January 1998

View consistency for optimistic replication

Ashvin Goel

Calton Pu

Gerald J. Popek

View Consistency for Optimistic Replication

Ashvin Goel, Calton Pu
Department of Computer Science and Engineering
Oregon Graduate Institute, Portland

Gerald J. Popek
Computer Science Department
University of California, Los Angeles

Abstract

Optimistically replicated systems provide highly available data even when communication between data replicas is unreliable or unavailable. The high availability comes at the cost of allowing inconsistent accesses, since users can read and write old copies of data. Session guarantees [15] have been used to reduce such inconsistencies. They preserve most of the availability benefits of optimistic systems. We generalize session guarantees to apply to persistent as well as distributed entities. We implement these guarantees, called *view consistency*, on Ficus an optimistically replicated file system. Our implementation enforces consistency on a per-file basis and does not require changes to individual applications. View consistency is enforced by clients accessing the data and thus requires minimal changes to the replicated data servers. We show that view consistency allows access to available and high performing data replicas and can be implemented efficiently. Experimental results show that the consistency overhead for clients ranges from 1% to 8% of application runtime for the benchmarks studied in the prototype system. The benefits of the system are an improvement in access times due to better replica selection and improved consistency guarantees over a purely optimistic system.

1 Introduction

Many distributed applications require *highly available* data, or data that can be accessed at any time. Data is often replicated to increase availability. However this introduces the *data consistency* problem. Data accesses are inconsistent when they reflect the intermediate states of data between accesses. For example, if replica A has been updated and the update has not reached replica B, then accesses to replica B may be inconsistent. Replicated systems aim to provide highly available data while preserving data consistency. These goals are conflicting because consistency may only allow access to certain replicas, while availability improves when any replica can be accessed.

Optimistically replicated systems provide high availability even in weakly connected environments by allowing accesses to any file replica. This continuous access, even during network partitions, is critical for many applications such as reservation systems, appointment calendars, design doc-

uments, meeting notes, and in general, mobile file accesses [7, 4, 16]. Inconsistencies due to updates being made to old copies (*conflicting updates*) are *eventually*[1] detected and resolved. Unfortunately, the lack of consistency guarantees during accesses can be very confusing to users.

Session guarantees [15] have been used to reduce inconsistencies observed in optimistic systems. They maintain most of the availability benefits of such systems. Session guarantees preserve read and write dependencies for processes. An application session is presented with a view of the database that is consistent with its own actions, even if it reads and writes from various, potentially inconsistent servers.

In this paper, we propose the *view-consistency* model that enhances session guarantees to apply to persistent as well as distributed sets of clients. This model provides "session guarantees" for a larger set of applications. View consistency provides conservative guarantees to each *single* client or each *group of closely-related*[2] clients, while eventual consistency is maintained across "distant" clients.

The view-consistency model attempts to capture a real-world working environment in which a single client or closely cooperating clients would like to access mutually consistent data all the time, but distant clients wish to synchronize with each other occasionally. Consider two groups of researchers working in two different countries on the same system. Each group is building new functionality for the system. Suppose the system code is optimistically replicated in the two countries. Within each group, view consistency maintains consistent accesses. However, the two groups are not synchronized with each other. The optimistic system will eventually make the two groups consistent. The advantages of the view-consistency model are two-fold: closely cooperating clients observe mutually consistent data, and distant clients do not pay the instantaneous cost of maintaining consistency (during accesses). Therefore view consistency enables useful collaboration in many large scale environments.

Distributed applications not only require availability of data but also fast access to data. Providing fast accessibility and consistency can be conflicting goals. For example, a highly performing data replica may not have consistent data. In this paper, we show that *replica selection* (providing data from available and high performing replicas) can be performed while maintaining view consistency, and the overhead of providing view consistency during replica selection is small.

The contributions of the paper are the following: first, we introduce the concept of a generalized client, called an *entity*, and provide session guarantees for an entity. Entities can be persistent as well as distributed. Defining a consistency model for entities rather than for sessions allows us to provide guarantees to a larger set of applications. Second, we describe the interaction of view consistency with replica selection and show that system performance can be improved by accessing high performing replicas while maintaining consistency. Our implementation approach has two advantages: first, we implement view consistency as a file system, and thus do not require any changes to user-level applications. Second, as with session guarantees, consistency is enforced by clients and thus minimal changes have to be made to data

[1] Detection and resolution of conflicts occurs as a separate process, and is often unrelated to the time at which files are accessed.

[2] We define the notion of "closely-related" in the next section.

servers.

Section 2 describes our consistency model. Section 3 explains the motivations for using view consistency. The view-consistency algorithm and the challenges in implementing it are discussed in Section 4. Section 5 presents a prototype implementation of view consistency on the Ficus optimistically replicated file system. The overhead of providing consistency and the performance benefits of replica selection are studied in our benchmarks in Section 6. Section 7 discusses related work and Section 8 draws conclusions and suggests future work.

2 View-Consistency Model

In this section, we define session guarantees and then view consistency. View-consistency guarantees are defined with respect to clients accessing the data. We explain the notion of "closely" cooperating clients by generalizing the definition of a client. Such a generalized client is called an *entity*.

Definition 1 *Session guarantees allow a client to access versions of data that are the same as or newer than (for brevity, we will call this* later than*) what the client had previously accessed.*

This definition of session guarantees is a combination of the *read your writes*, *monotonic reads*, *writes follow reads* and *monotonic writes* guarantees as described by Terry, et al. [15]. View consistency can be defined for each individual session guarantee, but we will ignore these distinctions in this paper for simplicity.

Session guarantees are provided for single clients. View consistency can be provided to groups of clients. A closely cooperating groups of clients is called an *entity*. Each individual within the entity is called a *component*. Examples of entities are a single process, a group of processes, a user working on a laptop, all the users on a machine, a group of machines, etc.

Definition 2 *View consistency allows an entity to only access later versions of data than what the entity had previously accessed. A data version is later if it is the same as or newer than the latest version accessed by any of the components of the entity.*

An entity is therefore a generalized client that may be persistent or distributed. Its components are cooperating closely since view consistency ensures that they access mutually consistent data.

Entity Classification

Entities can be of different types. Here, we describe these types and then present a representative set of entities. Long-lived entities that survive machine crashes are *persistent* entities, while short-lived entities are *transient* entities. The consistency information for a persistent entity must be kept on secondary storage. A *complex* entity can access replicas of data via multiple independent processes, whereas *simple* entities have a single execution thread. Complex entities must coordinate their accesses to see later versions of data. A complex entity can exist on a single machine (*centralized*) or on multiple machines (*distributed*). A distributed entity can be denied access to data either because later versions of data are not available, or because its sub-entities cannot be coordinated at a particular time. Mechanisms such as primary coordinator, token passing, or voting are needed to synchronize the accesses of a distributed entity. Note that these mechanisms are applied at the entity and not at the replicated data servers.

Common entities include a single process, a group of processes, a login session, a single machine, a closely related group of machines, a user on a single machine, or a group of machines, etc. Note that a *user* by itself is not a useful entity since a user's processes may originate from a large number of machines and be hard to locate or coordinate.

3 Why View Consistency?

The benefits of view consistency are illustrated with examples below. The underlying replicated service is assumed to allow accesses to any available data replica.

1. A user is accessing a *web* page that is replicated at several sites. If the current site becomes heavily loaded and disallows accesses, view consistency will ensure that the user does not access older versions of the web pages from another site.

2. A user edits a file and then checks in the new version of the file into a replicated version-control system. The replica that has the latest changes becomes inaccessible before these changes propagate to other replicas. If the user can access and edit the file from another replica, this action will necessarily create a conflicting update. View consistency will disallow accesses to any other file replica, since these replicas are older than the replica on which the user was working.

3. A user accesses a replicated *web* page and caches (or stashes [8]) the page. Later, this page is evicted to make space for other more important pages. View consistency ensures that remote accesses of the original web page yield later versions of data. Moreover, later stashes of the web page (when it is accessed and cached again) will also be data versions that are later than what the user has seen previously.

4. Suppose users A and B at one office are sharing files with users C and D at another geographically distant office. Each user has a replica of the files. A and B (and similarly C and D) are actively cooperating with each other. We define A and B to be an entity, and C and D to be another entity. View consistency will ensure that both A and B (and likewise C and D) access data that is later than each has accessed.

Discussion

View consistency is enforced by each client (or entity) accessing the data and not by the servers. This client consistency model has several implications. First, servers do not have to be changed to implement view consistency. Second, the consistency model implemented by the servers does not affect view consistency. The only requirement (as we will see later) is that the client should be able to compare file versions. Third, different clients can observe different guarantees. For example, one client may be view consistent while another may ignore view consistency while operating on the same data. Later, the two can be combined and observe view consistency as a single entity. Fourth, view consistency does not attempt to coordinate the accesses of different clients, and thus different clients can make conflicting updates. This lack of inter-client coordination, however, allows high data availability at each client.

The choice of entities is very important for view consistency. For a given set of files, this choice strongly depends on the file usage pattern. For

some files, each user of the file may choose to remain a separate entity. For shared files, a group of users or a group of machines may be chosen as the appropriate entity. Effectively chosen entities reduce concurrent accesses, without significantly affecting availability or the performance of the system, as compared to a completely optimistic system. Currently this choice is made explicitly by the user in our system. More experience is needed with our system regarding the appropriate choice of entities, and an automated method for choosing such entities.

4 Algorithm

The view-consistency criterion allows an entity to access data that is *later* than what the entity has seen *previously*. Entities can store the version of data that they last read or updated. This version can then be used to ensure that the next access yields a later data version.

Figure 1 shows the view-consistency algorithm for a generic entity. The `viewMediator` function is called by file operations that read or write some data (discussed in the next paragraph). The version and the replica (collectively called *view-entry*) that was *last* accessed for a particular file is obtained by `readViewEntry`. This information in the view-entry is used to switch to a later file replica in `switchToLaterReplica`. If the view-entry for a file does not exist, the file has never been accessed,[3] and `switchToFastReplica` is used to switch to *any* highly available replica. The "switchTo" functions are further described in Section 5.3 when we discuss replica selection issues for higher availability. The file operation is enhanced to return the data and the version of the data that was accessed. This version and the replica that were accessed are stored by `writeViewEntry` as the new view-entry for the file and are used for the next access.

```
viewMediator(file, entity, fileOperation)
{
    (fileId, replica) = file; // file consists of fileId, replica
    viewEntry = (viewVersion, viewReplica) =
                    readViewEntry(fileId, entity);
    if (viewEntry != NULL) {
        newReplica
            = switchToLaterReplica(file, viewEntry);
    } else {
        newReplica = switchToFastReplica(file);
    }
    (data, fileVersion) = fileOperation(fileId, newReplica);
    if (fileVersion > viewVersion) {
        writeViewEntry(fileId, entity,
            fileVersion, newReplica);
    }
    return data;
}
```

Figure 1: The general view-consistency algorithm

Besides file reads and writes, directory operations and file attribute must also invoke the view mediator. Without directory consistency, a renamed file may appear with its older name in the future. File attribute consistency is needed to ensure correctness of applications such as `make` that depend on data and attribute consistency.

The main challenges in the implementation of the algorithm are:

- The file replica version information must be available from the server (in `fileOperation`) and must be comparable with other replicas of the file.

- Efficient reading and writing of the stored

[3] Even if the file has been accessed, the view-entry may not exist because it has been garbage collected, as discussed in Section 5.4. Such a file can be treated as if it has never been accessed.

view-entries (file version and replica id) of an entity. This operation requires coordinating the accesses of a complex entity. In a distributed entity, the view-entry may exist separately from the components of the entity. As an example, volatile witnesses [11] can be used for storing and accessing the view-entries. These witnesses would be placed so that they are more available than the individual components of the entity.

- Replica selection is done while maintaining view consistency, and it must be efficient since it lies in the critical path of every file operation.

- The view-entry for a file at an entity must be deleted when it is not required anymore.

We discuss these issues in the next section.

5 Implementation

We have implemented view consistency and replica selection as a separate stackable layer [5] on Ficus [10], an optimistically replicated file system.

5.1 Comparable File Versions

Ficus allows accesses to any available file replica and detects writes to older replicas, when replicas communicate, by using vector timestamps [12]. Each file replica has a vector timestamp of length n, where n is the number of replicas of the file. A vector timestamp is later than another if each component of the timestamp is greater than or equal to the corresponding component of the other. Our implementation of view consistency uses this property to test the consistency criterion.

5.2 Accessing View-Entries

View-entries are stored, for each file that an entity has accessed. The view-entry storage service must take into account the following:

- View-entry reads and writes are in the critical path of file operations.

- Accesses to view-entries must be coordinated for distributed entities.

- View-entries must be stored persistently for persistent entities.

- View-entries are stored for every file that is accessed. Thus their number can grow large.

We chose Margo Seltzer's *db* database package [14] for view-entry storage. It is relatively small, and caches large chunks of the database in memory for efficient access. View consistency is implemented in the kernel while the database runs at the user level. Therefore, there is communication and context switch overhead every time the database is accessed. We therefore cache view-entries in the kernel.

View-Entry Caching We cache the view-entry for a file with the *vnode* (file handle obtained on a file open) of the file. Effectively, the view-entry is obtained from the database on each file open rather than on each file operation. Moreover, UNIX kernels cache vnodes in the name lookup cache even when the file is not being used. The view-entry therefore stays in the kernel as long as the vnode stays in the cache. The effectiveness of this cache and thus the overhead of view consistency depends on the locality of file accesses.

Optimistically Fetching View-Entries View-entry caching reduces the number of times

the database is accessed. Another method of reducing the consistency overhead is to remove `readViewEntry` from the critical path of the file operation. `ReadViewEntry` can be performed in parallel with the file operation *optimistically*. This operation is successful if the file operation yields a data version that is later than the version in the view-entry. Otherwise, the operation must be tried again with a consistent replica. We see in Section 5.3 that this optimism will generally pay off. Although it is possible to implement parallel kernel operations, we have not done so in the current implementation.

Coordinating Accesses to View-Entries Currently, users specify the entity type in their user-profile. Entities can be defined per Ficus volume [13]. The entity type is encoded in the entity parameter in Figure 1. This parameter is only used by the `readViewEntry` and `writeViewEntry` routines. Therefore, the cost and the complexity of providing a consistent view to a specific entity depends on the cost of reading and writing view-entries. For distributed entities, the storage service at each of the different components of the entity must coordinate the accesses to the view-entries. This can be done by using a token passing mechanism where the token is the view-entry. We have not completed the implementation of this functionality. In the future, we intend to study the issues related to the choice of distributed entities (described in Section 8).

Obtaining View-Entries The file operation in Figure 1 returns both the file data and the file version. We use NFS as our transport layer. NFS does not return the file version information for file operations. Although obtaining the file version information separately after the file operation does not violate view consistency (the file version will have a later version), it can be expensive since each file operation may require an extra (possibly remote) operation.

We have modified some of the file operations (such as `lookup`) in Ficus to return file data and version together. For other operations, the version information is obtained separately. There is significant performance difference between getting version information along with the data versus separately, as shown in Section 6. We therefore plan to change all the relevant Ficus operations to return both the file data and version together.

Writing View-Entries For persistent entities, modified view-entries must be stored on disk before the data is returned to the user. The cost of writing to disk after each view-entry update is high. Optimizations such as log record buffering are *not* possible since each file operation returns data and is similar to a *commit*. Instead, we write back updated view-entries to the database when 1) vnodes are destroyed, 2) on a file `close`, or 3) periodically every 30 seconds. The view-entry database flushes its own updated memory entries to disk every 30 seconds. These mechanisms together ensure that view-entries updated more than 60 seconds ago are stored on disk. After a machine crash, entities may see older versions of those files that they had been accessing a minute before the crash.

Local Replica Optimization An optimization is possible for centralized entities when a file replica is stored on the local machine. In this case, view-entries do not have to be stored, since the entity does not need to access remote replicas and local accesses yield later data versions. Suppose a

user (or an application) decides to add a local replica of a file that he has been accessing remotely. The local replica will be view consistent if the data is copied from the remote replica that is being accessed (and is view consistent). The view-entry in the local database can then be removed. A local file replica deletion must record the version of the deleted data. The next access to a remote replica uses this version to ensure view consistency.

5.3 Replica Selection

This section describes the Ficus replica selection policy and then presents modifications to support view consistency. Replica selection is done solely at the clients. This is possible because view consistency is enforced by an entity and not by the replicas.

The basic replica selection policy (ignoring view consistency and performance) ensures that the same replica is accessed for all the files in a Ficus volume. For availability and performance (again ignoring view consistency), Ficus aims to achieve the following: provide data as long as any replica is available (*availability* criterion), provide data from the fastest available replica (*optimality* criterion), and minimize the overheads of replica selection and switching. The system maintains a *delay value* for each replica that determines the bandwidth and latency to the replica from the client site. Optimality uses these delay values. Unlike availability, which rectifies a short-term failure condition, optimality improves the long-term throughput and efficiency of the system. `SwitchToFastReplica` shown in Figure 1 implements both the availability and the optimality criterion.

View Consistency and Availability

The `switchToLaterReplica` function in Figure 1 ensures that consistent replicas are accessed and, like `switchToFastReplica`, it tries to provide available as well as optimal replicas (within the constraints of consistency). As long as the same replica is accessed the data is view consistent. However, this data replica may not be the highest performing replica.

The `switchToLaterReplica` function uses the following order to select replicas:

- The view-entry replica is chosen since this replica is known to be consistent.
- If the view-entry replica is unavailable or much slower than the highest performing replica, the current replica is chosen.
- If the current replica is inconsistent, a search of all replicas is done (in ascending delay-value order) to find another consistent replica.
- If the current replica is consistent but slow, it tries to switch to a fast, consistent replica.

The normal path is that the current replica is the view-entry replica and it is not much slower than the fastest available replica; nothing needs to be done in this case. Note that at every step both consistency and availability are taken into account, but only a consistent replica is returned.

5.4 Garbage Collection of View-Entries

Each entity has a logically separate database that contains the view-entries for files that the entity has accessed. The number of view-entries grows as more files are accessed. These view-entries can be deleted when they are no longer

required. The database for transient entities can be entirely removed when the entity terminates.

For persistent entities, view-entries can be deleted when all the file replica versions are known to be later than (equal to or newer than) the version in the view-entry. The view-entry is not required anymore since any replica that is next accessed will yield a later version. Note that view-entry deletion can be done independently of file operations.

Deletion Algorithms A simple deletion algorithm can obtain the current version of each of the file replicas and delete the view-entry when all the versions are later than the view-entry version. This deletion mechanism is expensive since all replicas must be contacted and requires that all replicas be available at the same time.

A second solution to the deletion problem involves using the *reconciliation* process in Ficus. Reconciliation in Ficus occurs periodically, when replicas communicate pairwise and exchange updates until they are mutually consistent. For deletion, each replica can obtain file version information during reconciliation. A client site can contact any one replica and delete a view-entry when all the versions stored at this replica are later than the view-entry version. This solution has a high storage overhead because n version vectors (total size $O(n^2)$) have to be stored at each replica for each file. Moreover, there is a strong requirement that replicas communicate with every other replica directly.

Acknowledgment Algorithm Another deletion solution uses the information already made available to replicas after reconciliation. Recall that a view-entry can be deleted when all replicas have higher file versions. Alternately, the view-entry can be deleted when the view-entry replica (replica in the view-entry) has propagated the view-entry version to all other replicas. This slightly modified criterion requires that a replica know about what other replicas have learned about itself, i.e., have the other replicas seen a file version that this replica had stored in the past? We call such information an *acknowledgment*.

Suppose each replica stores an acknowledgment-timestamp vector, where replica r_i has heard about all updates from this replica until the time in the i^{th} component of this replica's vector. Now the client's view-entry can be deleted if the time at which the file replica was last accessed (the view-entry's timestamp) is less than all the components of the acknowledgment-timestamp vector at that file replica (this version has been seen by all other replicas). Note that the client has to communicate only with the replica in the view-entry for view-entry deletion.

Acknowledgments are obtained in Ficus during reconciliation in a two step process. In the first step, a replica learns about the state of the other replicas. In the second step, the replicas gossips this information to the other replicas, who now learn what other replicas know about themselves. See Guy [3] for further details. Acknowledgments have also been used by Wuu [17] and Ladin [9]. The difference between their work and ours is that they use acknowledgments for garbage collection at the replica servers while we use them at the clients also.

6 Experiments and Evaluation

We have implemented view consistency as a stackable file-system layer over the Ficus file system [5]. A user-level view-entry database provides

view-entries to the kernel. These view-entries are garbage collected by a deletion server that obtains the acknowledgment information from the reconciliation process. A delay server determines the latency and bandwidth to different replicas and provides these values to the kernel for replica selection. The experiments presented here evaluate two of the aspects of the system: 1) measuring the overhead of providing view consistency and 2) the costs of switching to the high performing replicas while providing view consistency.

6.1 View-Consistency Overhead

The overhead of view consistency is measured by comparing the cost of view consistent versus non-view consistent (or base Ficus) accesses. We use four Sun IPCs, each with 12 MB of main memory, connected by a 10Mb/s Ethernet connection. Accesses were done from one machine, while data replicas were stored remotely on the other three machines. No tests were done with locally stored replicas because view consistency can then be provided with no overhead (Section 5.2).

We performed seven benchmarks with one, two and three data replicas. The first test is the modified Andrew Benchmark (`mab`) [6] that is intended to model a mix of filing operations and hence be representative of performance in actual use. The second and third tests are local and remote recursive `cp` and the fourth test is `grep`. Each of these tests exercise the read and write file operations. The fifth and sixth tests are `find` and `rm` programs that primarily execute recursive directory operations. The last test is the `ls` program, which reads directory contents. The `mab` test is performed on 1.3 MB of data. The `grep` and `ls` tests operate on 104 files containing 336KB of data. All other tests operate on 1311 files with 4.2 MB of data.

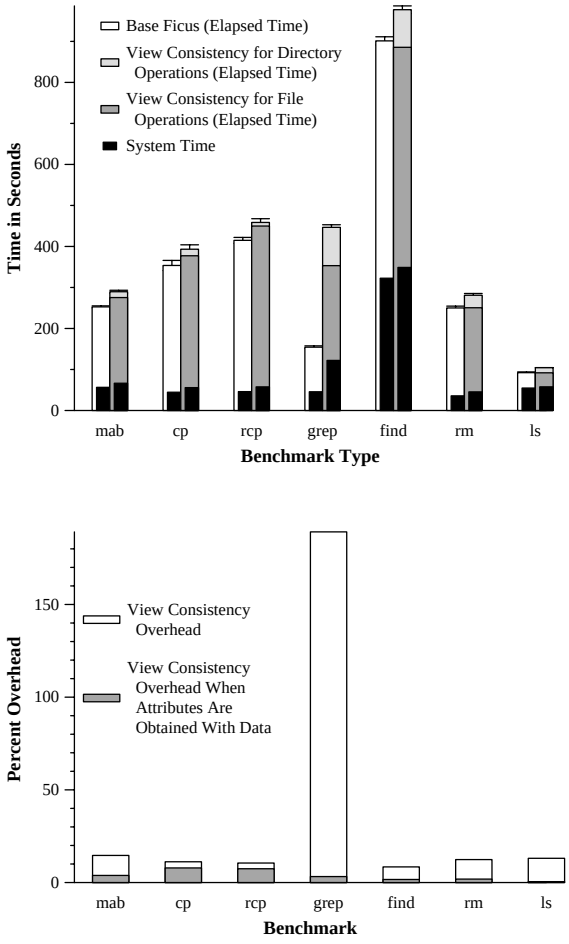

Figure 2: Remote access times of base Ficus and view-consistent Ficus. The lower graph shows the consistency overhead (in gray) when the view consistency attributes are obtained along with data.

The results of the three replica benchmarks are shown in Figure 2. Since view consistency is enforced by clients, the overhead does not change significantly with different numbers of replicas and the results for the single and two replica experiments are very similar. The upper graph in Figure 2 shows the elapsed and system times of base Ficus and view-consistent Ficus for remote accesses. The 95% confidence intervals are shown for the elapsed times. The costs of providing

view consistency for file operations and for directory operations are shown separately for view-consistent Ficus. Note from the upper graph that `find`, `rm` and `ls` have no view-consistency overhead for file operations, since these operations predominantly operate on directories. The overhead of view consistency for remote accesses is also shown in the lower graph of Figure 2. This overhead includes both file and directory operations. The overhead for all tests except `grep` is between 5 to 12 percent. The `grep` overhead is 185 percent.

To understand this large overhead, we compared `grep` and `cp` since they perform similar vnode operations. We found that most of the overhead in the `grep` benchmark occurs because we obtain view-consistency attributes separately from data, and thus go over the wire twice for each file operation (for which view consistency is provided). We also found that the cost of getting remote attributes is 8.5 ms per operation, cost of running `grep` on a single remote file is 17.4 ms, and the cost of performing a `cp` on a single file is 173.5 ms in Ficus. Therefore, the large `grep` overhead is because `grep` is a much faster operation and because getting attributes is a fixed cost operation. Moreover, while `cp` gets attributes once, `grep` gets attributes twice per file. This by itself doubles the time for executing `grep`.

The gray area in the lower graph shows the overhead of view consistency when attributes are obtained with data. The overhead for `grep` and for most other benchmarks decreases to between 1 to 8 percent. We measured this overhead by using attributes that are obtained during opens and by not updating these attributes from the server on each operation.[4] Finally, as explained in Section 5.2, the 1 to 8 percent overhead of view consistency can be reduced even further by getting the view-entries in parallel with the file operations.

6.2 Availability Measurements

We measured the cost and performance benefits of switching to the highest performing replica while providing view consistency. Accesses are switched to a new replica when it is view consistent and improves overall access times. The overall performance of each replica is measured in terms of replica delay values that are determined by a user-level delay server. The replica delay values were simulated in our experiment as shown in the upper-most graph of Figure 3. This was done because delay values did not change significantly (or frequently) in our experimental LAN environment. The delay[5] values were changed every 300 seconds. They were fixed at 15 for replica 1, varied periodically between 7 and 23 for replica 2, and varied randomly between 0 and 31 for replica 3.

The seven benchmarks used earlier (Section 6.1) were run with the simulated delay values. The replica that is accessed is shown in the middle graph of Figure 3. The number of replica switches during the experiment[6] is shown in the lowest graph in Figure 3. The lowest and the middle graph in Figure 3 show that multiple replicas are accessed during replica switching.

Replica switching takes place when the currently accessed replica is at least *switching-factor* (set at 2 for this experiment) slower than the best

[4] This can violate view consistency for operations other than open, but is nonetheless useful for understanding the overhead.

[5] A faster replica has a lower delay value.

[6] The total time to perform these benchmarks is approximately 50 minutes (as can be seen by taking the sum of the elapsed times of each of the benchmarks in the upper graph of Figure 2) but this experiment took 100 minutes because each access is logged to disk.

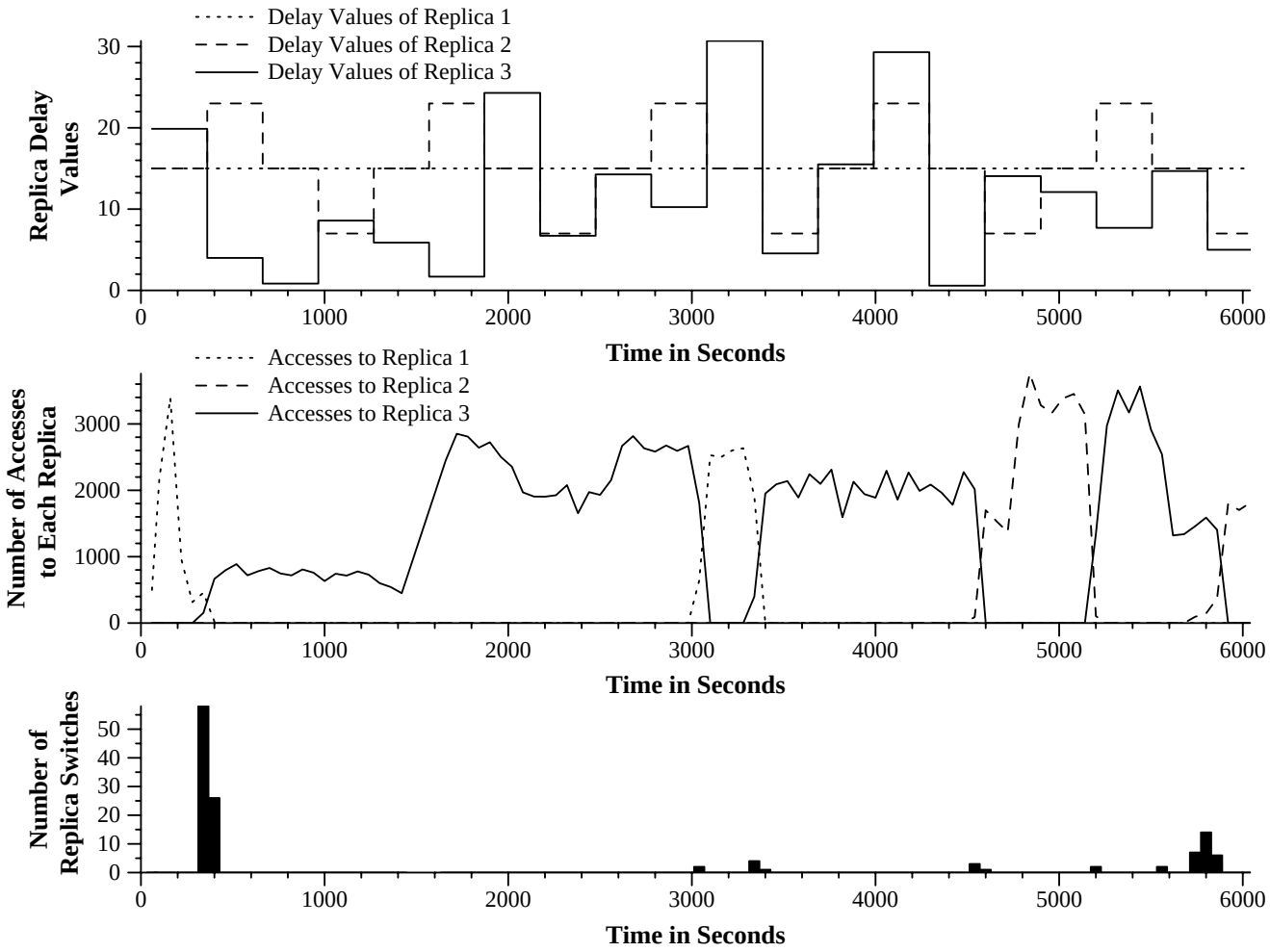

Figure 3: The replica delay values for three replicas, the replicas accessed, and the number of switches performed in a period of one hour and forty minutes

replica. This was true for all the replica switches except the last one (around time 5800 seconds) when accesses switched from replica 3 to replica 2 although replica 3 is the fastest replica. The delay value difference between replica 2 and 3 is not much at this time. Many files were accessing replica 2 in the past (around time 5000 seconds). This last accessed replica is known to be view consistent and is not much slower than the fastest replica. It is therefore given priority and these files still access replica 2. This condition does not occur in any other part of the experiment.

The total number of accesses in the experiment was 192215. With no replica switching, the average access time would be 15 (in terms of delay values), since replica 1 would be accessed each time. The ideal average access time (the replica with the lowest access time is accessed every time) is 9.68, a 36% improvement. The average time for each access with replica switching is 13.36, a 11% improvement. The choice of the switching-factor affects this improvement. A lower value improves

performances but can cause more replica switching overhead. The value also depends on the type of environment. Generally, a larger value should be chosen when the bandwidth and latency change rapidly, as in large-scale environments. A smaller value can be chosen when replica switching costs are low.

An interesting result of the experiment as seen in the lowest graph is that just a few replica switches induce every file to switch replicas. This happens because of the default replica switching rule. When a directory gets switched to a new replica, later accesses to files in the directory start by accessing this new replica. Although the last accessed replica has higher priority, this replica is very slow at each replica switching period (except the last one as explained above). Thus the new replica is given higher preference and accessed, and switching happens naturally for most files. Therefore, the explicit cost of switching in Ficus is very low.

7 Related Work

Our work is closely related to Bayou [15], an eventually consistent system, that provides session guarantees to reduce client inconsistency. These session guarantees are provided to process and process groups. We extend session guarantees to handle other transient, persistent and distributed entities. Instead of viewing the problem as providing guarantees for a session, we view it as providing guarantees to entities. We discuss view consistency for distributed entities and believe that many more applications will benefit from such guarantees. We also show how replica selection interacts with view consistency.

Causal ordering of reads and updates by Ladin, et al., [9] provides guarantees similar to view consistency. Unlike view consistency, causal ordering requires *application-specific* changes since applications must specify the causal relation between their operations. Causal ordering is enforced by replicas while view consistency is enforced by clients or entities. View consistency is therefore more scalable (in terms of server load) and requires minimal changes at the servers. An advantage of causal ordering is that it can provide more generic inter-client guarantees. View consistency deals with this issue partially by combining the clients into a single entity and providing consistency guarantees to this entity group. The combining of the clients into a single entity can be done dynamically.

Client-based consistency has been used by Alonso, et el., [1] to provide *quasi-copy* consistency. Quasi-copies are cached (or stashed) copies of data that may be somewhat out-of-date, but are guaranteed to meet certain consistency predicates. Client consistency for quasi-copies can generally be maintained for age-dependent predicates only. For example, the "not more than two versions old" predicate can only be enforced by the server.

Zadok and Duchamp [18] address the problem of providing data from the fastest available replica. They improve the auto-mounting daemon in UNIX systems and allow transparent switching of open files to replacement file systems that are dynamically discovered. The latency of the NFS `lookup` operation is monitored and used to assign delay values to different replicas. Their solution works for read-only file systems because they do not deal with replica consistency. Thus issues related to tradeoffs between consistency and availability do not have to be addressed.

8 Conclusions and Future Work

View consistency aims to provide consistent data in widely distributed and mobile systems. It covers the space between conservative and optimistic schemes by providing consistency to each entity while allowing high availability across entities. Effectively chosen entities can reduce concurrent accesses for various user working styles. The model is scalable in the number of replicas since clients enforce consistency. This paper focuses on whether view consistency can be achieved in practice. The prototype implementation on Ficus maintains the consistency information at each client efficiently. It provides view consistent data while taking data availability and performance into account. Our experiments show that view consistency for centralized entities can be provided at a low cost.

Future Work More experience is needed with view-consistent systems. Does view consistency satisfy the consistency demands of many applications? We are currently developing a user-level version of Ficus called *Rumor* that can be deployed more extensively. The benefits and costs of view consistency for large-scale disconnected and mobile use can then be measured more precisely. It will also give us an idea of the applications that benefit most from view consistency and the applications that need higher consistency.

We are currently implementing coordinating the view-entry databases for distributed entities. We are also examining suitable definitions of distributed entities. For non-overlapping accesses in time (such as using different machines at different times of the day) the view-entry database can be transfered from one machine, say on a PCMCIA card (or on a Java ring![2]), and integrated with the database of the new machine. We assume that view-entries are much smaller than files, and it is more useful to carry the consistency information rather than recently accessed files in the card. Files can then be loaded on demand from the previous machine if they are newer there.

References

[1] Rafael Alonso, Daniel Barbará, and Luis L. Cova. Using stashing to increase node autonomy in distributed file systems. In *Proceedings of the Ninth IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 12–21, October 1990.

[2] Stephen M. Curry. An introduction to the java ring. *Java World*, April 1998. `http://www.javaworld.com/javaworld/w-04-1998/jw-04-javadev.html`.

[3] Richard G. Guy, Gerald J. Popek, and Thomas W. Page, Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First International Conference on Network Protocols*. IEEE, October 1993.

[4] John S. Heidemann, Thomas W. Page, Jr., Richard G. Guy, and Gerald J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on Management of Replicated Data*, pages 2–5. IEEE, November 1992.

[5] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994.

[6] John Howard, Michael Kazar, Sherri Menees, David Nichols, Mahadev Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[7] James J. Kistler and Mahadev Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[8] Geoffrey H. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th Symposium on Operating Systems Principles*, St. Malo, France, October 1997. ACM.

[9] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[10] T. W. Page, R. G. Guy, J. S. Heidemann, D. Ratner, P. Reiher, A. Goel, G. H. Kuenning, and G. J. Popek. Perspectives on optimistically replicated peer-to-peer filing. *Software—Practice and Experience*, 1998. To appear.

[11] Jehan-François Pâris. Using volatile witnesses to extend the applicability of available copy protocols. In *Proceedings of the Second Workshop on Management of Replicated Data*. IEEE, November 1992.

[12] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[13] Mahadev Satyanarayanan, John H. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector, and Michael J. West. The ITC distributed file system: Principles and design. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 35–50. ACM, December 1985.

[14] Margo Seltzer. A new hashing package for UNIX. In *USENIX Conference Proceedings*. USENIX, January 1991.

[15] D.B. Terry, A.J. Demers, K. Petersen, M.J. Spreitzer, M.M. Theimer, and B.B. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 140–149, sep 1994.

[16] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, December 1995. ACM.

[17] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, August 1984.

[18] Erez Zadok and Dan Duchamp. Discovery and hot replacement of replicated read-only file systems, with application to mobile computing. In *USENIX Conference Proceedings*, pages 69–85, Cincinnati, OH, June 1991. USENIX.