated data. *Proceedings International Conference on Parallel and Distributed Information Systems (PDIS)*, September 1994, pages 140-149.
 - [26] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Proceedings ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995, pages 172-182.
 - [27] W. Vogels, R. van Renesse, and K. Birman. The power of epidemics: robust communication for large-scale distributed systems. *ACM SIGCOMM Computer Communications Review* 33(1):131-135, January 2003.
 - [28] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, Bretton Woods, New Hampshire, 1983, pages 49-70.
 - [29] G. T. J. Wu and A. J. Bernstein. Efficient solutions to the replicated log and dictionary problems. *Proceedings Third ACM Symposium on Principles of Distributed Computing*, Vancouver, British Columbia, Canada, August 1984, pages 233-242.
 - [30] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems* 20(3):239-282, August 2002.