Defining and Measuring Conflicts in Optimistic Replication*

John Heidemann Ashvin Goel Gerald Popek
University of California, Los Angeles

Technical report UCLA-CSD-950033

Abstract

Optimistic replication is often viewed as essential for large scale systems and for supporting mobile computing. In optimistic replication, updates can be made concurrently to different file replicas, resulting in multiple versions of the file. To recover from these *conflicting updates*, after-the fact *conflict resolution* actions are required to recombine multiple versions into one. This paper defines these concepts and discusses approaches to measure them in optimistically replicated systems.

Measurement of the number of conflicting updates and conflict resolution is important to judge the practicality of optimistic replication. An environment where conflicting updates are frequent will not be attractive since users cannot assume they have up-to-date data. Although many conflicts can be automatically resolved, some conflicts require user intervention; such conflicts cannot be too common. This paper shows an approach to measure the number of conflicting updates. From this measurement we derive the actual amount of work done by the user or system to resolve conflicts and the minimum amount of work required to resolve conflicts.

1 Introduction

Optimistic replication is a key approach to address a number of problems in distributed systems today. Data replication across multiple servers can make services more robust to failure. Replication is even more important when sharing data between distant locations where it can improve both performance and availability. Cooperative work requires data sharing: if automatic replication cannot be provided, or if it does not provide adequate quality of service, then information will be duplicated manually with correspondingly higher costs and error rate. This case can be illustrated by two examples. Two colleagues at MIT and Berkeley collaborate on writing a paper. Without replication their collaboration is dependent on the quality and reliability of their network connection as they access a single shared file. Alternatively they must duplicate relevant files on each coast and manually coordinate exchanging updates. As a second example, consider a single user with a laptop and an office computer. Without replication that person must manually insure that files stored on each computer are identical, remembering which files were last changed where. In each of these examples optimistic replication automatically maintains local copies of all relevant files, allowing local-disk performance for data stored in multiple locations.

Optimistic replication achieves high availability by allowing reads and writes whenever a single file replica is available. There are two costs to this policy. First, a read-operation can return stale data by reading from an out-of-date replica. Second, concurrent updates to different replicas can produce multiple, potentially conflicting versions of the same file. Without knowledge of data semantics, these two versions cannot be automatically merged into a single version. The *stale read* and the *conflicting update* problems are potential costs of optimistic replication. This paper focuses on the second of these costs as a means of evaluating optimistic replication.

Conflicting updates are an accepted and required cost of doing business in many environments [3, 2]. Bank automatic tellers and airline reservation systems are are the best known examples of commercial systems where performance and availability take precedence over conservative consistency. Several approaches have been suggested to detect and merge divergent databases [1, 5] in these systems.

*This work was sponsored by the Defense Advanced Research Projects Agency under contract N00174-91-C-0107. Gerald Popek is also affiliated with Locus Computing Corporation.

The authors can be reached at 3564 Boelter Hall, UCLA, Los Angeles, CA, 90024, or by electronic mail to `johnh`, `ashvin`, or `popek @cs.ucla.edu`.

File replication is another example where the benefits of optimistic replication can often outweigh the costs. There is substantial existing work analyzing the cost of conflicting updates in optimistic filing from several perspectives. Analysis of file system usage [4, 9] shows that file sharing is rare, and practical experience with optimistic filing [12] suggests that in many filing environments the number of conflicting updates can be extremely low. In addition, prior work has suggested that file system usage patterns can be exploited to minimize the chance of stale reads or conflicting updates [8], and file semantics can be employed to automatically recover from conflicting updates [12, 7, 10].

This paper augments existing empirical work by providing a theoretical framework for evaluating the cost of conflicting updates in optimistic replication. Intuitively, this cost is the ratio of the number of conflicting updates to the total number of updates in the system, modified by the difficulty of resolving each conflict. There are two problems associated with measuring the cost of optimism in this way. First, since *any* update could potentially be conflicting, an algorithm to detect a conflicting update must record *each* update. We will show that this algorithm requires information from each replica of the file at each update. Second, as we will see, this ratio is not an accurate measure of the cost of optimistic replication.

The paper begins by describing an abstract optimistic system based on version vectors. The version vector relationships are expressed in terms of a dominates graph. This relation is used to define a conflicting update and to count the number of conflicting versions in a static system. Section 4 develops an algorithm to count conflicting updates by examining each replica update. The next section presents a relationship between conflicting updates and conflict resolutions and uses this result to measure conflicting updates more efficiently. It also shows that the number of conflicting updates does not accurately reflect the total cost of optimistic replication. We then show that global state is required to correctly count conflicting updates and therefore our improved algorithm is optimal. Section 7 uses these results to re-examine the cost of optimistic replication, considering both the cost of handling conflicts in an actual system and the minimal possible cost. We conclude with a look at future directions.

2 System Model

Most optimistic file replication systems rely on version vectors or related techniques to record file update histories. This section describes version vectors, their properties, and the events which change them. To illustrate these concepts, we then map these abstract events to Ficus, an existing replicated file system [6].

2.1 File replication

An optimistically-replicated file-system consists of a number of logical files each stored as n separate file replicas. Conflicting updates are a property of a single file; notation in this paper refers to that file.

The version of the data present in each replica is captured by a *version vector* which uniquely identifies the update history of that replica [11]. Each replica p of this n-replica file has an n-element version-vector $vv_p[1 \dots n]$.

In principle, version vector elements are unbounded counters and n can grow arbitrarily large. In Ficus, elements are 32-bit integers and n is typically less than 20. While version vector length changes in a practical system, for simplicity this paper assumes that n is fixed. (Replica addition is possible by extending all version vectors with new zero elements as required; replica deletion requires a two-phase garbage collection algorithm.)

Before defining file events that affect version vectors, we need to define the relationships between version vectors.

2.2 Version vector relationships

There are several possible relationships between version vectors that represent file data.

Definition 1 *A version vector vv_p of file replica p* dominates *version vector vv_q if all elements of vv_p are greater than or equal to the corresponding elements of vv_q. That is, $vv_p \succeq vv_q$ iff*

$$\forall i \in [1 \dots n],\ (vv_p[i] \geq vv_q[i])$$

(Note that by this definition a file replica's version vector dominates itself.)

Definition 2 *A version vector vv_p* strictly dominates *version vector vv_q if vv_p dominates vv_q and at least one element of vv_p is strictly greater than the corresponding element of vv_q. That is, $vv_p \succ vv_q$ iff*

$$(vv_p \succeq vv_q) \wedge (\exists i \in [1 \dots n] :\ (vv_p[i] > vv_q[i]))$$

Definition 3 *A pair of version vectors are* compatible *if at least one version vector dominates the other. That is, $vv_p \sim vv_q$ iff*

$$((vv_p \succeq vv_q) \vee (vv_q \succeq vv_p))$$

Definition 4 *A pair of version vectors* conflict *if they are not compatible. That is, $vv_p \not\sim vv_q$ iff*

$$((vv_p \not\succeq vv_q) \wedge (vv_q \not\succeq vv_p))$$

2.3 File events and version vectors

In a *quiescent* file, all file replicas have the same version vector.

Definition 5 *A file is quiescent iff*

$$\forall p \in [1 \ldots n],\ vv_p = vv_1$$

There are three types of file events that can modify a version vector of the replicas of a file: *updates*, *version propagation*, and *conflict domination*. An update moves a file away from quiescence; the other events reduce file entropy, eventually returning a file to quiescence. Now we describe the three events.

Definition 6 Update: *An update to a file replica p modifies the version vector of that replica. It increments the* p^{th} *component of the version vector of p.*

$$vv_p[p]' \leftarrow vv_p[p] + 1$$

Updates to an optimistically replicated file results in file replicas with different version vectors. Concurrent updates to different replicas can result in conflicting file versions. Such a concurrent update is known as a *conflicting update*.

Definition 7 *A conflicting update to file replica p of a file occurs when that update causes* vv_p *to become in conflict with* vv_q *of some other file replica q with which it was compatible before the update. An update to p transforming* vv_p *to* vv'_p *is conflicting iff*

$$\exists q \in [1 \ldots n] :\ (vv_q \sim vv_p) \wedge (vv_q \not\sim vv'_p)$$

When an update occurs to replica p, the file has multiple, possibly conflicting, versions. Additional action must be taken to return the file to quiescence. *Update propagation* is the basic mechanism to reduce entropy.

Definition 8 Version propagation: *If file replica q dominates p* ($vv_q \succ vv_p$) *then in version propagation its contents are propagated and* vv_p *is set to* vv_q.

$$vv'_p \leftarrow vv_q$$

Version propagation is allowed only when one file replica dominates another. If this were not the case and propagation took place between two files in conflict then data could be lost as one update would be overwritten by the other.

When two replicas conflict, a second approach, *conflict domination*, is required. A special program must examine both files and merge their contents according to the semantics of the data, to produce a single new version. This new version is then written back over one of the replicas and its version vector is modified to reflect the merge.

Definition 9 Conflict domination: *Given two conflicting file replicas p and q, p is declared to have the "correct" data and is made to dominate q and then updated once.*

$$\forall i \in [1, n],\ vv_p[i]' \leftarrow \max(vv_p[i], vv_q[i]) + \begin{cases} 1 & i = p \\ 0 & i \neq p \end{cases}$$

2.4 An implementation

To gain a better understanding of how these operations behave in a practical system let us examine Ficus, an optimistically replicated file system developed and in use at UCLA [6].

In Ficus, there are several events that can cause version vectors to change:

Ficus file update At arbitrary times a user will update a file. This action results in a file replica update (Definition 6) immediately followed by *Ficus update notification.*

Ficus update notification Each file update is followed by a notification message sent to all other currently accessible file replicas. This message is a one-shot, best-effort attempt to trigger *Ficus update propagation* by other replicas.

Ficus update propagation When a site receives an update notification message it invokes version propagation (Definition 8) from the updated replica to the old replica. This propagation will fail if the versions conflict.

Ficus reconciliation To encourage files to reach quiescence we periodically compare file replica p with another file replica q. Replicas pass information indirectly through a gossip-based protocol [7] insuring that information exchanged in pairwise reconciliations eventually reaches all replicas. If $vv_q \succ vv_p$, p invokes version propagation from q. If this is not possible because the two file replicas conflict, *Ficus automatic conflict resolution* is attempted.

Ficus automatic conflict resolution When two file replicas conflict, the file name and its type are used to search for an applicable conflict resolver [12]. If a resolver is found, it is invoked to merge the replicas into one version. It updates one replica with the merged data and performs conflict domination, replacing the old replica with the merged one. It then invokes Ficus update notification to distribute the new version. If no automatic resolver is found than e-mail is sent to the file owner requesting *Ficus manual conflict domination.*

Ficus manual conflict domination As a last resort a user must manually merge file versions and invoke a utility which performs conflict domination and Ficus update notification.

These descriptions map our three fundamental file replica events to operations in a practical system.

3 File versions in a static system

Before examining conflicting updates in a dynamic system it is useful to examine the simpler static case. At a given point in time, we would like to determine which *significant* file replica versions represent unique data. Intuitively, versions are significant if they are not dominated by any other version. A non-significant version can be replaced by some other significant version which dominates it through version propagation. No data is lost in this process since all the updates in the non-significant version are present in the significant version.

Definition 10 *At a given time, version vector vv_p represents a significant version iff*

$$\nexists q \in [1 \ldots n] : (vv_q \succ vv_p)$$

Multiple significant versions arise from conflicting updates and must be eventually merged with conflict domination.

It is helpful to represent the relationship between file versions graphically. To do so, we will further investigate the dominates relation.

Theorem 1 *The dominates relation is a partial order.*

Proof: From inspection of Definition 1, the dominates relation is reflexive, antisymmetric and transitive and thus a partial order. □

Let us consider the graph G_0 induced by the dominates relation. Since by Theorem 1, the dominates relation is a partial order, G_0 is a DAG and may have multiple terminal vertices. Now consider the graph $G = (V, E)$ a transitive reduction graph of G_0.

$$V = \{vv_p \mid p \in [1, n]\}$$

$$\begin{aligned} E &= \{(vv_P, vv_Q) \in V \times V \mid (vv_Q \succ vv_P) \\ &\quad \wedge \nexists vv_R \in V : (vv_Q \succ vv_R \succ vv_P)\} \end{aligned}$$

G is obtained by removing edges from G_0 such that the closures of G and G_0 are the same. Figure 1 shows the dominates graph G for a five replica file. This graph has the property that all terminal nodes represent significant versions.

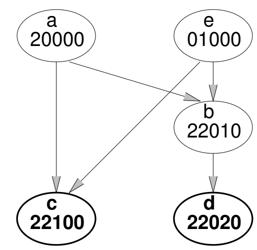

Figure 1: The graph $G = (V, E)$ for a five replica file. Ovals represent file versions; bold ovals represent significant versions.

Theorem 2 *Terminal nodes in the dominates graph G are equivalent to significant file versions.*

Proof: If vv_p is significant, by Definition 10 it is a maximal element of the dominates partial order. Graph G is induced by the dominates relationship. Maximal elements of a partial order correspond to terminal nodes in their induced graph. □

An example of a dominates graph can be seen in Figure 1. Replicas c and d are the significant versions. In the figures, the first element of the version vector corresponds to a, the second element corresponds to b and so on.

There are two important facts to note about G. First, determining G requires an atomic picture of version vectors of all file replicas. Second, since the same version of file data can exist at multiple replicas, there is a one-to-many mapping between file versions and file replicas. For example, in Figure 1, if version propagation took place from c to a, both would be merged into a single file version. Throughout this paper we use uppercase to refer to file versions and lowercase to refer to particular replicas.

4 A Simple Algorithm to Count Conflicting Updates

There is a close relationship between conflicting updates and whether replicas are significant or not. We will show that conflicting updates are updates which occur to non-significant replicas, and use this result to develop an algorithm to count conflicting updates. To develop this relationship we require two preliminary results.

Lemma 1 *The local component of replica p, $vv_p[p]$, is greater than or equal to the p^{th} component of any other replica.*

$$\forall q \in [1 \ldots n],\ (vv_p[p] \geq vv_q[p])$$

Proof: Let us look at the three system events. An *update* increments the local version vector component, thus the local component must be greater than any other. A *version propagate* increments the version vector of q only if p dominated it. So the p^{th} component of q can not be greater than that of p. Finally a *conflict dominate* assigns to the version vector of q, the maximum of the version vectors of p and q and then updates q once. Again the p^{th} component of q can not be greater than that of p. □

Lemma 2 *After an update to replica p, vv_p is significant.*

Proof by contradiction: An update to p changes vv_p to vv'_p. Since the update increases the local version vector component, $vv'_p \succ vv_p$. Suppose vv'_p is non-significant. Then there exists a replica q which dominates p', by the inverse of Definition 10. Therefore, $\forall i \in [1, n]$, $vv_q[i] \geq vv_p[i]' \geq vv_p[i]$. However by Lemma 1, $vv_p[p] \geq vv_q[p]$. The update to replica p makes $vv_p[p]' > vv_p[p] \geq vv_q[p]$. Thus q can not dominate p', a contradiction. □

Using these results we can now directly relate conflicting updates and replica significance.

Theorem 3 *An update is a conflicting update iff it is made to a non-significant replica.*

Proof: $\Rightarrow$: To prove this part of the theorem, let us show that an update to a significant replica p implies that there is no conflicting update. An update to a significant replica does not change the dominates graph.

If the significant replica p dominated some other replica q then it still dominates that replica after this update since the update increases the version vector of p. If the significant replica p was in conflict with replica q then there $\exists i : vv_q[i] > vv_p[i]$. Component i could not be equal to p by Lemma 1. Since only $vv_p[p]$ changes during the update, $vv_q[i]$ is still greater than $vv_p[i]'$. Similarly the component of p that was greater than the corresponding component of q is still greater. Thus p' still conflicts with q. Thus no replica has come in conflict with some other replica with which it was not in conflict before the update.

$\Leftarrow$: An update to a non-significant replica p makes that replica significant by Lemma 2. Before the update, some other replica q dominated the old p. After the update both the replicas are significant. Their version vectors are not equal because by Lemma 1, $vv_p[p] \geq vv_q[p]$. The update to replica p makes $vv_p[p]' > vv_p[p] \geq vv_q[p]$. Thus they are in conflict after the update although they were not before the update. □

```
algorithm simple_cu_count
    input: the time range [u, v]
    output: CU_[u,v]
begin
    T_update := all update events over [u, v]
    CU_[u,v] := 0
    foreach (t ∈ T_update)
    begin
        G(t⁻) = G(just before t)
        G(t⁺) = G(just after t)
        foreach (p ∈ [1, n])
        begin
            if (vv_p is significant in G(t⁺)
                ∧ vv_p is non-significant in G(t⁻))
                then CU_[u,v] := CU_[u,v] + 1
        end
    end
    return CU_[u,v]
end
```

Figure 2: The basic algorithm for detecting conflicting updates. G(t) is the dominates graph at time t.

With this relationship we can now develop a simple algorithm to detect conflicting updates (see Figure 2). This algorithm examines each update event on a replica and classifies that replica as significant or non-significant.

While this algorithm implements Theorem 3 to calculate the number of conflicting updates in a straightforward manner, it is not satisfactory for use in a practical system. A first problem with the algorithm is that it employes a global snapshot (to calculate G) which is expensive in number of messages and involvement of all replicas. Updates are quite frequent in a replicated file system, so the cost of a global snapshot with each update would be prohibitively expensive in any practical system. An even more serious problem with this algorithm is that optimistic replication is most useful in environments where communication between replicas is occasionally or even primarily unavailable. When communication is unavailable for hours, days, or weeks, an algorithm requiring global communication is not practical. Moreover, conflicting updates are most likely to occur when version propagation fails because of communication loss. It is at these very times that we need global communication to determine whether a conflicting update has been made.

We would like to improve this algorithm by addressing both of these problems. We examine each in the sections that follow.

5 A Better Algorithm to Count Conflicting Updates

Even though the events in T_{update} are *sufficient* to properly determine the number of conflicting updates, many of these events are not *necessary*. For example, experience with Ficus suggests that the vast majority of updates do not create conflicting updates [12].

Fortunately in many cases we can determine the same information much more efficiently. We next show that there is a relationship between the number of conflicting updates and the number of conflict domination events. This relationship can be exposed by considering how these events change the number of significant versions over time. We then will present a revised algorithm making use of these results.

5.1 File versions in a changing system

File events change number of significant versions over time, so it is useful to define ISV as a function of time, ISV(t). The number of significant versions of the file changes in a stepwise manner over the lifetime of a file; it is discontinuous when it changes. To characterize this discontinuous nature let $c(t) = \lim_{\epsilon \to 0}(ISV(t+\epsilon) - ISV(t-\epsilon))$, the change in $ISV(t)$ at time t.

We assume that file events do not occur simultaneously, so $c(t)$ is either -1, 0, or $+1$.

Let $T_u[u,v]$ and $T_d[u,v]$ be the set of time instants over the period $[u,v]$ when ISV(t) increases and decreases in value, respectively. Then $T_u[u,v]$ and $T_d[u,v]$ can be defined as:

$$T_u[u,v] = \{u < t < v \mid c(t) > 0\}$$
$$T_d[u,v] = \{u < t < v \mid c(t) < 0\}$$

(We will write T_u and T_d when we are not concerned about the time interval.)

Figure 3 shows an example of an ISV(t) graph. The sets T_u and T_d are indicated by different arrows across the top of the graph.

5.2 Characterizing file events

Now that we have defined the time instants during which ISV(t) changes, let us examine each system event that can alter file version vectors to see which of these events affect ISV(t).

Update: By Lemma 2, replica p is significant after an update. This update can be conflicting or non-conflicting:

Conflicting update: A conflicting update is an update to a non-significant node (3). The update makes it a significant node without changing any other dominates relationship in the dominates graph, and so the number of significant nodes or ISV(t) increases by one. Conflicting updates are thus a part of T_u.

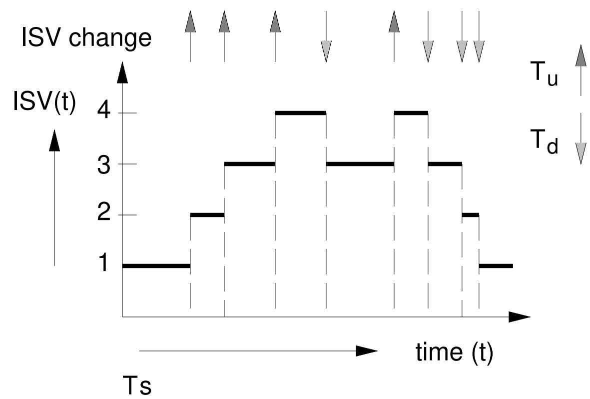

Figure 3: ISV plotted as a function of time.

Non-conflicting update: A non-conflicting update is an update to a significant node. This update does not change the dominates graph. Thus ISV(t) does not change.

Version propagation: By Definition 8, only a file replica with a non-significant version vector can be changed by version propagation. This non-significant version vector becomes equal to some other vector which had dominated it earlier. This does not lead to a change in ISV(t).

Conflict domination: Consider the dominates graph G. Let a conflict domination occur at time t between a pair of replicas p and q. A conflict domination event could be of three types:

S-S**:** In this interaction, both p and q are significant. A S-S event combines the two significant versions and forms a single new one, thus reducing ISV(t) by one. Figure 4 shows a S-S event. The conflict domination event is represented by the dotted arrow. Replica d is made to dominate c and updated locally. By Definition 9 its new version vector is $max(22020, 22100) + 00010 = 22130$ and $ISV(t)$ has decreased from 2 to 1. S-S events are therefore part of T_d.

S-NS**:** In this interaction, replica p is significant while q is non-significant. A S-NS event

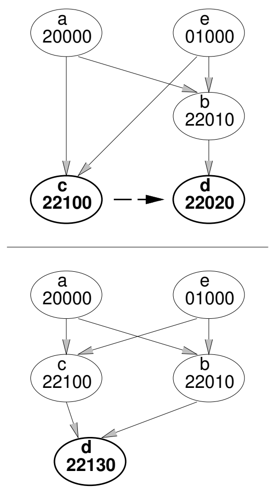

Figure 4: A significant/significant two way conflict-domination event (represented by the dotted arrow).

does not change the number of significant versions. One significant version is replaced by another one. Thus ISV(t) does not change in this event. In Figure 5 we show this event. Replica b is made to dominate c and updated locally. Its new version vector is $max(22010, 22100) + 01000 = 23110$.

NS-NS**:** In this interaction, both p and q are non-significant. An NS-NS event causes the creation of a new significant node from two non-significant node thus increasing the number of significant versions or ISV(t) by one. Figure 6 shows this event. Replica e is made to dominate a and updated locally, and its version vector becomes $max(01000, 20000) + 00001 = 21001$. $ISV(t)$ increases from 2 to 3. NS-NS events are therefore part of T_u.

Table 1 summarizes how file events affect ISV(t).

5.3 A better algorithm

We have shown how the different file events affect the number of significant versions. To develop a better algorithm for counting conflicting updates, we must show the relationship between conflicting updates and conflict domination events. This requires one more relationship regarding changes to $ISV(t)$.

Lemma 3 *If, at two times u and v, $ISV(u) = ISV(v)$, then $|T_u[u, v]| = |T_d[u, v]|$.*

Proof: $|T_u[u, v]|$ is the sum of the positive changes, and $|T_d[u, v]|$ is the sum of the negative changes in the

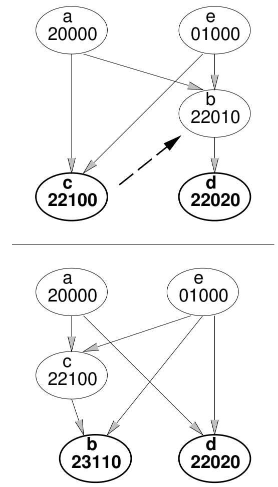

Figure 5: A significant/non-significant conflict domination event.

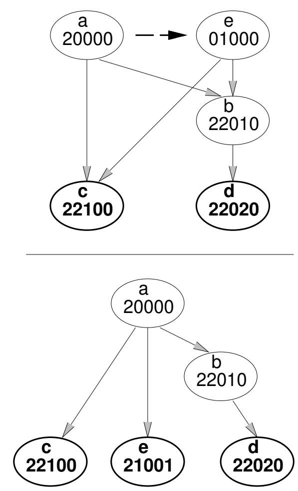

Figure 6: A non-significant/non-significant conflict domination event.

Event	ISV**(t)**
update:	
conflicting	$+1$
non-conflicting	0
version propagation:	0
conflict domination:	
S-S	-1
S-NS	0
NS-NS	$+1$

Table 1: Variation in ISV for each file event.

ISV graph. Since the end points are equal, these values must be equal. □

We can now prove the theorem that will be the basis of our improved algorithm.

Theorem 4 *If, at two times u and v, $ISV(u) = ISV(v)$, then $CU_{[u,v]} = |T_{\text{S-S}}[u,v]| - |T_{\text{NS-NS}}[u,v]|$.*

Proof: Since $ISV(u) = ISV(v)$, we know that $|T_u[u,v]| = |T_d[u,v]|$, from Lemma 3. From Table 1 we know that if $T_{CU}[u,v]$, $T_{\text{S-S}}[u,v]$ and $T_{\text{NS-NS}}[u,v]$ are the sets of conflicting-update, S-S, and NS-NS conflict domination events, then

$$T_u[u,v] = T_{CU}[u,v] \cup T_{\text{NS-NS}}[u,v]$$
$$T_d[u,v] = T_{\text{S-S}}[u,v]$$

Therefore $T_{CU}[u,v] \cup T_{\text{NS-NS}}[u,v] = T_{\text{S-S}}[u,v]$. □

From this theorem we derive the algorithm shown in Figure 7, So this algorithm takes advantage of the fact that we can derive $CU_{[u,v]}$ from $T_{\text{S-S}}$ and $T_{\text{NS-NS}}$. $T_{\text{S-S}}$ and $T_{\text{NS-NS}}$ occur only during conflict domination events ($T_{\text{conflict dominate}}$) while $T_{CU}[u,v]$ can occur during any update event (T_{update}). This algorithm represents a significant improvement in practice because experience with Ficus suggests that $|T_{\text{conflict dominate}}| \ll |T_{\text{update}}|$. Over a nine-month period with Ficus running on approximately a dozen workstations, 14,142,241 file updates occurred, while only 489 conflict dominations were required [12].

6 Optimality of the Global Snapshot Algorithm

We have shown that the number of conflicting updates is dependent on changes to $ISV(t)$, and how we determine *CU* by examining conflict domination events. Our current algorithms require a global snapshot of version vectors from each replica at each conflict domination event to determine *CU*. We now prove that this snapshot is essential to correctly measure resolutions.

Theorem 5 *A change in the $ISV(t)$ can be determined only by taking a global snapshot of all replica version vectors.*

Proof: We've shown that ISV(t) decreases only at conflict domination events, so consider such an event occurring between two replicas p and q. The improved algorithm requires knowing when a S-S or NS-NS conflict-domination event occurs. This requires knowing whether p and q are significant or not. We prove that it is necessary to take a global snapshot to determine whether p or q are significant by showing that without global information, errors can result.

```
algorithm better_cu_count
    input: the time range [u, v]
    pre-condition: ISV(u) = ISV(v)
    output: CU_[u,v]
begin
    T_conflict dominate := all conflict dominate
                events over [u, v]
    S-S := 0
    NS-NS := 0
    foreach (t ∈ T_conflict dominate)
    begin
        ISV(t⁻) := ISV(just before t)
        ISV(t⁺) := ISV(just after t)
        c(t) := ISV(t⁺) − ISV(t⁻)
        if (c(t) > 0) then
            NS-NS := NS-NS + 1
        if (c(t) < 0) then
            S-S := S-S + 1
    end
    CU_[u,v] := S-S − NS-NS
    return CU_[u,v]
end
```

Figure 7: A better algorithm for detecting conflicting updates.

Suppose we do not take a global snapshot, let us assume that we do not record the version vector of replica r during the conflict domination. Assume that $vv_p \succeq vv_r$ or $vv_q \succeq vv_r$. Let us also assume that after comparing all the version vectors except r, replicas p and q were both significant before the domination. If we now postulate an independent update occurring at r just before the domination, this update makes replica p non-significant. The conflict domination now occurs between S-NS replicas rather than S-S replicas, and so does not decrease *ISV*. Since an independent update to another replica can spoil *ISV* measurement, an *ISV* measurement must have information of all file version vectors. □

7 Cost of Optimistic Replication

We have presented two algorithms for counting conflicting updates. While the ratio of conflict updates to total updates provides one means of judging costs of optimistic replication, it is not the most direct way to measure the cost perceived by users of such a system. To a user a conflicting update appears no different from any other update; the real cost of optimism is the amount of effort required to remove all conflicts from a conflicted file.

Transforming a file with multiple conflicting versions into a file with a single significant version is *conflict resolution*. A conflict resolution event occurs each time $ISV(t)$ decreases.

We next explore the cost of conflict resolution from two perspectives. First we consider how to measure *actual cost*, the amount of time spent attempting to repair conflicts. We then examine the theoretical *minimal cost*, the ideal amount of work required to return a conflicted file to a single significant version.

7.1 Actual cost

The *actual cost* of optimistic replication is the amount of extra work expended attempting to return the file to a single significant version. Each conflict domination is an attempt to resolve a conflict and therefore is part of the actual cost. In the terminology described in Section 2.4, all Ficus automatic and manual conflict domination events contribute to this cost.

Unfortunately, not all conflict dominations are successful at moving the file to a single significant version. The analysis of conflict domination events (Table 1) shows that only S-S conflict dominations are helpful. Conflict dominations between S-NS replicas do not improve matters, while NS-NS dominations actually introduce a new conflict.

We consider all conflict domination events as part of the actual cost because even though not all such events resolve a conflict, they all represent work undertaken by the system in an *attempt* to resolve a conflict. Conflict domination events should therefore be considered as an upper bound to the minimum cost required to provide a single significant file version.

7.2 Minimal cost

While the actual cost is useful to evaluate the effort spent resolving conflict in a real system, it does not represent a tight bound of the minimum work required. Figure 8 shows how a poor choice of conflict dominate events can prevent $ISV(t)$ from ever reaching one while a better choice could resolve the conflct with only one domination. The NS-NS conflict domination in the top of the figure fails to resolve the conflict and in fact produces a new one. After the S-S domination in the middle of the figure, we have returned to the initial state. This sequence of events could be repeated infinitely. The problem here is that conflict domination between NS-NS versions produces a new file version which then requires further conflict domination.

The *minimal cost* assumes that all conflict domination events are conflict resolutions. Minimal cost is therefore the number of S-S conflict domination events. The minimal cost can be measured by the number of S-S events counted in the improved algorithm (Figure 7).

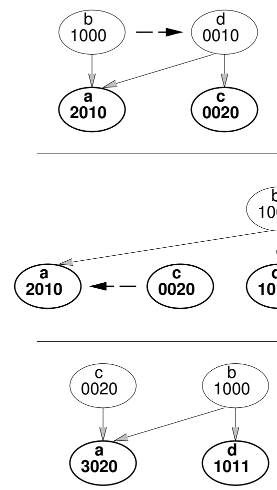

Figure 8: Poor choice of conflict domination prevents quiescence.

If new file updates are prohibited after time t, the minimal cost of resolving all conflicts for a file is simply $ISV(t) - 1$.

8 Future Work

This paper defines how optimistically replicated files interact, presents two algorithms to measure conflicting updates, and shows that any such algorithm requires global information. These results provide several insights into our existing work with optimistic replication and suggest several directions for future work.

8.1 Off-line measurement of conflicting updates

We have demonstrated that global knowledge is required to correctly count conflicting updates. Unfortunately, the communication requirements for global snapshots does not match the communications guarantees available in optimistic systems. With mobile computers, replicas may communicate infrequently, so as n grows beyond a small number, replicas may either never contact each other or communicate once in days or months.

Our simple algorithm for counting conflicting updates can be executed without taking on-line snapshots if it is decomposed into two parts. The on-line portion of the algorithm would log each file update event. These logs would be merged off-line and then analyzed to count conflicting updates. Logs would be gathered at a central site via a protocol robust to intermittent communication (for example, gossip or store-and-forward). By separating the algorithm into on-line and off-line components we make it robust to intermittent communication. Unfortunately, a complete log of all updates would be very large.

Our improved algorithm required snapshots at conflict domination events only, rather than all updates. This suggests that we study off-line algorithms that require logging fewer events.

8.2 Improved conflict domination

Conflict domination attempts to merge conflicting versions. Unfortunately, we have shown that NS-NS conflict domination events actually create new significant file versions. To minimize this possibility we plan to modify Ficus automatic conflict domination to examine all currently available replicas before performing a conflict dominate. Although this examination cannot guarantee that NS-NS domination events do not occur, it will minimize their possibility.

9 Conclusion

This paper has defined relationships between version vectors and file replicas in an optimistically replicated system. These relationships can be expressed as a dominates graph. We use these relationships to count the number of conflicting versions in a static system and to develop an initial and an improved algorithm to count conflicting updates. Finally we have shown that the number of conflicting updates does not directly relate to the cost of resolving conflicts in an optimistic system. Instead we found that that both actual and minimal costs depend on how conflicts are resolved.

We believe this paper makes three contributions to the field. First, we provide a clear framework for reasoning about conflicts in optimistic replication. Second, we present algorithms which can be used to measure the number of conflicts in an existing system. We have shown that global state is required by these algorithms and have suggested how the algorithms can be adapted to cope with intermittent communication. More importantly, we have shown how conflicting updates and conflict resolutions relate to the actual costs observed by the user of an optimistic system. These algorithms provide a clear means to judge the costs of optimism and therefore to compare its costs and benefits. Finally, the insights gained through definitions and measurements developed in this paper have suggested several changes we would like to make to an existing replicated system.

We believe that many truly scalable distributed systems of the future will require optimistic replication. A key question in the deployment of such systems is the costs of optimism. We believe that this paper provides a framework for analyzing these costs.

Acknowledgments

The authors would like to thank Dave Ratner and Greg Skinner for their contributions toward this work. We would also like to thank Elizabeth Borowsky for her comments on an early draft of this paper. The ideas presented here ideas grew from discussions regarding the Ficus replicated file system including Michial Gunter, Ted Kim, Geoff Kuenning, Peter Reiher, Qian Qin, John Salomone, and S. Suresh.

References

[1] B. Blaustein, H. Garcia-Molina, D. Ries, R. Chilenskas, and C. Kaufman. Maintaining replicated databases even in the presence of network partitions. In *Proceedings of the IEEE EASCON Conference*, September 1983.

[2] Stefano Ceri, Maurice A. W. Houtsma, Arthur M. Keller, and Pierangela Samarati. The case for independent updates. In *Proceedings of the Second Workshop on Management of Replicated Data*, pages 17–19. IEEE, November 1992.

[3] Alan R. Downing, Ira B. Greenberg, and Jon M. Peha. OSCAR: a system for weak-consistency replication. In *Proceedings of the Workshop on Management of Replicated Data*, pages 26–30. IEEE, November 1990.

[4] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report TR-177, University of Rochester, March 1986.

[5] Hector Garcia-Molina, Tim Allen, Barbara Blaustein, R. Mark Chilenskas, and Daniel R. Ries. Data-patch: Integrating inconsistent copies of a database after a partition. In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, pages 38–44, October 1983.

[6] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. Implementation of the Ficus replicated

file system. In *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[7] 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.

[8] 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*. IEEE, November 1992.

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

[10] Puneet Kumar and Mahadev Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 66–70, Napa, California, October 1993. IEEE.

[11] 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.

[12] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, pages 183–195. USENIX, June 1994.