Defining Weakly Consistent Byzantine Fault-Tolerant Services

Atul Singh[†‡], Pedro Fonseca[†], Petr Kuznetsov[†,§], Rodrigo Rodrigues[†], Petros Maniatis[⋆]
[†]MPI-SWS, [‡]Rice University, [§]TU Berlin/DT Labs, [⋆]Intel Research Berkeley

ABSTRACT

We propose a specification for weak consistency in the context of a replicated service that tolerates Byzantine faults. We define different levels of consistency for the replies that can be obtained from such a service—we use a real world application that can currently only tolerate crash faults to exemplify the need for such consistency guarantees.

Keywords

Byzantine fault tolerance, weak consistency, definitions

1. INTRODUCTION

Byzantine fault tolerance (BFT) enhances the reliability of replicated services. In the Byzantine failure model, no assumptions are made about the behavior of faulty components; this enables a BFT replicated service to withstand not only crash faults but also attacks, software errors, heisenbugs, and non-crash hardware faults, etc.

Existing proposals for the building blocks used in today's data centers behind major Internet services, such as Google's GFS [7], Bigtable [3], or Amazon's Dynamo [4], all replicate data for increased availability and reliability, but assume a benign failure model where nodes fail by crashing or omitting some steps.

While BFT techniques would appear instrumental in improving the resilience of such systems, current proposals for BFT replication algorithms are not aligned with the requirements of these building blocks for modern data centers. This is because existing BFT proposals try to ensure strong consistency (in particular, linearizable semantics [8]), which implies that each operation must contact a large "quorum" of more than $\frac{2}{3}$ (or in some protocols even more) of the replicas to conclude. As a result, the replicated service can become unavailable if more than $\frac{1}{3}$ of the replicas are unreachable due to maintenance events, network partitions, or other non-Byzantine faults, which contradicts the principal design choice for many of the systems running in these data centers: choose availability over consistency to provide continuous service to customers and meet tight SLAs [4].

In this paper we try to address the question of the right consistency model for BFT replication algorithms, in order to become aligned with the availability and performance requirements of systems operating in the backbone of modern data centers. We propose two correctness criteria corresponding to different levels of *eventual consistency*, and we motivate these criteria with examples of services that would require such consistency guarantees. We also discuss some of the bounds that may be required to implement such consistency levels.

Our correctness criteria build on the definition of a *linearizable* Byzantine fault-tolerant (BFT) service based on state machine replication, which is described by Castro [2] using the language of I/O automata [9, chapter 8]. Intuitively, our extension transforms a *legal sequential history* [8] [1, chapter 9], corresponding to a linearizable service, into a *legal history graph* corresponding to a collection of divergent views of the service evolution the correct clients may have. We also show that, by imposing conditions on the geometric properties of the graph, we can reason about the "degree of consistency" exported by the service.

2. PRELIMINARIES

In this section, we briefly describe our system model and recall the definition of a *strongly consistent* (*linearizable*) BFT service [2].

A generic BFT service is characterized by a set of *clients* $\mathcal{C}$, a set of *servers* (or *replicas*) Π, a set of *states* $\mathcal{Q}$, an initial state $q_0 \in \mathcal{Q}$, a set of operations $\mathcal{O}$ that can be applied on the service, a set of responses $\mathcal{O}'$ the service can return, and a transition function $g : \mathcal{C} \times \mathcal{O} \times \mathcal{Q} \rightarrow \mathcal{O}' \times \mathcal{Q}$. We assume that at most $f < N = |\Pi|$ replicas and any number of clients can be Byzantine faulty.

Figures 1 and 2 describe an I/O automaton corresponding to a linearizable BFT service.

Faults of clients and replicas are modeled as input actions $\mathsf{CLIENT\text{-}FAILURE}_c$ and $\mathsf{SERVER\text{-}FAILURE}_i$, respectively. Note that a $\mathsf{SERVER\text{-}FAILURE}_i$ action is only enabled if there are fewer than f faulty servers.

The input action $\mathsf{REQUEST}(o)_c$ accepts a request from client c and adds the request, equipped with the current logical timestamp of c, to the incoming buffer *in*. In case c is faulty, the action $\mathsf{FAULTY\text{-}REQUEST}(o)_c$ may put an arbitrary request to *in*.

Requests in the buffer *in* are processed by the internal action $\mathsf{EXECUTE}(o, t, c)$ that applies operation o to the current state of the service (*val*) using the transition func-

Signature:

Inputs:
REQUEST$(o)_c$
CLIENT-FAILURE$_c$
SERVER-FAILURE$_i$

Internals:
EXECUTE(o, t, c)
FAULTY-REQUEST(o, t, c)

Outputs:
REPLY$(r)_c$

Here $o \in \mathcal{O}$, $c \in \mathcal{C}$, $t \in \mathbb{N}$, $i \in \Pi$, $r \in \mathcal{O}'$

State components:

$val \in \mathcal{Q}$, initially q_0
$in \subseteq \mathcal{O} \times \mathbb{N} \times \mathcal{C}$, initially empty
$out \subseteq \mathcal{O}' \times \mathbb{N} \times \mathcal{C}$, initially empty
$\forall c \in \mathcal{C}$, $last\text{-}req_c, last\text{-}rep_c \in \mathbb{N}$, both initially 0
$\forall c \in \mathcal{C}$, $faulty\text{-}client_c \in Bool$, initially *false*
$\forall i \in \Pi$, $faulty\text{-}server_i \in Bool$, initially *false*
$failed \equiv |\{i | faulty\text{-}server_i = true\}|$

Figure 1: Specification of a linearizable service: signature and state components.

Transitions:

CLIENT-FAILURE$_c$
Eff: $faulty\text{-}client_c := true$

SERVER-FAILURE$_i$
Pre: $|failed| < f$
Eff: $faulty\text{-}server_i := true$

REQUEST$(o)_c$
Eff: $last\text{-}req_c := last\text{-}req_c + 1$
$\quad in := in \cup \{(o, last\text{-}req_c, c)\}$

FAULTY-REQUEST$(o, t, c)_c$
Pre: $faulty\text{-}client_c = true$
Eff: $in := in \cup \{(o, t, c)\}$

REPLY$(r)_c$
Pre: $faulty\text{-}client_c \vee$
$\quad \exists t : (r, t, c) \in out$
Eff: $out := out - \{(r, t, c)\}$

EXECUTE(o, t, c)
Pre: $(o, t, c) \in in$
Eff: $in := in - \{(o, t, c)\}$
$\quad$ if $t > last\text{-}rep_c$ then
$\quad\quad (r, val) := g(c, o, val)$
$\quad\quad out := out \cup \{(r, t, c)\}$
$\quad\quad last\text{-}rep_c := t$

Figure 2: Specification of a linearizable service: transitions.

tion g and adds the corresponding response r to the outgoing buffer *out*. The output action REPLY$(r)_c$ picks up a processed request in *out* and returns the response r to the client.

Note that the service guarantees that all requests are processed in a total order. Indeed, EXECUTE actions update the service state by applying operations in a sequential manner, so from correct clients' perspective the service looks like a single, correct, sequential server.

Liveness in the presence of faults can usually be achieved only if the environment "behaves well." Probably the weakest assumption about the environment one should make to be able to implement a live BFT service is that the system is *eventually synchronous* [5]. In this paper, we chose the following way to describe this synchrony assumption. Let Δ denote a default round-trip delay. A system is eventually synchronous if there is a time after which every two-way message exchange between two clients or servers takes at most Δ time units. Now a live linearizable BFT service ensures that in an eventually synchronous system every request issued by a correct client is eventually provided a response.

3. GENERALIZED WEAK CONSISTENCY IN BFT

In this section we present the correctness criteria for generalized weakly consistent BFT services. Our definitions extend the correctness criteria of *linearizable* BFT services [2] and *eventually consistent* crash fault-tolerant services [6].

3.1 Overview

A weakly consistent BFT service provides clients with two kinds of responses to their requests: *strong* responses corresponding to *strongly complete* (or *committed*) requests, *weak* responses, corresponding to *weakly complete* requests. On one hand, strong responses are based on a total order on requests: a history induced by clients' requests and the corresponding strong responses is linearizable. We further distinguish between two levels of strong responses: "oblivious" strong responses may not reflect the effects of some of the weak operations that completed before but reflects all strong responses, whereas "non-oblivious" strong responses account for all responses (either strong or weak) that were provided earlier.

On the other hand, the guarantee the service provides with respect to weak responses are (1) every weak response is based on *some* order of prior requests, and (2) they will eventually become committed (as soon as the network becomes sufficiently stable). Furthermore the number of such coexisting orders may also be bounded, which can be seen as a measure of consistency the service exports.

We motivate our consistency guarantees by means of an example. Consider a shopping cart application (which is one of Amazon's applications that uses Dynamo [4] as a storage substrate) that exports the following operations: *AddItem*, *RemoveItem*, and *CheckOut*. In this case, we may want the *AddItem* and *RemoveItem* operation to return after obtaining weak responses. This increases the availability and performance of these operations, at the expense that some subsequent operations may not see items that were added or still see items that were already removed (a slight inconvenience that the user is likely to tolerate). Now let's consider what happens when the *CheckOut* operation is run under different consistency guarantees. If it only waits for a weak response, then it may subsequently be assigned a different position in the final order of committed operations, which may not be desirable (e.g., the weak response to *CheckOut* does not see an *AddItem* operation that eventually is serialized before the checkout). If *CheckOut* waits for an "oblivious" strong response, this situation may not occur because the position in the serial order is stable; however, some items that were previously added may not appear in the check out (or a removed item may still appear in the cart). This is probably acceptable provided the customer is informed of which items were checked out (though the customer may subsequently see these items appear in subsequent sessions). Finally, if the check out waits for a "non-oblivious" strong response, it is guaran-

```
Signature:

Inputs:
$\mathsf{REQUEST}(o)_c$
$\mathsf{CLIENT\text{-}FAILURE}_c$
$\mathsf{SERVER\text{-}FAILURE}_i$

Internals:
$\mathsf{EXECUTE}(o,t,c)$
$\mathsf{ENTER}(o,t,c)$
$\mathsf{FAULTY\text{-}REQUEST}(o,t,c)$
$\mathsf{FORK}(o,t,c)$
MERGE

Outputs:
$\mathsf{WEAK\text{-}REPLY}(r)_c$
$\mathsf{STRONG\text{-}REPLY}(r)_c$

$(o \in \mathcal{O},\ t \in \mathbb{N},\ i \in \Pi,\ c \in \mathcal{C},\ r \in O')$

State components:

$vals$, multiset on $(\mathcal{O} \times \mathbb{N} \times \mathcal{C})^*$, initially $\{\bot\}$
$in \subseteq \mathcal{O} \times \mathbb{N} \times \mathcal{C}$, initially empty
$out \subseteq \mathcal{O}' \times \mathbb{N} \times \mathcal{C}$, initially empty
$\forall c \in \mathcal{C}$, $last\text{-}req_c \in \mathbb{N}$, initially 0
$\forall c \in \mathcal{C}$, $faulty\text{-}client_c \in Bool$, initially $false$
$\forall i \in \Pi$, $faulty\text{-}server_i \in Bool$, initially $false$
$committed$, map from $vals$ to $(\mathcal{O} \times \mathbb{N} \times \mathcal{C})^*$

$failed \equiv |\{i | faulty\text{-}server_i = true\}|$

Transitions:

$\mathsf{REQUEST}(o)_c$
Eff: $last\text{-}req_c := last\text{-}req_c + 1$
     $in := in \cup \{(o, last\text{-}req_c, c)\}$

$\mathsf{CLIENT\text{-}FAILURE}_c$
Eff: $faulty\text{-}client_c := true$

$\mathsf{ENTER}(o,t,c)$
Pre: $(o,t,c) \in in\ \wedge\ \exists v \in vals : (o,t,c) \notin v$
          $\wedge\ |v - committed(v)| < L_{max}$
Eff: $v :=$ select $v \in vals : (o,t,c) \notin v$
          $\wedge\ |v - committed(v)| < L_{max}$
     add $(o,t,c)$ to the end of $v$

$\mathsf{SERVER\text{-}FAILURE}_i$
Pre: $|failed| < f$
Eff: $faulty\text{-}server_i := true$

$\mathsf{FAULTY\text{-}REQUEST}(o,t,c)_c$
Pre: $faulty\text{-}client_c = true$
Eff: $in := in \cup \{(o,t,c)\}$

$\mathsf{WEAK\text{-}REPLY}(r)_c$
Pre: $faulty\text{-}client_c = true \vee \exists t : (r,t,c) \in out$
Eff: $out := out - \{(r,t,c)\}$

$\mathsf{STRONG\text{-}REPLY}(r)_c$
Pre: $faulty\text{-}client_c = true \vee \exists t : (r,t,c) \in out\text{-}commit$
Eff: $out\text{-}commit := out\text{-}commit - \{(r,t,c)\}$

MERGE
Pre: $|vals| \geq 2$
Eff: select $\{v, v'\} \subseteq vals$
     $v'' :=$ merge $v$ and $v'$
     (application-specific conflict resolution)
     (remove duplicates)
     $committed(v'') := \max(committed(v), committed(v'))$
     $vals := vals - \{v, v'\} + \{v''\}$

FORK
Pre: $|vals| < D_{max}$
Eff: select $v \in vals$
     $vals := vals + \{v\}$

COMMIT
Eff: select $v \in vals$ with the longest committed prefix
     $committed(v) := v$

$\mathsf{EXECUTE}(o,t,c)$
Pre: $\exists v \in vals$: $(o,t,c) \in v$
Eff: select $v \in vals$: $(o,t,c) \in v$
     $r :=$ response of $(o,t,c)$ in $v$
     if $(o,t,c) \in committed(v)$ then
          $out\text{-}commit := out\text{-}commit \cup \{(r,t,c)\}$
     else
          $out := out \cup \{(r,t,c)\}$
```

Figure 3: Specification of a weakly consistent service.

teed to see all of the operations that concluded previously (both strong and weak).

In the following, we first describe a weakly consistent service that provides weak and oblivious strong responses, and then show how the service can be extended to cover the case of non-oblivious commitments.

3.2 State and transitions

Figure 3 describes a generic weakly consistent service. Below we pinpoint what makes the weakly consistent service different from the linearizable service described in the previous section.

The global state of the weakly consistent service, denoted *vals*, is modeled now as a multiset of *histories*: sequences of elements of the form (o,t,c) where $o \in \mathcal{O}$, $t \in \mathbb{N}$, and $c \in \mathcal{C}$. Each history v in *vals* is characterized by a set of client requests, an order in which the requests are applied, and a prefix of *committed* operations in v, denoted *committed*(v), i.e., requests whose position in the order is fixed in the current execution. The committed prefixes of histories in *vals* monotonically grow and are related by containment. Additionally, a request appears exactly once in *committed*(v).

The service maintains two parameters, D_{max} and L_{max}. D_{max} bounds the number of concurrent histories that can be maintained by the service. L_{max} bounds the number of not yet committed requests in a concurrent history.

A request o generated by a correct client c and the corresponding responses is modeled as an input action $\mathsf{REQUEST}(o)_c$ that computes the timestamp of the current request of c and adds an element (o,t,c) to the input buffer *in*. Internal action $\mathsf{ENTER}(o,t,c)$ adds a request $(o,t,c) \in in$ to the end of one or more histories in *vals*, if the number of not yet committed requests in each of these selected histories is less than L_{max} and the request is not already there. $\mathsf{EXECUTE}(o,t,c)$ chooses a history v in *vals* that contains (o,t,c), computes the response r that the request (o,t,c) returns after applying the se-

```
Signature:
...
Internals:
...
COMMIT-ALL
...

Transitions:
...
COMMIT-ALL
Eff: $v$ := merge all histories in $vals$ preserving committed order
        (application-specific conflict resolution)
     $vals := \{v\}$
     $committed(v) := v$
...
```

Figure 4: Specification of a non-oblivious weakly consistent service.

quence of operations prescribed by v to the initial state and adds the entry (r, t, c) to *out-commit* or *out*, depending on whether (o, t, c) is in *committed*(v). The corresponding (weak or strong) response r is modeled by an output action WEAK-REPLY$(r)_c$ that is enabled if *out* contains an element (r, t, c), or STRONG-REPLY$(r)_c$ that is enabled if *out-commit* contains an element (r, t, c). If a WEAK-REPLY$(r)_c$ (respectively, STRONG-REPLY$(r)_c$) action is triggered in response to request (o, t, c), we say that the request is *weakly complete* (respectively, *strongly complete*). Note that, since all committed prefixes are related by containment, the strongly complete requests are totally ordered.

Our specification allows an operation (o, t, c) to appear in multiple histories and hence EXECUTE can be called multiple times for the same operation. Hence, *out* can contain multiple responses for the same request. This is to model situations, such as a flaky network, where an *AddItem* operation appears in both sides of a partition and is processed independently. However, *out-commit* contains only one response per request since each request gets a unique position in *committed(v)*.

A history in *vals* may *fork*, i.e., decompose in a number of identical "clones" (internal action FORK). An internal action MERGE produces a single history v from a set of histories V in *vals*, adopting the longest committed prefix in V, removing duplicates, and ordering the rest of the requests that appear in histories in V (here the service may use application-specific conflict resolution policies [11]). At any time, one of the histories with the longest committed prefix can commit all its requests (action COMMIT).

3.3 Non-oblivious commitment

In the previous definition, even though committed requests are totally ordered, it is still possible that a strong operation misses some weak operations that had concluded earlier. Figure 4 describes a modification to the automaton in Figure 3 that ensure that committed requests are non-oblivious: a committed request does not miss any preceding complete request (weak or strong). Essentially, we introduce a new COMMIT-ALL action that merges all concurrent histories in *vals* implying that every subsequent complete request will be based on the extension of the committed history.

In a more general way, oblivious and non-oblivious operations can be combined. As a result, a client can specify which requests should be committed in a non-oblivious way, and which requests, once complete, should not be missed by a subsequent non-obliviously committed operation.

Note that the evolution of the service state can now be represented in the form of a directed acyclic graph G. Vertices of the graph are histories and there is a directed edge if (1) $v' = v \cdot (o, t, c)$ where an ENTER(o, t, c) action extended $v \in vals$ with the request (o, t, c), or (2) v' is a result of merging a set of histories V such that $v \in V$. A COMMIT-ALL action produces a vertex that is either predecessor or successor for any other vertex in the graph. Now the parameter D_{max} bounds the number of pairwise concurrent vertexes in G, i.e., vertexes that are not connected by a directed path.

3.4 Defining Liveness

One way to define the liveness properties is to require that a weakly consistent service guarantees progress for correct clients that can communicate with *enough* replicas in a timely manner. More precisely, we say that a tuple $(\mathcal{C}', \Pi')$, where $\mathcal{C}' \subseteq \mathcal{C}$ and $\Pi' \subseteq \Pi$, is an *eventually synchronous partition* if there is a time after which every two-way message exchange among correct agents in $\mathcal{C}' \cup \Pi'$ takes at most Δ time units. We distinguish between *strong* partitions in which Π' contains a *strong quorum* of Q_S correct replicas and *weak* partitions in which Π' contains a *weak quorum* of Q_W correct replicas, respectively. The parameters Q_S and Q_W affect the level of consistency of the service and will be specified later.

Assuming that every two correct replicas eventually reliably communicate, we put our liveness requirements as follows: (1) If there exists an eventually synchronous weak partition $(\mathcal{C}', \Pi')$, then every request issued by a correct client $c \in \mathcal{C}'$ eventually triggers a (weak or strong) reply. (2) If there exists an eventually synchronous strong partition $(\mathcal{C}', \Pi')$, then every weakly complete request is eventually committed.

Note that the properties imply that if a correct client c belongs to an eventually synchronous strong partition, then each request from c will eventually be committed.

3.5 Implementing Weakly Consistent BFT

We have designed and implemented a weakly consistent BFT protocol, called Zeno [10], that meets both the safety specification (with oblivious commitment) and liveness requirements described earlier. Zeno is live and safe for $f < N/3$, $Q_W = f + 1$ and $Q_S = \lceil \frac{N+f+1}{2} \rceil$ At a high level, the system ensures that a client makes progress as soon as the client receives at least $f + 1$ *matching* replies to its request, i.e., $f + 1$ replies based on the same history. This implies that the request is produced by some correct replica based on its history. To commit requests, Zeno requires, like traditional BFT protocols, quorums of size $2f + 1$.

When a correct replica learns about a conflicting history, it initiates a merge operation that combines the requests of the concurrent histories. In case the service partitions, a single history may fork into a number of concurrent histories. If a merge operation involves a strong quorum of replicas, then all involved requests are committed.

4. OPEN QUESTIONS

In our framework, the weakest form of consistency does not bound the number of concurrently existing histories (D_{max}) and the number of requests a given history may process in a speculative manner (L_{max}). We anticipate that, in many practical scenarios, these parameters can in fact be bounded.

First, we expect that in an execution with f' faulty servers ($f' \leq f$), a weakly consistent service can be implemented with $D_{max} \leq \lfloor \frac{n-f'}{f-f'+1} \rfloor$. The intuition here is that $n - f'$ correct servers can be split into at most $\lfloor \frac{n-f'}{f-f'+1} \rfloor$ groups of size $f + 1$, and thus at most that many histories can coexist in that execution. In fact, Zeno can be shown to match this bound.

On the other hand, we expect that L_{max} can only be bounded by strengthening synchrony assumptions of the system (or by weakening the liveness requirements, by saying the system may have to halt at some point until an eventually synchronous strong partition is formed). Indeed, L_{max} is proportional to the length of the longest period of partition, i.e., the period of time during which a number of divergent concurrent histories are allowed to coexist in the system. If this time can be bounded (e.g., by periodic human intervention), L_{max} can be bounded too.

It can be shown that with $N = 3f + 1$ replicas, $Q_W = f + 1$, and $Q_S = 2f + 1$, it is not possible to implement a safe and live weakly consistent *non-oblivious* service. This is because a Q_W and a Q_S intersect in only one replica and it could be faulty and not report the speculative operations it participated in during merges. One potential approach is to increase the size of Q_W to $2f + 1$. However, this does not provide tangible benefits compared to traditional BFT protocols since quorum sizes are now identical. But if $N = 5f + 1$, the non-obliviousness property can be achieved with $Q_W = 2f+1$ and $Q_S = 4f + 1$. On the other hand, traditional BFT protocols with $5f + 1$ replicas are not available as soon as $\geq f$ replicas are unreachable.

Proving the aforementioned conjectures and considering related questions opens an avenue for future research that both combines interesting theoretical challenges and addresses actual practical needs in fault-tolerant distributed computing.

5. REFERENCES

[1] H. Attiya and J. L. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* Wiley, 2004.

[2] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, MIT, Jan. 2001. Technical Report MIT/LCS/TR-817.

[3] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, Seattle, WA, USA, Dec. 2006.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, Oct. 2007.

[5] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288 – 323, April 1988.

[6] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1):113–156, 1999.

[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, 2003.

[8] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, June 1990.

[9] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[10] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually Consistent Byzantine Fault Tolerance. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, Apr. 2009.

[11] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, Colorado, USA, Dec. 1995.