Discarding Obsolete Information in a Replicated Database System

SUNIL K. SARIN, MEMBER, IEEE, AND NANCY A. LYNCH

Abstract—**A replicated database architecture is described in which updates processed at a site must be saved to allow reconcilliation of newly arriving updates in a way that preserves mutual consistency. The storage space occupied by the saved updates increases indefinitely, and periodic discarding of old updates is needed to avoid running out of storage. A protocol is described which allows sites in the system to agree that updates older than a given timestamp are no longer needed and can be discarded. This protocol uses a "distributed snapshot" algorithm of Chandy and Lamport and represents a practical application of that algorithm. A protocol for permanent removal of sites is also described, which will allow the discarding of updates to continue when one or more sites crash and are expected not to recover.**

Index Terms—**Distributed databases, distributed snapshots, mutual consistency, network partitions, replicated data, timestamps.**

I. Introduction

A RECOGNIZED approach to providing high availability of replicated data is to allow sites to issue updates at any time and to reconcile their database copies after the fact when they learn of each others' updates. This approach was proposed by Johnson and Thomas [6] and was used successfully in Grapevine [2]. Mutual consistency of sites' database copies is ensured by defining a total order on all updates issued, typically using *timestamps* [7]. A site's database copy at any given time must reflect those updates that it has received so far, as if they had been executed in timestamp order.

Eventual mutual consistency in systems such as the above requires that all sites eventually see all updates; copies of updates may therefore need to be retained for retransmission to failed or disconnected sites. Furthermore, because updates may not arrive at a site in timestamp order, it is necessary for sites to retain information (including timestamps of deleted items) that will allow late-arriving updates with old timestamps to be correctly processed. This information grows over time, and may eventually exhaust available storage at a site if it is not discarded. Johnson and Thomas use a two-phase protocol in which sites first report to each other which deleted items they have seen, and then discard deleted items which every site is known to have seen. (This information is compressed into a single timestamp for efficiency.) Grapevine, on the other hand, retains deleted items for two weeks, to protect against communication delays and to preserve the record for administrative purposes. This arbitrary expiration period may sometimes be overly conservative (holding on to old information for longer than necessary) or may sometimes be overly optimistic (discarding information that may later be needed to recover from an error or from a prolonged network partition).

This paper describes a more systematic method for discarding old information in systems such as the above. Our protocol takes into account both administrative needs for historical information (as does Grapevine, but without setting an arbitrary fixed expiration period) and the status of communication among sites (as does the Johnson and Thomas protocol). We assume that sites can issue updates with old timestamps (older than their clock readings), representing real-world events that have not yet been reported or corrections of past errors. Discarding of old information requires agreement among site administrators not to issue any further updates with timestamps older than some "cutoff" value. To achieve this agreement and establish a cutoff timestamp, our protocol uses the "distributed snapshot" algorithm of Chandy and Lamport [3] to record a consistent global state of the system and avoid anomalies. The protocol for discarding old information can be run in "background," and does not hold up the issuing and processing of new updates.

The protocol we describe was developed for CCA's System for Highly Available Replicated Data (SHARD) [8]. (A prototype of SHARD has been implemented, but does not yet include this protocol.) The updates issued by applications in SHARD may be complex computations, including abstract operations such as incrementing or decrementing a numeric quantity, and conditional updates whose effects depend on the contents of the database. The latter can be used to detect nonserializable transaction execution; for example, the data that a user saw when he issued an update can be tested for changes when the update is later executed. This allows the application to perform additional database updates and external actions that may be needed to compensate for inconsistencies. This

Manuscript received January 31, 1986; revised June 16, 1986. This work was supported by the Defense Advanced Research Projects Agency of the Department of Defense and by the Air Force Systems Command at Rome Air Development Center under Contract F30602-84-C-0112. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, RADC, or the U.S. Government.

S. K. Sarin is with Computer Corporation of America, Cambridge, MA 02142.

N. A. Lynch is with the M.I.T. Laboratory for Computer Science, Cambridge, MA 02139.

IEEE Log Number 8611360.

capability is not supported in previous systems of this kind, which assume that an update can only replace a data item with a specified new value, or add or delete an element of a set.

In order to maintain mutual consistency in the presence of complex update types, a site in SHARD must retain all updates that it has seen. When a new update is received that conflicts with a previously executed update that has higher timestamp, the latter must be rolled back and the updates executed in timestamp order [9]. Because it may need to be reexecuted, an update with timestamp T must be retained until the site is certain that it will not subsequently receive any updates with timestamps smaller than T; this is established using the protocol we describe. In systems that support only the simpler update types described above, updates are never reexecuted; it is only necessary to remember one timestamp per data item changed or added or deleted, rather than every update. The protocol we describe can be used in such systems as well, to determine when these timestamps (and the records of deleted items) can be forgotten.

Since discarding of old information requires receiving assurances from every site, it can be held up indefinitely by a site failure. We describe a protocol by which site administrators can remove one or more sites that they have declared to be "dead" for all intents and purposes; discarding of updates can then proceed at the remaining sites. The site removal protocol requires careful synchronization to ensure that all remaining sites receive the same set of updates from the sites being removed.

Section II of this paper presents an abstract model of the operation of our system, which will be used in describing the algorithms. Section III describes how updates can be determined to be obsolete and thereby discarded. Section IV describes the site removal protocol. Section V concludes the paper by summarizing the key features of the protocols.

II. System Model

The system consists of a fixed collection of *sites*. Each site x carries the following state information:

Copy$\{x\}$:	The site's database copy, whose value is some element of the set of possible database states.
History$\{x\}$:	The set of updates (described below) seen by the site so far.
Num-Issued$\{x\}$:	A counter indicating how many updates this site has issued.

A database *update* has the following components:

1) A *site-id*, identifying the site that issued the update.

2) A *sequence number*, which is the value of the site's Num-Issued just after the update was issued.

3) A *timestamp*, used to totally order this update with respect to all other updates. Timestamps are globally unique in that no two updates have the same timestamp, even if issued by different sites. Updates issued by a site do not necessarily have increasing timestamps.

4) A *mapping* from database states to database states, which describes the intended effect of executing the update on the database.

This model does not specify the nature of the possible database states; it applies equally to traditional database applications with records of entities and attributes as to directory systems where values of various types are stored under arbitrary string names. The types of mappings allowed in an update, and their encoding for transmission and execution, are also not specified because the protocols in this paper apply to any kind of update so long as it is deterministic.

In addition to sites' local states, the state of the system includes a **Channel**$\{x, s\}$ for every site x and every site s, which is a first-in-first-out (FIFO) queue of updates issued by s but not yet received by x. The system starts up with every site's Copy in the same initial state C_0, with every History and every Channel empty, and every site's Num-Issued equal to zero. The operation of the system can be characterized as a sequence of the following kinds of atomic events:

1) Site x spontaneously issues an update. The update is assigned an appropriate timestamp, by any mechanism that guarantees global uniqueness. The counter Num-Issued$\{x\}$ is incremented, and the resulting value is the sequence number of the update. The update is enqueued at the end of Channel$\{s, x\}$ for every site s. This includes Channel$\{x, x\}$; we distinguish the issuing of an update by a site from the receipt and execution of that update by the same site.

2) Site x receives the update at the front of Channel$\{x, s\}$ for some s: The update is removed from Channel$\{x, s\}$ and added to History$\{x\}$. Copy$\{x\}$ is updated to reflect the database state that would have resulted if all the updates in History$\{x\}$ were executed in increasing timestamp order starting with the initial state C_0. In order to achieve this starting with the current value of Copy$\{x\}$, some undoing and reexecution of higher timestamped conflicting updates (that are already in History$\{x\}$) may be needed.

The above model is an abstraction of the SHARD prototype, that hides the details of the implementation and of how updates are issued (in response to a user command, or a stimulus from an external device, or a trigger condition on the database). In SHARD, site failures are masked by preserving state information on "stable storage" so that if a site does crash it can resume processing from the same state on recovery. A failed site is assumed not to corrupt its state information or issue bad messages, i.e., Byzantine failures are not handled. The logical Channels of the model are implemented, on whatever physical communication network is available, using an efficient reliable broadcast protocol [1], [4]. In this strategy, the implementation of each Channel is distributed among all the sites. When an update is issued by some site s, delivery of the update to site x does not require that s and x ever be in direct physical communication; any site that did receive the update (directly or indirectly) from s

can forward it to any site that did not receive it. Because updates are permanently recorded in sites' Histories and are transmitted whenever communication is possible, the logical Channels are also "stable." That is, site failures and network partitions are masked; their only observable effect is a very long communication delay during which newer updates enqueued on the affected Channels cannot be received and processed.

III. Discarding Old Updates

In the system model we described above, updates are only added to a site's update history, never removed. This section addresses the problem of discarding updates from the history in order to reclaim storage space.

A. The Problem

If a site has received and processed an update with timestamp T, it can discard the update when it knows that all of the following conditions are true:

P1(T): Every site in the system has received the given update.

P2(T): There are no updates in transit to this site with timestamp smaller than T.

P3(T): No site in the system will issue any further updates with timestamp smaller than T.

Condition P1 implies that the update is no longer needed by the reliable broadcast algorithm for retransmission to sites that have not received it. Conditions P2 and P3 together imply that no further updates with timestamp smaller than T will be received by this site, which means that the given update (with timestamp T) will never have to be undone and reexecuted by the mutual consistency algorithm. A record of the update is therefore no longer needed.

We can extend the above to obtain the following conditions on when a site can discard all updates with timestamp up to (but not including) a given timestamp T:

Q1(T): All sites in the system have received all updates that were ever issued with timestamp smaller than T.

Q2(T): No site in the system will issue any further updates with timestamp smaller than T.

That the above conditions do allow *all* updates with timestamp smaller than T to be discarded can be seen from the following:

$$Q1(T) \text{ implies } P1(T') \text{ and } P2(T') \text{ for all } T' < T$$
$$Q2(T) \text{ implies } P3(T') \text{ for all } T' < T$$

Any timestamp T satisfying Q1 and Q2 will be called a **Global-Cutoff** timestamp. If T is a Global-Cutoff timestamp, so is any timestamp smaller than T. This means that, while sites should strive to determine the "maximal" Global-Cutoff timestamp possible, it is sufficient for them to determine some such timestamp in order to discard older updates. Different sites do not have to determine a Global-Cutoff at exactly the same time, nor do they have to determine exactly the same value.

B. Issuing Updates with Old Timestamps

In the design of SHARD, we have chosen not to assume that sites issue updates in increasing timestamp order. In particular, an update that reports a real-world event is assumed to carry as its timestamp the time of occurrence of the external event, not the time that the event is reported to a site in the system. (Errors in reporting the external time can be compensated for by issuing additional updates with old timestamps.) Since such reports may be subject to arbitrary delays outside the system (e.g., the event is recorded on paper and data entry is performed at a later time), there is no guarantee that any site will issue them in increasing timestamp order. The ability to submit updates with old timestamp is also needed to correct past errors, since the system does not allow an update, once issued, to be retracted. The timestamp of an update is thus selected by the agent submitting the update (be it a human user or an automatic sensing device), not by the site to which the update is submitted. The site may, however, append additional bits to the selected time to ensure uniqueness of all timestamps ever issued.

If arbitrarily old updates can arrive at any time, then existing updates from the history can never be discarded. In reality, the relevance of old data decreases over time, and most applications choose not to deal with very old updates even if they are not completely certain that no such updates are missing. In a centralized database system, we would expect an *administrator* to specify a cutoff timestamp such that he is certain (to a very high probability) that no further updates with timestamp smaller than the cutoff will be received or will need to be issued to correct past errors; updates older than the cutoff would then be discarded. If in fact some information with timestamp smaller than the cutoff subsequently needs to be entered or corrected, this will no longer be possible. In this unlikely situation, the administrator would probably have to intervene and decide to either ignore the old update or approximate its effect as best as possible using updates that have timestamp greater than the cutoff.

In a distributed system such as SHARD, there may be multiple administrators each of whom has only limited knowledge of the external world and of what the other sites are doing. No site administrator can on his own specify a cutoff timestamp and discard older updates because some other site may issue an update with older timestamp. Instead, some kind of agreement among the site administrators is needed in order to properly determine a timestamp that satisfies our condition for being a Global-Cutoff. This is done using the protocol described below.

We assume that every site has an administrator who is responsible for monitoring the database at that site, checking for inconsistency, investigating errors and correcting for erroneous or missing reports, and so on. The administrator role at a site may be filled by one person or by many, or may be partially or wholly automated; dif-

ferent sites may exhibit different degrees of automation. Each site's administrator periodically determines a **Local-Cutoff** timestamp such that, as far as this administrator can ascertain, there are no updates missing or errors to be corrected with timestamp earlier than the Local-Cutoff. When the administrator specifies a Local-Cutoff timestamp, he is making the following guarantees:

LC1: So long as the site does not subsequently receive any update with timestamp smaller than Local-Cutoff, it will not issue any update with timestamp smaller than Local-Cutoff.

LC2: If the site does receive an update with timestamp smaller than Local-Cutoff, it may reset Local-Cutoff to a smaller value, subject to the conditions described below. The guarantees (LC1 and LC2) will now apply for the new, smaller, value of Local-Cutoff.

When a site receives an update with timestamp smaller than the current value of its Local-Cutoff, it applies some *rule* to determine the oldest possible timestamp that might be needed to correct errors introduced by the receipt of this update. This timestamp is used as the site's new Local-Cutoff. The most conservative rule a site may use is to reset Local-Cutoff to the timestamp of the incoming update, implying that the site's administrator, in the light of the new information received about the past, may change his mind about the absence of errors and need to issue error-correcting updates as far back as the timestamp of the new update. At the opposite extreme, the rule used by the site may leave Local-Cutoff unchanged, reflecting an unconditional commitment by the site's administrator not to issue any updates with timestamps earlier than Local-Cutoff, even if such updates are subsequently received. Arbitrary rules between the above extremes are permitted, so long as Local-Cutoff is not reset to be smaller than the timestamp of the incoming update and is not moved forward by the rule. The rule may take into account the nature of the new update, the database contents, and other related updates already received.

The rule used in resetting Local-Cutoff at a site can be tailored to the application's needs. It is also possible to have different sites follow different rules. For example, the system designers may tailor the protocol to be dependent on a small set of human administrators (rather than on one at every site) by making all but a few key sites advance Local-Cutoff along with their local clocks and never set it backward; these sites will always issue updates in increasing timestamp order. How the rules should be chosen for different sites is a problem for future research.

If a site's Local-Cutoff is reset by the receipt of an old update, the site's administrator may need to issue updates with old timestamps to correct newly introduced errors. Whether or not he actually issues such updates, the administrator will advance Local-Cutoff to a higher value when he is once again satisfied about the absence of errors. Note that only the administrator can advance Local-Cutoff, while only the receipt of an earlier timestamped update can move Local-Cutoff backwards.

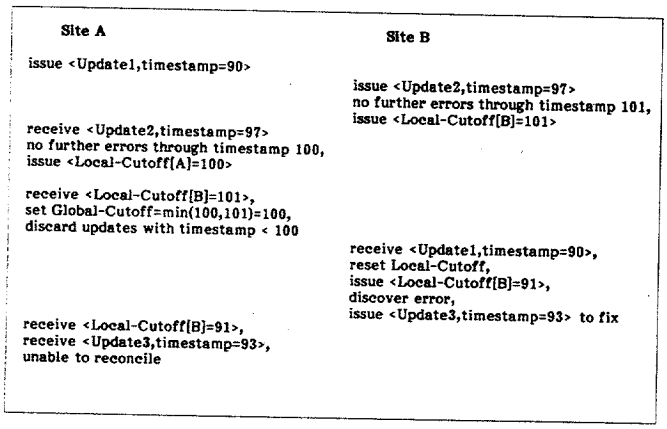

Fig. 1.

If every site's administrator specifies a Local-Cutoff, it might appear that no further updates will ever be issued with timestamp smaller than the smallest of the Local-Cutoffs. This does not take into account updates in transit that have not yet been received. If we use a naive approach in which sites simply inform each other of their Local-Cutoffs and take the minimum to be the value of Global-Cutoff, errors may occur, such as illustrated in Fig. 1. When A both issues a $<$Local-Cutoff[A] $= 100>$ and receives a $<$Local-Cutoff[B] $= 101>$, it mistakenly believes that no new updates with timestamps earlier than 100 will ever be issued; A therefore discards all earlier updates. The reason this is a mistake is that the Local-Cutoff from B is obsolete, in that B had not yet seen Update1, with timestamp 90, when it issued $<$Local-Cutoff[B] $= 101>$. On receiving this update, B changes its mind about the absence of errors (which it is allowed to do) and issues Update3 with timestamp 93. Previously issued updates (such as Update2, with timestamp 97) that had timestamps between 93 and 100 will have been discarded by A. Therefore, A can no longer reconcile Update3 and establish a database copy consistent with B's. Furthermore, A may no longer be able to reacquire the discarded updates from B: If B's administrator, after issuing Update3, decides again that there are no further errors through timestamp 101, then on receiving $<$Local-Cutoff[A] $= 100>$ B will also set Global-Cutoff to 100 and discard earlier timestamped updates.

C. Distributed Snapshot Solution

To avoid the problem above, it is necessary to take into account updates in transit (sent but not yet received) when computing Global-Cutoff. If a "snapshot" of the global system state could be taken at the time when A and B set their Local-Cutoffs to 100 and 101, respectively, it would include the undelivered Update1 with timestamp 90. If we take the smallest of the Local-Cutoffs *and* the timestamps of updates in transit, the value of Global-Cutoff would be only 90. The sites would therefore not prematurely discard updates with timestamps smaller than 100, and A

would be prepared to deal with Update3, with timestamp 93, subsequently issued by B.

To formalize the above, we define **Smallest(s)** to be the smallest of the following timestamps in global state s:

- All sites' Local-Cutoff timestamps.
- The timestamps of all updates in transit.

We can prove that no update issued in the future can have timestamp smaller than Smallest(s) by establishing that Smallest(s) cannot decrease. This is done by induction on global system states. Consider the next event that occurs in state s. This event can only be one of the following kinds:

1) Some site receives an update. If this update has timestamp greater than or equal to the site's Local-Cutoff, the site's Local-Cutoff will not change and therefore Smallest(s) will not change. If the timestamp of the update is smaller than Local-Cutoff, the site's Local-Cutoff may be reset but will be no smaller than the timestamp of the update. Since the definition of Smallest(s) includes updates in transit, the timestamp of the incoming update cannot be smaller than Smallest (s). Therefore, the site's new Local-Cutoff cannot be smaller than Smallest(s), and Smallest(s) remains unchanged.

2) Some site issues an update. The timestamp of this update cannot be smaller than the site's Local-Cutoff, and therefore cannot be smaller than Smallest(s). Since Local-Cutoff does not change on issuing an update, Smallest(s) remains unchanged.

3) Some site's administrator specifies a new Local-Cutoff for the site. Since the administrator cannot make the site's Local-Cutoff smaller, the new Local-Cutoff cannot be smaller than Smallest(s). That is, Smallest(s) can only increase or remain unchanged; it cannot become smaller.

Thus, the value of Smallest(s) cannot decrease, and no future update can have a smaller timestamp (condition Q2). By definition, no update in transit in state s has a timestamp smaller than Smallest(s); all updates issued that did have smaller timestamp have already been received by all sites (condition Q1). Therefore, Smallest(s) satisfies our conditions for being a Global-Cutoff timestamp. (In fact, for a given state s, Smallest(s) is the highest timestamp that is a valid Global-Cutoff.) The problem, in a distributed system without a global clock, is that it is not possible to take an instantaneous snapshot of all sites' local states (Local-Cutoff timestamps in this case) and messages in transit. However, Chandy and Lamport have developed a distributed snapshot algorithm [3] in which the recording of pieces of the snapshot at the various sites is synchronized in such a way as to obtain a global state that could have occurred at some instant of time. More precisely, if s_r is the state recorded by the snapshot, and s_i and s_f are the system states at the beginning and end of the snapshot, respectively, then s_r is reachable from s_i and s_f is reachable from s_r.

Because of the reachability guarantee, and because Smallest(s) is nondecreasing, the following must be true:

$$\text{Smallest}(s_i) \leq \text{Smallest}(r_r) \leq \text{Smallest}(s_f)$$

Even though the recorded state s_r may not actually have occurred during the execution of the system, we see that Smallest(s_r) is an acceptable if not maximal Global-Cutoff. If a site computes the value of Smallest(s_r) from a completed snapshot, it can safely use this value as Global-Cutoff and discard earlier timestamped updates.

The Chandy–Lamport snapshot algorithm works as follows. An arbitrary directed graph of FIFO communication channels connects the sites (or processes) in the system. The goal of the algorithm is to have every site take a snapshot of its local state and of the unreceived messages on its incoming communication channels, such that the collection of local snapshots represents a meaningful global snapshot. The algorithm uses special *marker* messages to synchronize the local snapshots, according to the following rules:

1) When a site records its local state, it issues a marker on every outgoing channel. (In our system model, this includes the channel carrying messages from the site to itself.)

2) For each incoming channel, a site remembers in a buffer all messages received between its recording of the local state and the receipt of a marker on that channel.

3) A site must record its local state no later than the receipt of the first marker on an incoming channel. If a marker is received and the site has not yet recorded its local state, the marker is not processed (in effect not "received") until the state is recorded. For the channel on which this first marker was received, the buffer of messages (between the saving of the local state and the receipt of the marker) will be empty.

The snapshot algorithm is superimposed on the underlying distributed computation and does not affect it in any way. The messages remembered in the channel buffers are not held up by the snapshot; the site continues to process them normally as they arrive.

A distributed snapshot is initiated by any site saving its local state and issuing markers. A snapshot may also be initiated concurrently by more than one site. Once a site has received a marker on every incoming channel, it has constructed its piece of the global snapshot, namely the local state that it saved and the messages subsequently received on all incoming channels; these messages are the ones that were in transit in the state recorded by the snapshot. If the sites then wish to use the snapshot to perform some computation or test some property, e.g., determine a Global-Cutoff timestamp, relevant information from the snapshots can be collected in a variety of ways. Sites are fully connected in our system, and broadcast of a message along all outgoing channels is particularly efficient. Therefore, we choose to have every site broadcast its piece of the snapshot to all sites. The computation of Global-Cutoff is then replicated at all sites.

Assuming one or more site administrators initiate a snapshot, the computation of Global-Cutoff at a site proceeds as follows. Each site uses a register **Saved-Local-Cutoff** to record the value of Local-Cutoff at the beginning of the snapshot, when the site issues its markers. (Saving this value is necessary because the site adminis-

trator may advance Local-Cutoff while the snapshot is taking place, in preparation for a future snapshot.) Once the site's local snapshot is complete (i.e., it has received markers from all other sites), it examines the timestamps on the updates saved in the channel buffers and computes the minimum of Saved-Local-Cutoff and the timestamps of the buffered updates. The site broadcasts this value, and then waits to receive such a value from every site. When all of these are received, Global-Cutoff is set to the smallest of the values reported (this is the value of Smallest(s_r) for the snapshot). The site can then discard updates with timestamp earlier than this Global-Cutoff.

The distributed snapshot algorithm can be simplified somewhat for computing Global-Cutoff. It is not necessary to actually accumulate all updates received (before the corresponding markers) in buffers and then examine them only at the end of the snapshot. Instead, the relevant information can be extracted from each update as it arrives. When an update arrives, Saved-Local-Cutoff is set to the smaller of the timestamp of the update and the current value of Saved-Local-Cutoff. At the end of the snapshot the value of Saved-Local-Cutoff will be the smaller of the original Local-Cutoff and the timestamp on the lowest-timestamped update in transit; this is the same value that is broadcast in the above computation of Global-Cutoff.

A site administrator can advance Local-Cutoff at any time, and can start a new snapshot at any time. Typically, the administrator will wait until the previous snapshot completes, but this is not strictly necessary. Thus, snapshots may be "pipelined" and a site may be involved concurrently in multiple instances that are in different stages of progress. Incoming updates must therefore be checked against the Saved-Local-Cutoffs of all of the snapshots in progress at the site. Messages related to different snapshots (such as markers and final values of Saved-Local-Cutoff) can be sorted out by assigning sequence numbers to successive snapshots.

D. Discussion

Our definition of "Global-Cutoff" is similar to Jefferson's "Global Virtual Time" [5], which is the minimum of all sites' local clock timestamps and the timestamps of all messages in transit. Like sites' Local-Cutoff timestamps, a site clock in Jefferson's model is set backward on the arrival of a lower-timestamped update. The main difference is the context in which the concept is being applied; Jefferson's method is used for all processing in the system, while ours applies only to the determination of a cutoff timestamp and not to the normal issuing and processing of updates. We have also extended the idea to allow more flexibility in determining how far back to reset Local-Cutoff when a lower timestamped update is received.

The protocol we presented for computing a Global-Cutoff can also be used for distributed termination or shutdown of the entire system. If we let *infinity* be a special timestamp larger than any timestamp that can be assigned to an update, the system can shut down when a value of Global-Cutoff equal to infinity is established. This will happen when all sites declare themselves to be "idle" (by setting Local-Cutoff to infinity) and there are no updates in transit. (This point has been noted by Jefferson as well.) A Global-Cutoff of infinity implies that all stored updates are no longer needed and all sites' database copies are in a mutually consistent state that reflects all updates ever issued.

The protocol as described assumes that the set of sites in the system is fixed. We are currently developing a protocol for SHARD that will allow new sites to be added dynamically. The addition of a new site interacts with the discarding of old updates (by existing sites), in that the new site's initial database copy must already reflect any updates that have been discarded by existing sites, and the new site must not issue updates with timestamps smaller than the timestamp of any update already discarded by existing sites. The site addition protocol will be documented separately.

IV. Removing Sites

The protocol for discarding updates requires the active participation of and communication among all sites in the system. If there is a network partition or if some site is down, Global-Cutoff cannot be advanced because one or more sites will be unable to report their Saved-Local-Cutoffs for a given snapshot. When the partition is repaired or the down site recovers, Global-Cutoff can be advanced and updates with earlier timestamp discarded. However, this may take an arbitrary amount of time during which storage space at one or more sites may be exhausted.

Since each site in our system is permitted to issue updates without consulting other sites, there is no way to avoid the above vulnerability to single site failures or disconnections. However, if one or more sites are known to be permanently dead, it is desirable to allow the remaining sites to exclude the dead sites from further consideration and advance Global-Cutoff based on a smaller set of sites. We do not specify how a site is declared to be dead; this will typically involve external investigation by site administrators, possibly triggered by lack of communication from the site in question. It is not necessary for a site to actually have failed in order for it to be removed, so long as the other site administrators decide that they wish to stop communicating with the given site. For example, a site may be faulty and be issuing incorrect updates, in which case removal of the site will limit the damage done to the rest of the system. (Correcting any damage already done will require additional effort by site administrators.)

When the administrator at site x decides that site r is dead and begins *removing* r, x ceases all further communication (over network links) with r. However, if other sites that x is still communicating with have received more updates from r than x has, the system's reliable broadcast algorithm will continue to deliver these updates to x. Thus, x cannot immediately *expunge* r, i.e., forget all

knowledge of r and ignore r in the protocol for discarding updates, until these updates are delivered. The objective of the protocol below is to ensure that a group of sites, say R, is expunged by the remaining group of sites, say S, only when each site in S has received exactly the same set of updates issued by each site in R and will receive no more updates from these sites. The removal of sites must not be held up indefinitely if additional sites fail during execution of the protocol. The protocol allows a site administrator to add more sites to the current set of sites being removed, and then wait for agreement from the remaining sites to expunge the expanded set.

The state information needed by the site removal protocol consists of two two-dimensional tables indexed by site identifier:

Is-Removing[x, r]: A Boolean flag indicating whether or not site x is in the process of removing site r. A site administrator cannot change his mind about removing a given site; once Is-Removing[x, r] becomes true, it cannot become false. Is-Removing[x, x] is always false; a site never attempts to remove itself.

Num-Received[x, r]: The sequence number of the last update from r received by x. Num-Received[x, r] never decreases because updates issued by a given site are delivered in order.

Each site maintains a copy of the above tables. Site x always has the "true" value of Is-Removing[x, r] and Num-Received[x, r]; other sites' copies learn of these values only via messages (described below) and may lag behind the true values. Because of this, Is-Removing[x, r] cannot become true in site y's copy unless it first becomes true in site x's copy; similarly, the value of Num-Received[x, r] at site y is never greater than the value of Num-Received[x, r] at x. A given site x manipulates its copy of the above tables according to the following rules:

1) The administrator at site x issues a command to begin removing site r (other than x itself): Set Num-Received[x, r] to be the sequence number of the last update received from r, and broadcast this value to all sites. Then set Is-Removing[x, r] to true, and broadcast this value.

2) A message is received carrying Is-Removing[y, r] for some y and r: Enter the value in the local copy of Is-Removing. If Is-Removing[x, r] is currently false, alert the site's administrator; he may wish to investigate the situation and decide whether x should also begin removing r.

3) A message is received carrying Num-Received[y, r] for some y and r: Enter the value in the local copy of Num-Received.

4) A new update (with sequence number one greater than Num-Received[x, r]) is received from some site r that is being removed (i.e., Is-Removing[x, r] is true): Increment Num-Received[x, r] and broadcast the new value. Note that this new update is received over the *logical* communication channel (see Section II) from r to x. Because the network link from r and x has been severed, this update can be received by x only if some other site y that x is still in communication with (Is-Removing[x, y] is false) has already received the update.

In order to expunge one or more sites, site x waits for the following condition to become true:

```
for each site r such that Is-Removing[x, r]:
    for each site y such that not Is-Removing[x, y]:
        Is-Removing[y, r] is true
        and Num-Received[y, r] = Num-Received[x, r]
```

Essentially, all sites y that this site is not removing must have agreed to remove all sites r that this site is removing and must have received the same number of updates from each such r. If while waiting for the above the site's administrator decides to begin removing one or more additional sites, the site must now wait for the condition to become true for the expanded set of sites.

The correctness of the above condition can be demonstrated as follows. Suppose that, at time $T1$, site x determines that the above condition is true and expunges some set of sites that includes site r after having received N updates from r; that is, Num-Received[x, r] = N at site x at time $T1$. We then know that the following are true, at site x at time $T1$, for all sites y that x is not removing:

```
Is-Removing[x, y] is false
Is-Removing[y, r] is true
Num-Received[y, r] = N
```

Because the value of Num-Received[y, r] at site x cannot be greater than its value at site y, there can be no site y that x is not removing that has received fewer than N updates from r. Therefore, the only way the protocol can do the wrong thing is if there is some site y that x is not removing such that y receives more than N updates from r. We will prove that this is impossible by contradiction.

Let y be the first site, from the sites that x is not removing, that receives the $(N + 1)$st update from r. Since Is-Removing[y, r] is true at x at $T1$, it must have become true at y (i.e., y began removing r) at some earlier time, say $T0$. Because y broadcasts Num-Received[y, r] when it sets Is-Removing[y, r] true, and because Num-Received[y, r] equals N at time $T1$ at site x, Num-Received[y, r] cannot be greater than N at time $T0$ at site y. Therefore, y can receive the $(N + 1)$st update from r only after it began removing r. However, the rules governing receipt of a new update from a site being removed state that y can receive this update only if some site s that y is not removing (Is-Removing[y, s] is false when y receives this update) has already received the $(N + 1)$st update. If there is such an s, the following must be true at site x at time $T1$ (when x expunges r):

1) Is-Removing[y, s] must be false. If Is-Removing[y, s] were true, y must have set it true at some earlier time. Site y must have received r's $(N + 1)$st update from s even earlier, at which point it must have broadcast Num-Received[y, r] = $N + 1$. Therefore, if x knows that Is-Removing[y, s] is true at time $T1$, it must have already learned that Num-Received[y, r] equals $N + 1$ (or

greater), which contradicts the condition that Num-Received[y, r] equals N at time $T1$.

2) Is-Removing[x, s] must be false (i.e., x is not removing s), because if it were true then x could not expunge r without simultaneously expunging s, which is not possible because y's agreement is needed and Is-Removing[y, s] is false.

Therefore, y could have received the $(N + 1)$st update from r only if some other site s that x is not removing received the update before y did. This is however impossible because it contradicts our assumption that y is the first such site to receive the $(N + 1)$st update.

The protocol above does not allow a site administrator to change his mind once he has initiated the removal of a site. (A future extension of the protocol will allow a site being removed to be "added" back into the system again.) If the administrator at site x decides to remove r but the administrator at y does not, then x's removal of r will not terminate until either y decides to remove r or x decides to remove y as well. That is, x must get all sites other than r to also remove r, or else remove those sites that are not removing r. How the administrators make such decisions is not addressed by the protocol; this may involve communication and negotiation among administrators outside the system. If it is desired to reduce the dependence of the protocol on a human administrator at every site, some or all sites may be programmed to automatically join in the removal of a site when they hear of such removal from any other site, or from any of a set of specified sites, or from some number of sites, or from a set of sites satisfying some other arbitrary predicate.

The site removal protocol allows one group of sites S to remove a group R from its view of the system while R is concurrently removing S. (It is even possible for the administrator at one site to "remove" the rest of the system and continue operation on his own.) This results in a partition of the replicated database into two or more replicated databases. If the two groups are able to communicate at some later time, reintegration of the database copies will not be possible if any group issued a new update with timestamp smaller than the timestamp of any update discarded by the other group. This problem could be avoided by requiring that only a majority group of sites can remove the sites outside the group, but that is too restrictive for our purposes because it does not allow the system to continue operation if a majority of sites die. Rather, we rely on site administrators to perform site removal only when absolutely certain that they do not wish to communicate further with the sites being removed. Recovering from mistakes requires mechanisms (such as manual intervention to patch the divergent database copies) outside the scope of this paper.

V. Conclusion

We presented a protocol that allows sites in a replicated database system to discard old updates that are no longer needed for maintaining mutual consistency. This protocol represents a practical application of Chandy and Lamport's distributed snapshot algorithm, which we used to check whether there are any updates in transit that would cause a site to retract its declared "cutoff" timestamp.

When compared to other systems that discard old updates after an arbitrary expiration period, our protocol provides a more systematic approach that will prevent an old update from being prematurely discarded if it is needed for maintaining mutual consistency, and may allow an unneeded update to be discarded sooner than the expiration time. The discarding of old updates is dependent on how long past errors and missing updates remain relevant in the application, not just on system communication delays. It is left to site administrators to specify the relevance of old information via the sites' cutoff timestamps and the rules used for resetting them. The system configuration can be varied by having different sites implement different rules, and by having some sites automate the advancing of their cutoff timestamps. This provides the system designer with a fine degree of control over how dependent the protocol is on each site's local administrator.

We also presented a site removal protocol that will permit the discarding of old updates to proceed if it has been held up due to lack of response from a dead site. The protocol allows for the simultaneous removal of multiple sites, and therefore works even if additional sites fail during the removal of a site. The site removal protocol is in fact orthogonal to the protocol for discarding old updates, and is not constrained to timestamp-based systems. It is applicable in any situation where one group of sites wishes to cease communication with another group of sites but wishes to do so with a consistent view of what messages were issued by the other group. We hope to explore more general formulations and applications of this protocol.

Acknowledgment

We would like to thank B. Blaustein, C. Kaufman, and M. Siegel for their suggestions during the development of these protocols.

References

[1] B. Awerbuch and S. Even, "Efficient and reliable broadcast is achievable in an eventually connected network," in *Proc. ACM Symp. Principles of Distributed Comput.*, 1984, pp. 278–281.

[2] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder, "Grapevine: An exercise in distributed computing," *Commun. ACM*, vol. 25, no. 4, pp. 260–274, Apr. 1982.

[3] K. M. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, Feb. 1985.

[4] H. Garcia-Molina, N. Lynch, B. Blaustein, C. Kaufman, S. Sarin, and O. Shmueli, "Notes on a reliable broadcast protocol," Computer Corporation of America, Tech. Rep. CCA-85-08, 1985.

[5] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang., Syst.*, vol. 7, no. 3, pp. 404–425, July 1985.

[6] P. R. Johnson and R. H. Thomas, "The maintenance of duplicate databases," Bolt Beranek and Newman Inc., Arpanet Request for Comments (RFC) 677, Jan. 1975.

[7] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, July 1978.

[8] S. K. Sarin, B. T. Blaustein, and C. W. Kaufman, "System architecture for partition-tolerant distributed databases," *IEEE Trans. Comput.*, vol. C-34, pp. 1158–1163, Dec. 1985.

[9] S. K. Sarin, C. W. Kaufman, and J. E. Somers, "Using history information to process delayed database updates," in *Proc. Twelfth Int. Conf. Very Large Data Bases*, Aug. 1986.

Sunil K. Sarin (S'76–M'82) received the B.Tech. degree in electronics and electrical communication engineering from the Indian Institute of Technology, Kharagpur, India, in 1974, and the S.M. and Ph.D. degrees in computer science from the Massachusetts Institute of Technology, Cambridge, in 1977 and 1984, respectively.

He is now a Computer Scientist in the Research and Systems division of Computer Corporation of America. His research interests are in the management of shared distributed information: database systems, concurrency control and recovery, object management for cooperative design applications, network protocols and distributed operating systems, and user interfaces.

Nancy A. Lynch received the B.S. degree in mathematics from Brooklyn College, Brooklyn, NY, in 1968 and the Ph.D. degree in mathematics from Massachusetts Institute of Technology, Cambridge, in 1972.

She is currently Professor of Computer Science at M.I.T., and heads the Theory of Distributed Systems group in M.I.T.'s Laboratory for Computer Science. Her interests are in all aspects of distributed computing theory, including formal models, algorithms, analysis, and correctness proofs. She has served on the faculty of Tufts University, the University of Southern California, Florida International University, and Georgia Tech.