Session Guarantees for Weakly Consistent Replicated Data

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike J. Spreitzer, Marvin M. Theimer, and Brent B. Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304

Abstract

Four per-session guarantees are proposed to aid users and applications of weakly consistent replicated data: Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes. The intent is to present individual applications with a view of the database that is consistent with their own actions, even if they read and write from various, potentially inconsistent servers. The guarantees can be layered on existing systems that employ a read-any/write-any replication scheme while retaining the principal benefits of such a scheme, namely high-availability, simplicity, scalability, and support for disconnected operation. These session guarantees were developed in the context of the Bayou project at Xerox PARC in which we are designing and building a replicated storage system to support the needs of mobile computing users who may be only intermittently connected.

1. Introduction

Techniques for managing weakly consistent replicated data have been employed in a variety of systems [4,10,12,17,19,20]. Such systems are characterized by the lazy propagation of updates between servers and the possibility for clients to see inconsistent values when reading data from different replicas. Weakly consistent systems are popular due to their high-availability, good scalability, and simplicity of design. These advantages arise from the ability to allow reads and writes to be performed with little or no synchronization among replicas. For example, Grapevine [4], the first widely used computing system with weak consistency, used a *read-any/write-any* replication scheme, in which clients could read from any server and could write to any server.

More recently, the use of weakly consistent replicated data has been driven by the needs of mobile computing applications [11,13,22]. For example, disconnected users may want to read and update data copied onto their portable computers even if they did not have the foresight to lock it before either a voluntary or an involuntary disconnection occurred. Also, the presence of slow or expensive communications links in the system can make maintaining closely synchronized copies of data difficult or uneconomical.

Unfortunately, the lack of guarantees concerning the ordering of read and write operations in weakly consistent systems can confuse users and applications, as reported in experiences with Grapevine [21]. A user may read some value for a data item and then later read an older value. Similarly, a user may update some data item based on reading some other data, while others read the updated item without seeing the data on which it is based. A serious problem with weakly consistent systems is that inconsistencies can appear even when only a single user or application is making data modifications. For example, a mobile client of a distributed database system could issue a write at one server, and later issue a read at a different server. The client would see inconsistent results unless the two servers had synchronized with one another sometime between the two operations.

In this paper, we introduce *session guarantees* that alleviate this problem of weakly consistent systems while maintaining the principle advantages of read-any/write-any replication. A *session* is an abstraction for the sequence of read and write operations performed during the execution of an application. Sessions are not intended to correspond to atomic transactions that ensure atomicity and serializability. Instead, the intent is to present individual applications with a view of the database that is consistent with their own actions, even if they read and write from various, potentially inconsistent servers. We want the results of operations performed in a session to be consistent with the model of a single centralized server, possibly being read and updated concurrently by multiple clients.

To achieve this, we propose four guarantees that can be applied independently to the operations belonging to a session. The guarantees are summarized as follows:

Read Your Writes - read operations reflect previous writes.

Monotonic Reads - successive reads reflect a non-decreasing set of writes.

Writes Follow Reads - writes are propagated after reads on which they depend.

Monotonic Writes - writes are propagated after writes that logically precede them.

These properties are "guaranteed" in the sense that either the storage system ensures them for each read and write operation belonging to a session, or else it informs the calling application that the guarantee cannot be met.

The above guarantees can easily be layered on top of a weakly-consistent replicated data system. Each read or write operation is performed at a single server, and the writes are propagated to other servers in a lazy fashion. To ensure that the guarantees are met, the servers at which an operation can be performed must be restricted to a subset of available servers that are sufficiently up-to-date.

Because enforcement of the guarantees restricts the set of servers that may be used within a session, requesting a guarantee can have an adverse impact on availability. Applications must make a trade-off between availability and consistency. For this reason, guarantees can be requested individually on a per-session basis. Requests for one or more of the guarantees within a session have no affect on the availability seen by applications that are using other sessions or on applications that require no guarantees.

The particular guarantees we present were derived primarily from the needs of applications being investigated in the Bayou project at Xerox PARC. Bayou is a weakly consistent replicated storage system we are designing and building to support collaborative applications running in a mobile computing environment. The examples is this paper discuss some of these applications and their possible use of the guarantees. We do not claim that the set of four guarantees is complete in any way; variations are possible and may be suggested by new application domains.

This paper presents precise definitions for each of the four guarantees, gives examples of their usage by various applications, and shows how to implement them efficiently using version vectors. It also demonstrates how the guarantees can be added to existing systems, such as Coda, with minimal or no changes to the server implementation. Finally, we discuss their effect on availability, raise issues requiring further research, and compare this work to other proposals for adding increased semantics to weakly consistent replicated data.

2. Data storage model and terminology

In order to provide concrete definitions of the session guarantees, it is first necessary to present the assumptions made about the underlying replicated storage system and the terminology used throughout the paper.

The most basic assumption concerns the existence of a weakly consistent replicated storage system to which the guarantees will be added. Such a system consists of a number of *servers* that each hold a full copy of some replicated *database* and *clients* that run applications desiring access to the database. The session guarantees presented are most applicable to systems in which clients and servers may reside on separate machines and a client accesses different servers over time. For example, a mobile client may choose servers based on which ones are available in its region and can be accessed most cheaply.

The term “database” is not meant to imply any particular data model or organization, nor are the techniques presented in this paper specific to any data model. A database is simply a set of *data items*, where a data item could be anything from a conventional file to a tuple in a relational database. For simplicity, the discussion in this paper focuses on a single, fully replicated database, though the guarantees are equally applicable to a system that manages a collection of replicated databases.

Two main operations on a database are considered: *Read* and *Write*. The Read operation represents a query over the contents of the database. A Read could be a simple retrieval operation such as “return the contents of file foo” or a complicated query such as “return the names of all employees who make more than their boss.” The Write operation updates the database. A Write may involve creating, modifying, or deleting data items. It may also represent a transaction that atomically updates multiple items in a server’s database. The definition and implementation of session guarantees is unaffected by whether Writes are simple database updates or more complicated atomic transactions. Each Write has a globally unique identifier, called a “*WID*”. The server that first accepts the Write, for instance, might be responsible for assigning its WID.

Read and Write operations may be performed at any server or set of servers. Our guarantees are presented assuming that each Read or Write is executed against a single server’s copy of the database. That is, for the most part, we discuss variants of a read-any/write-any replication scheme. However, the guarantees could also be used in systems that read or write multiple copies, such as all of the available servers in a partition [5].

We define DB(S,t) to be the ordered sequence of Writes that have been received by server S at or before time t. If t is known to be the current time, then it may be omitted leaving DB(S) to represent the current contents of the server's database. Conceptually, server S creates its copy of the database, which it uses to answer Read requests, by starting with an empty database and applying each Write in DB(S) in the given order. In practice, a server is allowed to process the Writes in a different order as long as their effect on the database is unchanged. The order of Writes in DB(S) does not necessarily correspond to the order in which server S first received the Writes (as discussed below).

Weak consistency permits database copies at different servers to vary. That is, DB(S1,t) is not necessarily equivalent to DB(S2,t) for two servers S1 and S2. However, practical systems generally desire *eventual consistency* in which servers converge towards identical database copies in the absence of updates. Eventual consistency relies on two properties: *total propagation* and *consistent ordering*. We assume that the replicated system provides eventual consistency and thus includes mechanisms to ensure these two properties as follows.

Writes are propagated among servers by a process called *anti-entropy*, also referred to in some papers as rumor mongering, lazy propagation, or update dissemination [1,6]. Anti-entropy ensures that each Write is eventually received by each server. In other words, for each Write W there exists a time t such that W is in DB(S,t) for each server S. This paper makes no further assumptions about the anti-entropy protocol, the frequency with which it happens, the policy by which servers choose anti-entropy partners, or other characteristics of the anti-entropy process.

Additionally, all servers must apply non-commutative Writes to their databases in the same order. Let *WriteOrder(W1,W2)* be a boolean predicate indicating whether Write W1 should be ordered before Write W2. The system ensures that if WriteOrder(W1,W2) then W1 is ordered before W2 in DB(S) for any server S that has received both W1 and W2. In a strongly consistent system, WriteOrder would reflect the order in which individual Writes or transactions are committed. In an eventually consistent system, servers could use any of a variety of techniques to agree upon the order of Writes. For example, the Grapevine system orders Writes by their origination timestamp [4]. Using timestamps to determine the Write order does not imply that servers have synchronized clocks since there is no requirement that Writes be ordered by the actual time at which they were performed. This paper makes no assumption about how servers agree on the ordering of Writes or about how servers make their copies of the database conform to this ordering. It only assumes that the system has some means by which Writes are ordered consistently at every server, as required for eventual consistency, and uses the WriteOrder predicate to represent this ordering.

Finally, weakly consistent systems often allow conflicting Writes to occur. That is, two clients may make concurrent and incompatible updates to the same data item. Existing systems resolve conflicting Writes in different ways. In some systems the Write order may determine which Write "wins", while other systems rely on humans to resolve detected conflicts. How the system detects and resolves Write conflicts is important to its users but has no impact on our session guarantees.

3. Read/Write guarantees

This section precisely defines the four session guarantees in terms of server state and Read/Write operations performed at servers. Although session guarantees are directly applicable to systems with client caching, we present the guarantees as if clients do not cache data that they read. An application may use one or more sessions to control the scope of the guarantees that it requests. This implies that each Read and Write operation issued by the application is associated with a session. How this association is made and whether sessions can be shared across programs, processes, or machines is an implementation detail that is left unspecified.

3.1 Read Your Writes

The Read Your Writes guarantee is motivated by the fact that users and applications find it particularly confusing if they update a database and then immediately read from the database only to discover that the update appears to be missing. This guarantee ensures that the effects of any Writes made within a session are visible to Reads within that session. In other words, Reads are restricted to copies of the database that include all previous Writes in this session. Specifically:

RYW-guarantee: If Read R follows Write W in a session and R is performed at server S at time t, then W is included in DB(S,t).

Applications are not guaranteed that a Read following a Write to the same data item will return the previously written value. In particular, Reads within the session may see other Writes that are performed outside the session.

Consider a couple of examples to illustrate how the RYW-guarantee might be used in practice.

Example 1. After changing his password, a Grapevine user would occasionally type the new password and receive an "invalid password" response. This annoying problem would arise because the login process contacted a server to which the new password had not yet propagated. The problem is not specific to Grapevine but could occur in any weakly consistent system that manages passwords. It can be solved cleanly by having a session per user in which the RYW-guarantee is provided. Such a session should be created for each new user and must exist for the lifetime of the user's account. By performing updates to the user's password as well as checks of this password within the session, users can use a new password without regard for the extent of its propagation. The RYW-guarantee ensures that the login process will always read the most recent password. Notice that this application requires a session to persist across logouts and machine reboots.

Example 2. Consider a user whose electronic mail is managed in a weakly consistent replicated database. As the user reads and deletes messages, those messages are removed from the displayed "new mail" folder. If the user stops reading mail and returns sometime later, she should not see deleted messages reappear simply because the mail reader refreshed its display from a different copy of the database. The RYW-guarantee can be requested within a session used by the mail reader to ensure that the effects of any actions taken, such as deleting a message or moving a message to another folder, remain visible.

3.2 Monotonic Reads

The Monotonic Reads guarantee permits users to observe a database that is increasingly up-to-date over time. It ensures that Read operations are made only to database copies containing all Writes whose effects were seen by previous Reads within the session.

Intuitively, a set of Writes completely determines the result of a Read if the set includes "enough" of the database's Writes so that the result of executing the Read against this set is the same as executing it against the whole database. Specifically, we say a Write set WS is *complete* for Read R and DB(S,t) if and only if WS is a subset of DB(S,t) and for any set WS2 that contains WS and is also a subset of DB(S,t), the result of R applied to WS2 is the same as the result of R applied to DB(S,t).

Let *RelevantWrites(S,t,R)* denote the function that returns the smallest set of Writes that is complete for Read R and DB(S,t). Some complete set exists since DB(S,t) is itself complete for any Read. If the smallest complete set is not unique, the tie may be broken in an arbitrary, but deterministic manner. Intuitively, RelevantWrites(S,t,R) is a smallest set that is "enough" to completely determine the result of R. Given this function, the Monotonic Reads guarantee can be defined precisely as follows:

MR-guarantee: If Read R1 occurs before R2 in a session and R1 accesses server S1 at time t1 and R2 accesses server S2 at time t2, then RelevantWrites(S1,t1,R1) is a subset of DB(S2,t2).

Example 3. A user's appointment calendar is stored online in a replicated database where it can be updated by both the user and automatic meeting schedulers. The user's calendar program periodically refreshes its display by reading all of today's calendar appointments from the database. If it accesses servers with inconsistent copies of the database, recently added (or deleted) meetings may appear to come and go. The MR-guarantee can effectively prevent this since it disallows access to copies of the database that are less current than the previously read copy.

Example 4. Once again, consider a replicated electronic mail database. The mail reader issues a query to retrieve all new mail messages and displays summaries of these to the user. When the user issues a request to display one of these messages, the mail reader issues another Read to retrieve the message's contents. The MR-guarantee can be used by the mail reader to ensure that the second Read is issued to a server that holds a copy of the message. Otherwise, the user, upon trying to display the message, might incorrectly be informed that the message does not exist.

3.3 Writes Follow Reads

The Writes Follow Reads guarantee ensures that traditional Write/Read dependencies are preserved in the ordering of Writes at all servers. That is, in every copy of the database, Writes made during the session are ordered after any Writes whose effects were seen by previous Reads in the session.

WFR-guarantee: If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2) and WriteOrder(W1,W2).

This guarantee is different in nature from the previous two guarantees in that it affects users outside the session. Not only does the session observe that the Writes it performs occur after any Writes it had previously seen, but also all other clients will see the same ordering of these Writes regardless of whether they request session guarantees.

Example 5. Imagine a shared bibliographic database to which users contribute entries describing published papers. Suppose that a user reads some entry, discovers that it is inaccurate, and then issues a Write to update the entry. For instance, the person might discover that the page numbers for a paper are wrong and then correct them with a Write such as "UPDATE bibdb SET pages = '45-53' WHERE bibid = 'Jones93'." The WFR-guarantee can ensure that the new Write updates the previous bibliographic entry at all servers.

The WFR-guarantee, as defined, associates two constraints on Write operations. A constraint on Write order ensures that a Write properly follows previous relevant Writes in the global ordering that all database replicas will eventually reflect. A constraint on propagation ensures that all servers (and hence all clients) only see a Write after they have seen all the previous Writes on which it depends. Example 5 requires both of these properties. Some applications, however, may require only one of them. For such applications, systems may wish to provide relaxed variants of the Writes Follow Reads guarantee, one to guarantee how Writes are ordered and the other to guarantee how they propagate among servers:

WFRO-guarantee: If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then WriteOrder(W1,W2) for any W1 in RelevantWrites(S1,t1,R1).

WFRP-guarantee: If Read R1 precedes Write W2 in a session and R1 is performed at server S1 at time t1, then, for any server S2, if W2 is in DB(S2) then any W1 in RelevantWrites(S1,t1,R1) is also in DB(S2).

Example 6. Consider a weakly consistent replicated bulletin board database that requires users to post articles or to reply to articles by performing database Writes. The WFRP-guarantee can be used within this system to ensure that users see the replies to a posted article only after they have seen the original. A user who replies to an article must simply issue the reply in the same session as used to read the article being replied to. Users who are only reading articles need not request any guarantees. While the full WFR-guarantee would suffice for this application, the ordering property is not necessary since the posting of an article and the posting of a reply are commutative operations as far as the database is concerned.

Example 7. Let's revisit the shared bibliographic database discussed in Example 5. Suppose that Write operations always contain complete bibliographic entries rather than partial updates. For instance, to update the page numbers in an entry, a user would read the previous entry, correct the "pages" field and then Write back the full updated entry including all of the unmodified fields. In this case, the WFRO-guarantee could be used instead of the WFR-guarantee. The reason is that permitting users to see the newest version of a bibliography entry is acceptable even if older versions have not yet reached the server they are using.

3.4 Monotonic Writes

The Monotonic Writes guarantee says that Writes must follow previous Writes within the session. In other words, a Write is only incorporated into a server's database copy if the copy includes all previous session Writes; the Write is ordered after the previous Writes.

MW-guarantee: If Write W1 precedes Write W2 in a session, then, for any server S2, if W2 in DB(S2) then W1 is also in DB(S2) and WriteOrder(W1,W2).

This guarantee provides assurances that are relevant both to the user of a session as well as to users outside the session. As with the Writes Follow Reads guarantee, one could define two variants that allow applications to separately control Write order and Write propagation.

Example 8. The MW-guarantee could be used by a text editor when editing replicated files to ensure that if the user saves version N of the file and later saves version N+1 then version N+1 will replace version N at all servers. In particular, it avoids the situation in which version N is written to some server and version N+1 to a different server and the versions get propagated such that version N is applied after N+1.

Example 9. Consider a replicated database containing software source code. Suppose that a programmer updates a library to add functionality in an upward compatible way. This new library can be propagated to other servers in a lazy fashion since it will not cause any existing client software to break. However, suppose that the programmer also updates an application to make use of the new library functionality. In this case, if the new application code gets written to servers that have not yet received the new library, then the code will not compile successfully. To avoid this potential problem, the programmer can create a new session that provides the MW-guarantee and issue the Writes containing new versions of both the library and application code within this session.

Under certain circumstances, the Monotonic Writes guarantee may be implied by the combination of Writes Follow Reads and Read Your Writes. For instance, suppose that in Example 8 the text editor always Reads version N before producing version N+1. In this case, Writes Follow Reads is sufficient to ensure the proper ordering of versions. In general, this does not hold. Consider a session that consists of the following series of operations: W1 R W2. The WFR-guarantee says only that W2 will follow the relevant Writes of R. If some other application were to submit a Write Wx between W1 and R, and Wx overwrites the data written by W1, W1 would be absent from the relevant Writes of R. Since there is no ordering between W1 and Wx, the ordering imposed by Monotonic Writes on W1 and W2 would be lost.

4. Providing the guarantees

Techniques for implementing the four guarantees are presented in this section. These simple techniques are then refined into more practical implementations in a later section. The emphasis here is on devising correct implementations of the session guarantees and on precisely stating the requirements placed on the underlying replicated storage system. Problems concerning the amount of bandwidth used between clients and servers, the storage space needed by clients and servers, and the computation costs involved are addressed in Section 5. Also, the implementations ignore issues of client caching. This allows us to present the basic implementation of session guarantees without the obscuring details of dealing with caches that are too small or get flushed at inopportune times.

The implementations require only minor cooperation from the servers that process Read and Write operations. Specifically, a server must be willing to return information about the unique identifier (WID) assigned to a new Write, the set of WIDs for Writes that are relevant to a given Read, and the set of WIDs for all Writes in its database.

The burden of providing the guarantees lies primarily with the *session manager* through which all of a session's Read and Write operations are serialized. The session manager can be considered a component of the client stub that mediates communication with available servers. For each session, it maintains two sets of WIDs:

read-set = set of WIDs for the Writes that are relevant to session Reads

write-set = set of WIDs for those Writes performed in the session

An alternative to recording WIDs would be to keep copies of the Writes themselves. These Writes could then be passed from clients to servers to bring a server sufficiently up-to-date before it could process the session's Read and Write requests. This option was rejected because allowing Writes to pass between servers via clients could violate the propagation guarantees associated with Writes Follow Reads and Monotonic Writes.

Providing the Read Your Writes guarantee involves two basic steps. Whenever a Write is accepted by a server, its assigned WID is added to the session's write-set. Before each Read to server S at time t, the session manager must check that the write-set is a subset of DB(S,t). This check could be done on the server by passing the write-set to the server or could be done on the client by retrieving the server's list of WIDs. The session manager can continue trying available servers until it discovers one for which the check succeeds. If it cannot find a suitable server, then it reports that the guarantee cannot be provided.

Providing the Monotonic Reads guarantee is similar in that before each Read to server S at time t, the session manager must ensure that the read-set is a subset of DB(S,t). Additionally, after each Read R to server S, the WIDs for each Write in RelevantWrites(S,t,R) should be added to the session's read-set. This presumes that the server can compute the relevant Writes and return this information along with the Read result.

Unlike the Read Your Writes and Monotonic Reads guarantees, implementing the Writes Follow Reads and Monotonic Writes guarantees requires placing two additional, but reasonable, constraints on the servers' behavior:

C1. When a server S accepts a new Write W2 at time t, it ensures that WriteOrder(W1,W2) is true for any W1 already in DB(S,t). That is, new Writes are ordered after Writes that are already known to a server.

C2. Anti-entropy is performed such that if W2 is propagated from server S1 to server S2 at time t then any W1 in DB(S1,t) such that WriteOrder(W1,W2) is also propagated to S2.

Actually, these requirements as stated are slightly stronger than needed for the guarantees. Strictly speaking, the two conditions discussed above must hold for any Write W1 in the session's read-set or write-set rather than for any Write in DB(S,t). This subtle distinction is not likely to have a practical consequence since the weaker requirements would require a server to keep track of clients' read-sets and write-sets. The stronger requirements allow a server's behavior to be independent of the session state maintained by clients.

Fortunately, these conditions are met in many systems providing weakly consistent replicated data. In general, ordering new Writes after previous Writes, as requested in C1, is a desirable system property. Moreover, it is easy to ensure. If the WriteOrder predicate is computed by comparing timestamps, for instance, then a new Write must simply be timestamped later than previous Writes received by the server. Constraint C2, in practice, means that either servers atomically transfer their complete databases during anti-entropy or else they transfer Writes according to the order that these Writes were applied to their database. As an example of a popular system that satisfies C2, the latest version of Coda's server-to-server "resolution" protocol brings the file volumes maintained by two servers into a consistent state as an atomic action (according to information provided by Jay Kistler). The Grapevine system, however, does not meet this requirement because of its use of electronic mail for propagating updates and the fact that mail messages may get reordered during delivery.

Given these constraints on servers' behavior, the Writes Follow Reads guarantee can be provided as follows. As with Monotonic Reads, each Read R to server S at time t results in RelevantWrites(S,t,R) being added to the session's read-set. Before each Write to server S at time t, the session manager checks that this read-set is a subset of DB(S,t).

Providing the Monotonic Writes also involves two steps. In order for a server S to accept a Write at time t, the server's database, DB(S,t), must include the session's write-set. Also, whenever a Write is accepted by a server, its assigned WID is added to the write-set.

The four guarantees can be readily provided together or in any combination. Table 1 summarizes for each guarantee what operation causes the session state to be updated and what operation requires checking this state to find a suitable server.

Table 1: Read/Write guarantees

Guarantee	session state updated on	session state checked on
Read Your Writes	Write	Read
Monotonic Reads	Read	Read
Writes Follow Reads	Read	Write
Monotonic Writes	Write	Write

5. Practical implementation of the guarantees

This section introduces the use of version vectors to obtain more efficient implementations of the four guarantees. The implementations discussed in the previous section follow straightforwardly from the definitions of the guarantees, but have several practical problems:

- The session state, i.e. the set of WIDs maintained for a session, could get large.
- The set of relevant WIDs returned from a Read operation could get large.
- The set of WIDs checked on a Read or Write operation could get large.
- The information used by servers to record the Writes they have seen could be large.
- Finding a suitable server, including checking that a server's database contains all of the necessary Writes, could be expensive.
- The bookkeeping required of servers to determine the relevant Writes for a Read could be excessive.

Version vectors, which were introduced in Locus [18] and are used by several systems to detect Write conflicts, can alleviate many of these problems. A version vector is a sequence of <server, clock> pairs, one for each server. The *server* portion is simply a unique identifier for a particular copy of the replicated database. The *clock* is a value from the given server's monotonically increasing logical clock. The only constraint on this logical clock is that it must increase for each Write accepted by the server; for instance, it could be a Lamport clock [15], a real-time clock or simply a counter. A <server,clock> pair serves nicely as a WID, and this section assumes that WIDs are assigned in this manner by the server that first accepts the Write.

Each server maintains its own version vector with the following invariant: if a server has <S,c> in its version vector, then it has received all Writes that were assigned a WID by server S before or at logical time c on S's clock. For this invariant to hold, servers must transfer Writes in the order of their assigned WIDs during anti-entropy. A server's version vector is updated as part of the anti-entropy process so that it precisely specifies the set of Writes in its database.

Assuming the use of version vectors by servers, more practical implementations of the guarantees are possible in which the sets of WIDs are replaced by version vectors as follows:

To obtain a version vector V that provides a compact representation for a set of WIDs, Ws, set V[S] = the time of the latest WID assigned by server S in Ws (or 0 if no Writes are from S).

To obtain a version vector V that represents the union of two sets of WIDs, Ws1 and Ws2, first obtain V1 from Ws1 and V2 from Ws2 as above. Then, set V[S] = MAX(V1[S], V2[S]) for all S.

To check if one set of WIDs, Ws1, is a subset of another, Ws2, first obtain V1 from Ws1 and V2 from Ws2 as above. Then, check that V2 "dominates" V1, where dominance is defined as one vector being greater or equal to the other in all components [18].

With these rules, the state maintained for each session compacts into two version vectors: one to record the session's Writes and one to record the session's Reads (actually the Writes that are relevant to the session's Reads). To find an acceptable server, the session manager must check that one or both of these session vectors are dominated by the server's version vector. Which session vectors are checked depends on the operation being performed and the guarantees being provided within the session.

Servers return a version vector along with Read results to indicate the relevant Writes. In practice, servers may have difficulty computing the set of relevant Writes. For one thing, determining the relevant Writes for a complex query, such as one written in SQL, may be costly. For another, it may require servers to maintain substantial bookkeeping of which Writes produced or deleted which database items. In real systems, servers typically do not remember deleted database entries; they just store a copy of the database along with a version vector. For such systems, a server is allowed to return its current version vector as a gross estimation of the relevant Writes. This does not violate the Monotonic Reads or Writes Follow Reads guarantees, it merely causes the session manager to be overly conservative when choosing acceptable servers.

The vector-based Read and Write procedures are presented in Figure 1.

```
Read(R,S) = {
    if MR then
        check S.vector dominates read-vector
    if RYW then
        check S.vector dominates write-vector
    [result, relevant-write-vector] := read R from S
    read-vector := MAX(read-vector,
        relevant-write-vector)
    return result
    }

Write(W,S) = {
    if WFR then
        check S.vector dominates read-vector
    if MW then
        check S.vector dominates write-vector
    wid := write W to S
    write-vector[S] := wid.clock
    }
```

Figure 1. Implementation of the guarantees using version vectors.

As an additional performance improvement, the checks for a suitable server can be amortized over many operations within a session. In particular, the previously contacted server is always an acceptable choice for the server at which to perform the next Read or Write operation. Thus, if the session manager "latches on" to a given server, then the checks can be skipped. Only when the session manager switches to a different server, like when the previous server becomes unavailable, must a server's current version vector be compared to the session's vectors. To facilitate finding a server that is sufficiently up-to-date, the session manager can cache the version vectors of various servers. Since a server's database can only grow over time in terms of the numbers of Writes it has received and incorporated, cached version vectors represent a lower bound on a server's knowledge.

Caching of data at clients can also be used to improve overall performance and data availability. However, notice that circumstances may exist under which data that is available in the cache cannot be read by an application because it does not meet the application's session guarantees. As illustrated in the following example, this situation can arise when applications with different consistency requirements are sharing the cache. Suppose a client machine is executing two applications, a mail reader and a program that collects statistics on the mail messages that the user receives. The statistics gathering program has no consistency requirements and hence requests no session guarantees. On the other hand, the mail reader requires the Monotonic Reads and Read Your Writes guarantees as explained in Examples 2 and 4. Assume that at some point the statistics program reads from a server that holds an out-dated copy of the user's mail database, thereby filling the cache with old data. When the mail reader executes, allowing it to retrieve data from the cache would likely violate its Monotonic Reads guarantee. It is important to point out that this type of scenario can occur for any weakly consistent replicated system with client caching, regardless of the existence of mobile clients.

6. Adding guarantees to existing systems

Session guarantees could be utilized in many existing systems that provide weakly consistent replicated data semantics. Version vector based systems are of the most interest since they represent systems to which our guarantees could be added with relatively minimal effort. Examples of systems that employ version vectors to check the consistency of database copies include Coda [13,20], Ficus [10] (a successor to Locus [19]), OSCAR [7], and refdbms [8]. The Coda distributed file system is in many ways representative of modern-day systems that use version vectors, so we briefly examine how to add the four session guarantees to Coda.

Because Coda servers already export access to the version vectors they maintain, one could readily add the Monotonic Reads and Read Your Writes guarantees to a Coda client using the techniques presented in Section 5 without having to change any code running on Coda servers. Moreover, this upgrade could be done incrementally; only those users who wanted the session guarantees would need to install new client software. The interface between applications and the Coda client code would have to be extended so that applications could associate a session with their read and write operations and indicate the session guarantees they require.

Adding the Writes Follow Reads and Monotonic Writes guarantees would be equally simple since Coda meets the propagation requirements described in Section 4. Specifically, a server that is determined to be out-of-date obtains newer copies of updated files by atomically synchronizing the contents of a volume with that of another server.

Two different levels of granularity are possible for session guarantees since Coda maintains version vectors for both files and volumes. Using volume version vectors would allow session guarantees to cover operations spanning multiple files within a volume. However, since volume version vectors reflect all the updates made to any file in a volume, clients interacting with only a single file might prefer to keep track of individual file version vectors to obtain better availability.

7. Related work

A number of other systems and replication schemes exist that provide guarantees between strong consistency and weak (eventual) consistency or that provide individual applications some control over their perceived consistency.

The work that is closest to ours is that of Ladin, Liskov, Shrira, and Ghemawat on "causal operations" [14]. In their design, weakly consistent copies of a database are updated via "gossip" messages. Clients ensure causal ordering of their Read and Write operations by means of version vectors that accompany each client interaction message. The key difference between their design and ours is that theirs has no notion of sessions or of being able to specify the more fine-grained consistency requirements that we introduce in this paper. Also, a client in their system cannot switch to another host and have its operations be causally ordered with respect to its previous activity without doing a heavyweight synchronization action.

In many systems with lazy replication, such as Lotus Notes [12], clients desiring consistency among multiple read and/or write operations must, in general, use the same server for their interactions. In contrast, our approach focuses on providing guarantees to clients that routinely interact with multiple servers.

As an example of a system that offers mobile computing users and applications a choice of consistency levels, the file system of Tait and Duchamp supports both *strict* and *loose* read operations [22]. The former provides strongly consistent semantics, which ensures that the most recent version of a file existing in the system is returned, and the latter provides weakly consistent semantics where any available copy is returned.

Much work has been done on providing various "degrees of consistency" in database systems [9]. This work focuses on relaxing the isolation between transactions, yielding reduced consistency, in order to increase concurrency. Weakly consistent systems, on the other hand, generally provide ample concurrency with little or no isolation. Our session guarantees are intended to provide applications with increased consistency, but do not address the problem of isolation between concurrent applications. For an example of an attempt to provide increased isolation for clients of a replicated file system, see the recent proposal by Lu and Satyanarayanan for "Isolation Only Transactions" [16].

Another form of intermediate consistency involves controlling the amount of inconsistency that may occur among data replicas. Two examples are bounded inconsistency [3] and quasi-copies [2]. These approaches provide a different kind of consistency than session guarantees and should be viewed as complementary techniques.

8. Conclusions

Four new per-session guarantees have been proposed to aid users and applications of weakly consistent replicated data: Read Your Writes, Monotonic Reads, Writes Follow Reads, and Monotonic Writes. These guarantees can provide an application with a view of the replicated database that is consistent with its own Reads and Writes performed in a session even though these operations may be directed at different servers. The goal is to achieve semantics close to those of a shared, centralized database while retaining the principal benefits of a read-any/write-any replication scheme, namely high-availability, simplicity, scalability, and disconnected operation.

Even with the guarantees, an application must be aware that other users and applications may be concurrently updating data that it Reads or Writes. In particular, our guarantees do not attempt to provide atomicity of multiple updates or serializability of concurrent activities. Indeed, these are orthogonal issues and solutions to them can be added to a system independent of the guarantees discussed in this paper.

A key aspect of our design is that applications may choose just the guarantees that they require. This is accomplished by providing the guarantees within the context of a session. Moreover, since sessions are lightweight entities, a single application may create several sessions that it uses to exercise fine-grain control over the guarantees it desires.

The main cost of requesting session guarantees is a potential reduction in availability compared to a basic read-any/write-any replication scheme. We expect the impact on availability to be small in practice. Indeed, for many clients, such as those employing local caching, availability may be only rarely affected.

Practical implementations of our guarantees have been developed. Since no system-wide state is maintained and no additional coordination among servers is needed, an efficient, localized implementation is possible. The amount of per-session state needed to ensure all of the guarantees is small, consisting of only two version vectors. Also, the cost of checking those version vectors against a server's vectors to determine if the server is sufficiently up-to-date is small, and frequently can be amortized over many session operations.

The implementation techniques make only a few reasonable assumptions about how servers order and propagate updates. In general, the guarantees could be added to systems employing a variety of data models, Read/Write operations, conflict resolution procedures, and schemes for replica control.

Several existing replicated systems already successfully use version vectors in a manner similar to the implementation we propose, implying that our guarantees can be layered on these systems. In the case of the Coda file system, the guarantees can be provided simply by changing the clients; no changes to the servers are required.

These session guarantees were developed while designing a replicated storage system to support mobile computing users who may be frequently disconnected yet wish to collaborate. An implementation of session guarantees for databases that are shared between workstations and laptop computers is currently underway as part of the Bayou project at Xerox PARC.

9. Acknowledgments

We thank our colleagues with whom we have discussed many of the ideas in this paper, especially Tom Anderson, Mary Baker, Helen Davis, Dan Greene and Carl Hauser, who provided valuable feedback on earlier versions of this paper. Finally, we appreciate Mark Weiser's leadership in pursuit of the Ubiquitous Computing vision.

10. References

[1] D. Agrawal and A. Malpani. Efficient dissemination of information in computer networks. *The Computer Journal* 34(6):534-541, December 1991.

[2] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Transactions on Database Systems* 15(3):359-384, September 1990.

[3] D. Barbara-Milla and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. To appear in *VLDB Journal*.

[4] A. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM* 25(4):260-274, April 1982.

[5] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: A survey. *ACM Computing Surveys* 17(3):341-370, September 1985.

[6] 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 Sixth Symposium on Principles of Distributed Computing*, Vancouver, B.C., Canada, August 1987, pages 1-12.

[7] A. R. Downing, I. B. Greenberg, and J. M. Peha. OSCAR: A system for weak-consistency replication. *Proceedings Workshop on the Management of Replicated Data*, Houston, Texas, November 1990, pages 26-30.

[8] R A. Golding. Weak consistency group communication for wide-area systems. *Proceedings Second Workshop on the Management of Replicated Data*, Monterey, California, November 1992, pages 13-16.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[10] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page, Jr., G.J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conference Proceedings*, pages 63-71, USENIX, June 1990.

[11] T. Imielinski and B. R. Badrinath. Data management for mobile computing. *ACM SIGMOD Record* 22(1):34-39, March 1993.

[12] 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.

[13] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *Proceedings Thirteenth ACM Symposium on Operating Systems Principles*, Pacific Grove, California, October 1991, pages 213-225.

[14] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems* 10(4):360-391, November 1992.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM* 21(7):558-565, July 1978.

[16] Q. Lu and M. Satyanarayanan. Isolation-only transactions for mobile computing. *ACM Operating Systems Review* 28(2):81-87, April 1994.

[17] D. C. Oppen and Y. K. Dalal. The Clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Transactions on Office Information Systems* 1(3):230-253, July 1983.

[18] D. S. Parker, 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* SE-9(3):240-246, May 1983.

[19] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A network transparent, high reliability distributed system. *Proceedings Eighth Symposium on Operating Systems Principles*, Pacific Grove, California, December 1981, pages 169-177.

[20] 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, April 1990.

[21] M. D. Schroeder, A. D. Birrell, and R. M. Needham. Experience with Grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems* 2(1):3-23, February 1984.

[22] C. D. Tait and D. Duchamp. Service interface and replica management algorithm for mobile file system clients. *Proceedings First International Conference on Parallel and Distributed Information Systems*, December 1991, pages 190-197.