Dynamic Version Vector Maintenance*

David Ratner Peter Reiher Gerald J. Popek†
Department of Computer Science
University of California, Los Angeles

Abstract

Version vectors or variants thereof are used in many distributed file systems to track updates, and are the main data structure behind optimistic replication. Mobile computing, however, places new strains on the version vector. Decreased bandwidth and increased replication factors will exacerbate the scaling problems that version vectors have always suffered from, but until now have never been important enough to remedy. Here we present algorithms for dynamic version vector maintenance, which dramatically improves their scalability and therefore the scalability of the entire replication system.

1 Introduction

Mobile computing is rapidly becoming standard in all types of environments: academic, commercial, and private. Widespread mobility impacts multiple arenas, but one of particular importance is data replication. Replication is especially important in mobile environments, since disconnected or poorly connected machines must rely primarily on local resources. The monetary costs of communication when mobile, combined with lower bandwidth, higher latency, and reduced availability, effectively require that important data be stored locally on the mobile machine. In the case of data shared between multiple mobile users or between mobile and stationary machines, replication is often the best and sometimes only viable approach.

Conservative replication strategies such as primary-site [12] or majority-vote [14] are generally not appropriate in mobile scenarios, due to communication restrictions. In contrast, optimistic strategies have proven themselves as the correct model in the mobile context, demonstrated by systems such as CODA [11], LITTLE WORK [2], RUMOR [9], and BAYOU [13]. Optimistic strategies require a method of tracking updates to individual object replicas, allowing future comparisons between replicas, propagation of the latest version, and detection of conflicting versions. Most optimistic strategies base their update-tracking mechanism on the version vector [6] or some variant thereof. Version vectors are easy to use and have proven themselves as one of the main mechanisms behind optimistic replication, but are not without their problems.

In brief, the version vector is an array of length equal to the number of replicas. Each replica $\mathtt{R}_j$ independently maintains its own vector, and tracks in position $\mathtt{i}$ the number of updates generated by each replica $\mathtt{R}_i$ that $\mathtt{R}_j$ knows about. In other words, each element is a monotonically increasing counter, and each counter tracks the total number of known updates generated by the corresponding replica. Consistency is enforced by comparing version vectors and copying the data and its associated version vector when appropriate.

The key problem is that version vectors are unbounded in size. Many have noted the scaling problems associated with version vectors, and partly for that reason have opined that wide-scale peer replication does not scale [10]. Additionally, simulations of RUMOR [15] have demonstrated the scaling problems concretely. However, we believe peer replication to be essential in the mobile context for a host of reasons, and even in client-server systems such as CODA there are still a number of peer servers constructing the replication backbone. [and it would be useful in other arenas too] We have therefore designed a *dynamic* method of managing version vectors, dramatically reducing their size and impact even in face of large scale peer replication. We call these methods dynamic version vector *expansion* and *compression*. They are currently being implemented at UCLA as part of a replication system specifically aimed at mobility called ROAM, built using the *ward model* [7].

We first provide an example of version vector use. We then present an overview of the ideas behind dynamic version vector maintenance, and delve into the

*This work was sponsored by the Advanced Research Projects Agency under contract DABT63-94-C-0080. The authors can be reached at the Department of Computer Science, UCLA, Los Angeles, CA 90095, or by email to {ratner,popek,reiher}@cs.ucla.edu.

†Gerald Popek is also affiliated with Platinum *technology*.

details of both expansion and compression. We conclude with a few words regarding future work.

2 Version Vector Example

Assume there are n replicas of a given object. Each replica $\mathtt{R}_n$ maintains its own version vector $\mathtt{V}_n$. Each version vector has n elements: element i of vector $\mathtt{V}_j$ represents the number of updates generated at replica $\mathtt{R}_i$ and known at replica $\mathtt{R}_j$.

For example, let $n = 3$ and let the version vectors be as follows:

replica 1	$\mathtt{V}_1 = \{2,\ 0,\ 3\}$
replica 2	$\mathtt{V}_2 = \{2,\ 1,\ 3\}$
replica 3	$\mathtt{V}_3 = \{2,\ 1,\ 5\}$

A vector $\mathtt{V}_x$ is said to *dominate* $\mathtt{V}_y$ when $\mathtt{V}_x$'s elements are all pairwise greater than or equal to $\mathtt{V}_y$'s. In the example, $\mathtt{V}_3$ dominates $\mathtt{V}_2$ which itself dominates $\mathtt{V}_1$, identifying that replica 2 has more recent data than replica 1, but that replica 3's data is the globally most recent version. Should replica 2's version vector had instead been $\mathtt{V}_2 = \{2,\ 2,\ 3\}$, then it would have still dominated $\mathtt{V}_1$, but $\mathtt{V}_3$ would no longer dominate $\mathtt{V}_2$. $\mathtt{V}_2$ and $\mathtt{V}_3$ are said to *conflict*, representing data versions that incorporate concurrent updates. Conflict resolution [5, 8] will resolve the conflict, resulting in one of the version vectors dominating the other.

3 Dynamic Maintenance

As seen from the above example, as more and more replicas join the system, each replica's version vector increases in size. Mobility will most likely lead to increased replication factors, due primarily to the requirement that data accessed when mobile must be local. Connectivity cannot be relied upon, and even when it does exist it is typically expensive and degraded. Each mobile user requires a local replica; in addition, replicas must also be stored on stationary machines for non-mobile personnel and system administration activties like back-ups. Finally, *appliance* mobility [3], moving from device to device or system to system, could potentially lead to situations where users have multiple devices, each one of which is capable of storing and updating replicated data. All of these scenarios become more difficult if at all possible without an underlying replication mechanism that scales well.

Toward that end, We have made two key observations regarding version vectors:

1. Updates typically occur in a few isolated "hot spots", as opposed to concurrently at all replicas.

2. Once all replicas have the same element for replica $\mathtt{R}$, replica $\mathtt{R}$'s element is no longer relevant for version vector comparison purposes.

From these two observations, we realized that *dynamic* maintenance of the version vector would dramatically increase its scalability. By dynamic we mean:

1. Rather than pre-allocating a vector element for each replica $\mathtt{R}$, we can instead dynamically *expand* the vector and create $\mathtt{R}$'s element when $\mathtt{R}$ generates its first update. Zero-valued elements in the vector are insignificant in the comparison routines, and can therefore be trivially removed.

2. Once all replicas have the same value for replica $\mathtt{R}_j$ in their vector, $\mathtt{R}_j$'s position can be removed from each vector with no loss of distinguishing information. Should replica $\mathtt{R}_j$ generate future updates, it can re-expand the vector via dynamic expansion. The challenge, and one of the main contributions of this paper, is to provide dynamic version vector *compression* on $\mathtt{R}_j$'s element while still allowing $\mathtt{R}_j$ to generate updates during the compression process.

We describe dynamic expansion and compression separately, though the true benefit is achieved only by utilizing both mechanisms.

4 Dynamic Vector Expansion

Zero-valued elements in the version vector add no distinguishing information. Given two version vectors V_1 and V_2 and a dominance relation $\mathtt{D}$ between them, such as V_1 dominates V_2, adding unmatched zero elements (elements in one vector with no corresponding element at the other) to either vector leaves $\mathtt{D}$ unchanged. Additionally, our experience with replicated file systems like Ficus and Rumor seems to indicate that some replicas *never* generate updates. It is important, especially as we increase scale, that the non-writers do not increase the size of the version vector.

However, we cannot in general predict the set of writers, and it is important to give each replica the *ability* to generate updates, regardless of how often updates occur. Therefore we must preserve the ability for each replica to generate updates while taking advantage of the fact that many replicas will generate no updates.

The approach we take is to never store zero elements, and to dynamically *expand* the vector when a replica generates its first update. As a result, replicas that never generate updates are never mentioned in the version vector, though they reserve the ability to generate updates at any time.

Dynamic version vector expansion requires minor changes to the definition of a version vector element. A basic version vector is simply a list of counters, one counter per replica. Since new replicas typically join the system optimistically by communicating with only one other replica, version vectors will not always be the same length; identifying which element belongs to which replica is vital. Since the order can be pre-defined to match a specific replica ordering, the replica identifiers themselves need not be present in the vector. A given replica's position can be identified using the pre-defined mechanism.

However, once we remove the zero elements and institute dynamic vector expansion, the ordering in the vector becomes indeterminate. Master lists, sorting, and other approaches cannot remedy the situation. An associative array indexed by replica identifier is required to adequately solve the problem.

Using an associative array increases the size of the version vector by the size of the key. Of course, we are also shortening the version vector by removing zero elements. However, the real benefit from dynamic expansion occurs when used in combination with dynamic vector compression, described below. When both dynamic compression and expansion are used together, the version vectors tend to list only the replicas that are actively generating updates. Therefore, while the space overhead of the associative array may seem excessive when considered by itself, dynamic version vector management as a whole has the ability to dramatically decrease the size of the version vector.

5 Dynamic Vector Compression

Studies of replicated file systems [5, 8] illustrate that conflict rates are generally quite low, meaning that concurrent writes rarely occur. Experience with replication systems like Ficus, Rumor, and Coda seems to expand upon this notion. Objects tend to have "hot spots" or small collections of writers within the large quantity of replicas. While the hot spots may change over time, it is rare to see a widely replicated object that is consistently updated by everyone. Simulation data by Wang supports the hot-spot notion. In studies of productivity environment data taken at Locus Computing [4], Wang found that replicated objects tend to have only a small set of common writers at any one time [15].

In short, while we must provide the *ability* for everyone to generate updates, it seems rare that a significant percentage of the replicas are simultaneously trying to do—an assertion we call the *hot-spot hypothesis*. The hot-spot hypothesis provides intuition into the expected update patterns, and illustrates that the elements for the *cold* replicas, the ones not in the current hot-spot, play only a minor role in the version vector. When a cold replica generates an update or even a burst of updates, its element quickly stabilizes at all replicas, since by definition that replica does not typically generate updates. Once the element stabilizes it no longer plays an important or distinguishing role in the version vector comparison algorithm, and could be removed everywhere with no loss of versioning information.

Note that the *hot* versus *cold* distinction is on a per-object, per-replica basis. A given site will typically be in the hot spot for some objects and not for others.

The situation is therefore ripe for dynamic version vector compression. We periodically execute a distributed algorithm to remove the elements generated by the cold replicas. Only the cold replicas are good candidates for comression, since the hot replicas are usually expected to generate more updates "soon."

We will first describe the algorithm requirements and a brief overview. Then we will describe the changes to the version vector definition, discuss the algorithm assumptions, and describe the details of the algorithm itself. We conclude with a series of remarks regarding the best time to initiate compression.

5.1 Algorithm requirements

First and foremost, the algorithm must be safe. For any two version vectors V_1 and V_2 and a dominance relation `D` between them, if the comparison of V_1 and V_2 yields `D` before compression, then barring other updates the comparison must also yield `D` after compression.

Additionally, we must not prevent updates by any replica, hot or cold, during the algorithm's execution. Even while we are attempting to remove replica `R`'s element, we must allow updates by `R` so we can maintain the high availability provided by optimistic strategies.

5.2 Algorithm overview

The algorithm works on a per-element basis, though multiple elements can be removed at once. For each element being removed, the algorithm achieves consensus on a value to be subtracted from the existing element. After consensus is reached, each replica subtracts the common value from the specified element. If the resulting element is zero, the entire element is removed from the vector. The consensus algorithm ensures that no replica with element x ever attempts to subtract a value greater than x.

Correctness is guaranteed by tagging each version vector (not each element) with a sequence number that counts the number of times compression has occurred, ensuring that vectors V_1 and V_2 do not accidently compare elements e_1 and e_2 when one of them has been compressed and the other has not. The in-place issue is resolved by associating with each element being compressed a "spare" position, allowing consensus to be achieved independently of new updates.

5.3 Data structure modifications

There are three substantial changes to the version vector structure. The first is the addition of a "spare" element, used only during the compression process. The second is a one-phase bit vector used to detect the correct completion of the compression algorithm. The third is a sequence number attached to the version vector to prevent communication between vectors when one has compressed and the other has not. Each will be discussed in turn. Note that the bit vector is described as if each individual element has its own, though multiple instances of the algorithm can easily be coalesced, so that the entire version vector only requires one bit vector.

5.3.1 The spare element

A typical version vector element e is simply a monotonically increasing integer value $\{a\}$.[1] Dynamic vector compression additionally requires that, while compression on a given element is occurring, the element take the form $\{a, b\}$, where both a and b are both integral values. b is referred to as the "spare" element. When compression starts, elements of the type $\{x\}$ are changed to $\{0, x\}$. Consensus is achieved using the "b-value" (the value in the b position of the element) while future updates increment the a-value; in this way we allow updates during the compression process. The b-value can be dynamically allocated, so it occupies no additional space except when compression is occurring.

Since the basic format of a version vector element has changed, we must modify the compare function, to account for the b-value. The approach used is to sum a and b, and use $a + b$ as the element's total for comparison purposes. The sum $a+b$ accurately reflects the current version, because the element had the value b when compression started, and since that time has received a more updates.

[1] Dynamic vector expansion (Section 4 above) adds a replica identifier to the element, but we ignore the replica identifier for now.

5.3.2 The one-phase bit vector

We cannot complete compression until all replicas have achieved consensus on the b-value. We use a one-phase bit vector to determine when consensus has occurred. In general, one phase is not enough to determine consensus [1]. However, in our case the actual b-value itself provides the effect of two phases, since consensus on the value stored in b acts as phase 1, and the bit vector itself actually functions as phase 2. The rules for setting bits in the bit vector are intricate, but in general bits are not set unless the communicating replicas concur on the value of b. We therefore save space by using a one-phase bit vector.

5.3.3 The sequence number

To block comparisons between vectors that have undergone different numbers of compression "cycles", we add a sequence number to the version vector. The sequence number counts the number of times compression has completed on any element in the vector. Comparisons can only occur between vectors with equivalent sequence numbers, though different sequence numbers still yield important information. For instance, if vector V_1 has completed compression and V_2 has not, waiting on confirmation from V_1, then V_2 can conclude from the sequence-number differential that V_1 has finished compression and therefore it can also finish. Sequence numbers can never differ by more than one, since all replicas must participate in the compression algorithm. Therefore, communication is never actually restricted between replicas; different sequence numbers only imply that the lesser replica should complete compression before comparing vectors.

5.4 Algorithm details

We present the algorithm in stages. First we describe a key assumption that simplifies algorithm presentation. Then we present the algorithm as operating on a single version vector element. We will then generalize the approach to compress multiple elements at once, and finally describe a mechanism to remove the initial simplifying assumption. While extending the algorithm to allow compression in more structured environments is possible, such as with hierarchical groupings of replicas, the details are not discussed here.

5.4.1 Simplifying assumption

We make the initial assumption that only replica `R` can initiate compression of its element. We refer to the replica that "owns" a particular element as the *gen-*

eration replica, since it is the replica that generates updates for that element. Under this assumption, two replicas will always either agree on the b-value (the consensus value) or else one of the b-values will be zero (that replica doesn't know compression has begun), simplifying algorithm presentation. We later remove this assumption, which leads to situations where $b_1 \neq b_2$ and neither b_1 nor b_2 is zero.

5.4.2 Basic algorithm

The basic compression algorithm operates on a single version vector element. When replica R initiates compression on its element, the element changes form from $\{x\}$ to $\{0, x\}$. Additionally, replica R sets its bit in the one-phase bit vector.

The compression information is now gossiped from replica to replica according to the following rules. Assume that replica R_1 acquires information from R_2 as part of synchronization. The version vectors V_1 from R_1 and V_2 from R_2 are compared, looking at matching elements e_1 and e_2 from each vector respectively. Assume that replica R_2 knows that compression has begun, and therefore e_2 has the form $\{a_2, b_2\}$; R_1 may or may not be knowledgeable, and therefore e_1 either has the form a_1 or $\{a_1, b_1\}$. If R_2 does not know compression has started, R_1 takes no action with regard to the compression algorithm. Under these assumptions, the rules are:

1. If e_1 has the form a_1, R_1 changes e_1 to have the form $\{0, a_1\}$. R_1 then applies the rules below.

2. R_1 can set its own bit in its bit vector whenever it knows that compression has begun (i.e., e_1 has the form $\{a_1, b_1\}$).

3. As part of standard reconciliation, if V_2 dominates V_1 and new data is installed at R_1, then V_1 is set equal to V_2. Specifically, this copies the b-value b_2 when $b_2 > b_1$. R_1 then applies rule 4.

4. If $b_1 = b_2$, then R_1 merges R_2's bit vector with its own using the bit-wise "or" function.

5. When all bits are set in the bit vector, R_1 can complete compression on element e_1 in vector V_1.

These rules guarantee that when any replica has all bits set in the bit vector, all replicas have reached or will reach consensus on the b-value. The completing replica changes the element $e = \{a, b\}$ by subtracting b from the total value for the element $a+b$. If $(a+b)-b = a = 0$, then the entire element is removed; otherwise e simply becomes a.

Since R is the only replica that increments its position in the version vector, no replica S can ever have a larger element for R's position than replica R. Therefore, once a replica S learns that compression has begun on replica R's element, S must have the highest b-value in the system, guaranteed by rules 1-3 above, assuming they are applied atomically. If two version vectors are in conflict and have different b-values, then by rule 4 above the bit vector will not be completely set until the conflict is resolved. Therefore we can be assured that all replicas have achieved consensus on the same b-value.

Each replica, when it completes compression, increments its sequence number on the associated version vector by one. When replica R_1 reconciles from replica R_2, and, for a given version vector, V_2's sequence number is greater than V_1's, R_1 knows that R_2 has completed compression, and therefore R_1 can complete compression as well on V_1.

5.4.3 Compression of multiple elements

The algorithm as described so far works on a single element, requiring each element to have its own set of bit vectors. Since we could potentially have multiple instantiations of the algorithm simultaneously running, each on a different element, it seems to make more sense to merge the multiple instantiations into one.

Multiple instantiations can be easily detected by any replica R when it learns that an additional element in the vector is being compressed. At this point, R restarts the algorithm on the merged set of elements by resetting all bit vectors, setting its own bit according to rule 2, and proceeding. R is allowed to restart because the nature of the two-phase algorithm ensures that no replica completes until it has learned that R has restarted. Therefore no replica prematurely completes and all replicas upon completion agree on the set of elements being compressed.

The foregoing rules for the algorithm remain unchanged, and an additional rule is added:

6. When R_1 pulls data from R_2, and compares V_1 against V_2, if the number of elements being compressed in V_2 is greater than the number of elements being compressed in V_1, then R_1 starts compression on the other elements, clears all bit vectors, and sets its own bit in its bit vector.

Termination is guaranteed only if new replicas are added to the version vector more slowly than the algorithm restarts and completes. A termination guarantee can be enforced by restricting the number of elements that can be simultaneously compressed to some finite number n: the algorithm can restart at most $n-1$ times. There is clearly a penalty for restarting, which depends on when in the execution ("just started" versus "almost done") replicas notice that compression is occurring on multiple elements. The penalty can be minimized by not allowing new elements to start compression when existing elements are "almost done" being compressed. Detecting such times and reducing the penalty is clearly difficult, and while partially based on the number of unset bits in a given bit vector, it must ultimately be based on heuristics.

5.5 Sequence number management

Since all file replicas participate in the algorithm, no replica can ever be more than one compression cycle or algorithm invocation behind any other replica. Therefore, instead of a full sequence number, all that is required is a single bit that distinguishes between "odd" and "even" compression cycles.

5.6 Removing the simplifying assumption

The simplifying assumption in the above algorithm is that only replica `R` can initiate compression on its element. The assumption makes practical sense, given that replica `R` is the best authority concerning when compression should begin on `R`'s element. However, for full generality we may desire an algorithm without such a restriction—an algorithm that allows any replica `R` to initiate compression on any replica `S`'s element. Clearly, the fully general algorithm is more complex. The main difficulty arises in forming consensus on the b-value. Replica `S`, for instance, may store the value n_s for `R`'s element when it decides to initiate compression on `R`'s element, while replica `R` stores the value $n_r > n_s$.

We can relatively easily form consensus on n_s by modifying the rule governing how elements begin compression. However, forming consensus on n_s is only mildly interesting, because if $n_r > n_s$ then the element being compressed is not completely removed, but only reduced in ordinal value.

It is more effective to form consensus on n_r, though also more difficult. To do so, we essentially "re-start" the algorithm when necessary. Whenever replica $\mathtt{R}_1$ reconciles from $\mathtt{R}_2$ and discovers that $b_2 > b_1$, replica $\mathtt{R}_1$ updates its version vector to reflect the latest b-value (and the accompanying data), resets all bit vectors, and sets its own bit according to rule 2. The bit vectors represent a set of sites that store a common b-value; when the b-value changes, we must reset the bit vectors. Since each replica participates in both phases of the algorithm, we are guaranteed that when any replica completes, all replicas will achieve consensus on the same b-value.

5.7 When to initiate compression

Since compression requires communicating with all replicas that maintain a version vector, and compression of multiple elements requires algorithm restarts, there can be a significant penalty when compression is initiated at an "inopportune" moment. Ideally, compression on `R`'s element should not be initiated until `R` has entered a quiescent (cold) state, and its update activity has at least temporarily ceased. Compression should therefore never be attempted for hot replicas. Of course, differentiating between hot and cold replicas is not always easy; nor is it easy to determine when a burst of updates has stopped. Heuristics to determine these events are an ongoing area of future work.

6 Conclusion

We have presented algorithms and a mechanism for dynamic control of a version vector's size, dramatically increasing scalability. We have not, however, described the entire problem, which includes determining when to initiate compression and on which elements. It is our belief that heuristics currently being developed will adequately solve this problem. In this way the scalability of the entire replication system is increased, making optimistic replication more viable for large-scale and mobile systems.

References

[1] R. G. Guy, G. J. Popek, and T. W. Page, Jr. Consistency algorithms for optimistic replication. In *Proceedings of the First International Conference on Network Protocols*. IEEE, Oct. 1993.

[2] P. Honeyman, L. Huston, J. Rees, and D. Bachmann. The Little Work project. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 11–14. IEEE, Apr. 1992.

[3] L. Kleinrock. Nomadicity. Presentation at the GloMo PI Meeting (February 4) at the University of California at Los Angeles, 1997.

[4] G. H. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *USENIX Conference Proceedings*, pages 291–306. USENIX, June 1994.

[5] P. Kumar and M. 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, Oct. 1993. IEEE.

[6] D. S. Parker, Jr., G. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[7] D. Ratner, G. J. Popek, and P. Reiher. The ward model: A scalable replication architecture for mobility. In *Workshop on Object Replication and Mobile Computing*, Oct. 1996.

[8] P. Reiher, J. S. Heidemann, D. Ratner, G. Skinner, and G. J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Conference Proceedings*, pages 183–195. University of California, Los Angeles, USENIX, June 1994.

[9] P. Reiher, J. Popek, M. Gunter, J. Salomone, and D. Ratner. Peer-to-peer reconciliation based replication for mobile computers. In *Proceedings of the ECOOP Workshop on Mobility and Replication*, July 1996.

[10] M. Satyanarayanan. The influence of scale on distributed file system design. *IEEE Transactions on Software Engineering*, SE-18(1):1–8, Jan. 1992.

[11] 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, Apr. 1990.

[12] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed INGRES. *IEEE Transactions on Software Engineering*, 5(3), May 1979.

[13] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 172–183, Copper Mountain Resort, Colorado, Dec. 1995. ACM.

[14] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of the 16th IEEE Computer Society International Conference*. IEEE, Spring 1978.

[15] A. A. Wang, P. L. Reiher, and R. Bagrodia. A simulation framework for evaluating replicated filing environments. Technical Report CSD-970018, University of California, Los Angeles, June 1997.