SACRIFICING SERIALIZABILITY TO ATTAIN HIGH AVAILABILITY OF DATA IN AN UNRELIABLE NETWORK*

Michael J. Fischer
Yale University
New Haven, Connecticut

Alan Michael
University of Wisconsin
Madison, Wisconsin

ABSTRACT

We present a simple algorithm for maintaining a replicated distributed dictionary which achieves high availability of data, rapid processing of atomic actions, efficient utilization of storage, and tolerance to node or network failures including lost or duplicated messages. It does not require transaction logs, synchronized clocks, or other complicated mechanisms for its operation. It achieves consistency contraints which are considerably weaker than serial consistency but nonetheless are adequate for many dictionary applications such as electronic appointment calendars and mail systems. The degree of consistency achieved depends on the particular history of operation of the system in a way that is intuitive and easily understood. The algorithm implements a "best effort" approximation to full serial consistency, relative to whatever internode communication has successfully taken place, so the semantics are fully specified even under partial failure of the system. Both the correctness of the algorithm and the utility of such weak semantics depend heavily on special properties of the dictionary operations.

*This work was supported in part by the Office of Naval Research under Contract N00014-80-C-0221 and by the National Science Foundation under Grants MCS80-03337, MCS80-04111, and MCS81-16678. The research was carried out in part at the University of Washington.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1982 ACM 0-89791-070-2/82/003/0070 $00.75

1. Introduction

A common axiom taken for the correctness of a database system is that the transactions be serializable, that is, the results of any sequence of transactions should be the same as if they had been performed in some serial order [3, 5, 17, 19]. Serializability insures consistency of the database when concurrent transactions are being processed assuming only that each transaction is correct when run alone.

Achieving serial consistency in an unreliable distributed environment is considerably more difficult than in a central database, and much work has been done addressing this problem [2, 4, 9, 20]. (Cf. [16] for a nice survey of some of the issues.) Reasons for distributing data in the first place are to increase speed of access and to insure availability of data even when individual nodes or the network itself fails. Both of these goals require replication of the data, which introduces the new problem of keeping the replicated copies up-to-date. (Cf. [7, 8, 13, 15, 21].)

Unfortunately, the two goals of availability and serial consistency stand somewhat in conflict. For example, availability dictates that every node with a copy of the database be permitted to continue performing transactions on its local copy even when the network fails. Serializability, on the other hand, requires that at most one such node be allowed to proceed under such conditions, for otherwise the copies begin diverging and reads can return values inconsistent with any serial ordering of the transactions.

Several authors have noted that meaningful results can often be obtained even without serial consistency when additional information about the particular transactions is available [10, 11, 12]. Also, strict serializability is often not required for read-only transactions [6, 18]. We present an example of a problem which is adequately served by

a database satisfying much weaker conditions and give an algorithm for its solution. Our algorithm achieves high availability of data, rapid processing of atomic actions, efficient utilization of storage, and tolerance to node or network failures including lost or duplicated messages. It does not require transaction logs, synchronized clocks, or other complicated mechanisms for its operation.

The degree of consistency achieved by our solution depends on the particular history of operation of the system in a way that is intuitive and easily understood. The algorithm implements a "best effort" approximation to full serial consistency, relative to whatever internode communication has successfully taken place, so the semantics are fully specified even under partial failure of the system.

Johnson and Thomas [10] give an algorithm for a similar problem which uses timestamps to serialize updates (cf. [14]) but permits arbitrary reading. While it enjoys many of the same advantages of our algorithm, it requires deleted dictionary entries to be retained until all processes have updated their copies of the database. Also, their read semantics are somewhat weaker than ours.

Algorithms such as [7, 21] which use voting schemes are able to provide both serial consistency and data availability despite limited node failures, but like all serializable algorithms, updates in all but one subnet are disallowed when the network becomes partitioned.

2. Distributed Dictionaries

Abstractly, our problem is to maintain a database consisting of a dictionary, that is, a set of elements with two update operations INSERT and DELETE, and a single query operation LIST (cf. [1]). INSERT(x) adds element x to the set, DELETE(x) removes x from the set if it was there and does nothing otherwise, and LIST returns an enumeration of the elements currently in the set. All three operations are considered to be atomic transactions.

The database is to be implemented on an unreliable network of processors. Our goal is to make the database highly available, even under conditions in which individual nodes and the network are not always operational. By "available", we mean that any operational node should be able to perform any of the basic database operations at any time, regardless of the status of the rest of the system.

Each node maintains its own copy or view of the database, and all operations are performed initially only on the node's local view. From time to time a node sends information about its view to one or more other nodes. A node receiving such information then updates its own view. We have in effect added two new operations: SEND(m) and RECEIVE(m), where m is the message. As more and more messages are sent, information is thus propagated throughout the network, and the individual views of the data tend to converge to the view that would be "correct" were this all taking place in a centralized database.

Our notion of correctness depends not only on the particular update and query operations requested by the users of the system but also on the internal communications that have taken place, about which we make no assumptions. The intention is that in a correctly functioning system, enough communication will take place so that every node of the system will know about an insertion or deletion shortly after it occurs, and no view will be far out of date. However, our correctness condition is simply that an element x is in node i's view iff i knows of its insertion but does not know of its deletion.

We place two restrictions on the problem:

R1. We assume that there is at most one occurrence of the operation INSERT(x) for each element x, so that once an element has been deleted from the set, it can never again be reinserted.

R2. DELETE(x) is only legal at a node j if x is currently in j's view.

We need both restrictions for technical reasons. Among other things, they insure that INSERT(x) can never follow DELETE(x), so if a node discovers that both operations were performed sometime in the past, then x definitely does not belong in its view. Also, both restrictions arise naturally in many applications. One way to enforce restriction R1 is to tag the actual datum with a "timestamp" which uniquely identifies the particular insertion. Thus, two attempts to insert the same datum will in fact give rise to two different elements x and x' with different timestamps. Restriction R2 is natural in applications where the only way to specify an argument to DELETE is to "point" at the element among the ones in the current view. Such is generally the case, for example, when the elements are tagged with timestamps. Note that we do permit several deletions of the same element; their effect is the same as a single one.

This abstract problem was motivated by the practical problem of building a highly available electronic appointment calendar. Here the data items are individual appointments, and an appointment calendar is just a set of appointments. A user can read and modify his appointments from any node. He will see every appointment that it is possible to see, given the interprocess communication that has actually taken place. In a fully working system he would see all but possibly very recently entered appointments. Anything he can see he can manipulate as if he were working on a centralized system. Finally, any changes he makes will be reflected at the other nodes when the system is again working, even if the network happens to be unavailable while he is actually doing the modifications. Note that because the views are not always up-to-date, conflicting appointments may not be discovered immediately. Hence, it is necessary for the calendar system to be able to handle conflicts at times other than when an appointment is first entered. (This is probably a desirable property anyway.)

Other places where this problem arises are in distributed mail systems and distributed file directory systems, both of which abstractly just maintain dictionaries. In a distributed mail system, our solution could simplify the usual network mailer. The network mailer would only have to deliver a message to one of a user's mailboxes; the distribution of mail to the user's other mailboxes would then be handled by our algorithm. Indeed, if the recipient had a local mailbox, then only the local mailer would be needed and the network mailer would not have to be invoked at all.

3. Formal Problem Statement

For each natural number N, let [N] = {1, 2, ..., N}. Let D be the domain of elements. Let OP = {'INSERT(x)', 'DELETE(x)' | x ε D} U {'LIST'} U {'SEND(m)', 'RECEIVE(m)' | m is a message}. We formulate our correctness conditions in terms of a partial order of events which represents the history of information flow in the system.

Fix a particular execution of the system. Each instance of an operation ξ ε OP corresponds to an event e, where op(e) = ξ and node(e) is the node at which e occurs. Let E be the set of all events occurring in the execution, and let D(E) = {x ε D | op(e) = INSERT(x) for some e ε E}. E is partially ordered by "$\rightarrow$", which is the least reflexive and transitive relation such that:

P1. Events at the same node are totally ordered;

P2. If e_1 = SEND(m) and e_2 = RECEIVE(m) for the same message m, then $e_1 \rightarrow e_2$.

We now formalize a correct view of the database. We represent our notion of "knows about" by $\rightarrow$; hence, when i has just performed event e', it knows about an event e iff $e \rightarrow e'$. Let view: $E \rightarrow 2^D$ be defined as follows: x ε view(e') iff

V1. there is an event e such that $e \rightarrow e'$ and op(e) = INSERT(x), and

V2. for every event e, if op(e) = DELETE(x), then $e \not\rightarrow e'$.

We now define the N-node redundant dictionary problem to be the problem of finding a distributed algorithm on N nodes such that each node can process the operations of INSERT, DELETE, LIST, SEND and RECEIVE, subject to restrictions R1 and R2, and each node i maintains a correct view of the data V_i. That is, in the partial order of events corresponding to the history of operations in the system, if e is an event at node i, then just after the occurrence of that event, V_i = view(e).

4. The Algorithm

An obvious solution to our dictionary problem is the following: Each node i maintains two sets, I_i and D_i, which are the sets of elements that node i knows have been inserted and deleted, respectively. i's view of the dictionary is $V_i = I_i - D_i$. To implement SEND(m), node j sends a message m containing I_j and D_j. When a node i does a RECEIVE(m), it updates its own sets simply by taking unions.

The drawback to this solution is that the set $I_i \cup D_i$ contains every element that has ever been in i's view, and this set grows without bound, even if the size of the view is itself bounded. Our algorithm gets by with keeping only the current view, V_i, and a small amount of additional information which will be described shortly.

Clearly, it won't do to update V_i by replacing it with $V_i \cup V_j$, for there can be two reasons why an element $x \in V_i \cup V_j$ might be missing from one of the sets V_k, k ε {i,j}:

1. x used to be in V_k but it has since been deleted, or

2. x was inserted so recently that node k has not yet heard about it.

In case 2, x belongs in V_i (and in V_j, too), and in case 1, it should be in neither.

In order to be able to distinguish these two cases, each node maintains the following information in addition to its current view of the database:

1. Each node i has a cell "$clock_i$". Each reference to $clock_i$ returns a positive number that is larger than all previous values returned. ($Clock_i$ can be implemented by a physical clock or by a counter that gets incremented on each reference. We talk about the values of $clock_i$ as being "times", but they need bear no relation to real time nor to the values of $clock_j$ for any $j \neq i$.)

2. Each x in the view is tagged with a pair (cre_x, T_x), where cre_x, the "creator" of x, is the node at which x was originally inserted, and T_x is the time, according to the clock of cre_x, at which the insertion took place.

3. Each node i maintains a table t_i. $t_i(j)$ is i's posting time for insertions which took place at node j.

The posting times tell how current i's knowledge is about insertions that have occurred at the other nodes: i knows about an insertion at node j iff the time at which that operation occurred (according to $clock_j$) is $\leq t_i(j)$.

Given a view V, a posting time vector t, and an element x, we define a predicate:

$$del(V,t,x) \text{ iff } [x \notin V \text{ and } T_x \leq t(cre_x)].$$

It will follow that $del(V_i, t_i, x)$ holds iff node i knows that x has been deleted. We now describe how node i processes each of the kinds of operations.

Algorithm

INSERT(x): $t_i(i) := clock_i$;
$cre_x := i$;
$T_x := t_i(i)$;
$V_i := V_i \cup \{x\}$.

DELETE(x): $V_i := V_i - \{x\}$.

LIST: Return V_i.

SEND(m): Send the message $m = \langle V_i, t_i \rangle$.

RECEIVE(m): Let m be the message $\langle \bar{V}, \bar{t} \rangle$;
$V_i := \{ x \in (V_i \cup \bar{V}) \mid \sim del(V_i, t_i, x) \text{ and } \sim del(\bar{V}, \bar{t}, x) \}$;
$t_i(k) := \max\{ t_i(k), \bar{t}(k) \}$ for all $k \in [N]$.

Initially, $t_i(j) = clock_i = 0$ and $V_i = \phi$ for all i, j.

5. Proof of Correctness

Before stating and proving the correctness of this algorithm, we need some more notation. For each event $e \in E$, let V[e] (respectively t[e]) be the value of $V_{node(e)}$ (respectively $t_{node(e)}$) immediately after completing e. Let $ins_x[e]$ be the predicate $T_x \leq t[e](cre_x)$. Let $del_x[e] = del(V[e], t[e], x)$. Note that $del_x[e]$ iff $x \notin V[e]$ and $ins_x[e]$. We will show that V[e] corresponds to the current view, $ins_x[e]$ means that x is known to have been inserted and $del_x[e]$ means that x is known to have been deleted.

Let $e \xrightarrow{1} e'$ iff $e \rightarrow e'$, $e \neq e'$, and for all e", if $e \rightarrow e'' \rightarrow e'$, then e" = e or e" = e'. If $e \xrightarrow{1} e'$, we say that e is an immediate predecessor of e' and e' is an immediate successor of e.

Lemma 1: If $e \rightarrow e'$, then $t[e](i) \leq t[e'](i)$.

Thus, posting times are monotone over "$\rightarrow$".

Proof: Obvious by inspection of the algorithm and the conditions on $clock_i$. □

Lemma 2: If $x \in V[e']$, then there exists $e \in E$ such that op(e) = INSERT(x) and $e \rightarrow e'$.

Proof: This follows by an easy induction on "$\rightarrow$", using the fact that initially all $V_i = \phi$. □

Lemma 3: Let $x \in D(E)$, $e' \in E$. Then $ins_x[e']$ iff there exists $e \in E$ such that op(e) = INSERT(x) and $e \rightarrow e'$.

Proof: =>: Assume $ins_x[e']$ and $x \in D(E)$. First, $t[e'](cre_x) \geq T_x > 0$. Let e" be minimal in E such that $e'' \rightarrow e'$ and $t[e''](cre_x) = t[e'](cre_x)$. Inspection of the code shows that $node(e'') = cre_x$ and op(e") = INSERT(y) for some $y \in D$, for in every other case, at least one immediate predecessor f of e" has $t[f](cre_x) = t[e''](cre_x)$, contrary to the minimality of e". Since $x \in D(E)$, there exists $e \in E$ with op(e) = INSERT(x) and

node(e) = cre_x. By condition P1, either e $\rightarrow$ e" or e" $\rightarrow$ e. If e" $\rightarrow$ e, we have $T_x \leq t[e'](cre_x)$ = $t[e''](cre_x) < t[e](cre_x) = T_x$ (since op(e) = INSERT(x)), a contradiction. Hence, e $\rightarrow$ e" $\rightarrow$ e'.

<=: Immediate by the code for INSERT and Lemma 1. □

Lemma 4: If e" $\rightarrow$ e' and $del_x[e'']$, then $del_x[e']$.

Proof: It suffices to show that if e" $\xrightarrow{1}$ e' and $del_x[e'']$, then $del_x[e']$. Since $del_x[e'']$, then x $\notin$ V[e"] and $ins_x[e'']$. By inspection of the code and restriction R1, x $\notin$ V[e']. By Lemma 1, $ins_x[e']$, so $del_x[e']$ holds. □

Lemma 5: Let x $\in$ D(E), e' $\in$ E. Then $del_x[e']$ iff there exists e" $\in$ E such that op(e") = DELETE(x) and e" $\rightarrow$ e'.

Proof: =>: Assume $del_x[e']$ holds. Let e" $\in$ E be minimal such that $del_x[e'']$ and e" $\rightarrow$ e'. Then x $\notin$ V[e"] and $ins_x[e'']$, so by Lemma 3, there exists e $\in$ E such that op(e) = INSERT(x) and e $\rightarrow$ e". Let f be such that e $\rightarrow$ f $\xrightarrow{1}$ e" (possible since e $\neq$ e"). $\sim del_x[f]$ by minimality of e", and $ins_x[f]$ by Lemma 3; hence, x $\in$ V[f]. Since x $\notin$ V[e"], then op(e") is DELETE(x) or RECEIVE(m) for some m. However, if op(e") = RECEIVE(m), then x $\in$ V[e"] by the code for RECEIVE (since $\sim del_x$ holds for all predecessors of e"), a contradiction. We conclude that op(e") = DELETE(x).

<=: Assume op(e") = DELETE(x) and e" $\rightarrow$ e'. x $\notin$ V[e"] by the code for DELETE(x). By restriction R2, there is an immediate predecessor f of e" such that x $\in$ view(f). By condition V1, there is an e $\in$ E such that op(e) = INSERT(x) and e $\rightarrow$ f. Thus, $ins_x[e'']$ by Lemma 3, so $del_x[e'']$. By Lemma 4, $del_x[e']$. □

We now show the correctness of our algorithm:

Theorem: For all e' $\in$ E, view(e') = V[e'].

Proof: Suppose x $\in$ view(e'). By condition V1, there exists e $\rightarrow$ e' such that op(e) = INSERT(x). By Lemma 3, $ins_x[e']$. By condition V2, for every e" with op(e") = DELETE(x), then e" $\not\rightarrow$ e'. Hence, we can apply Lemma 5 to conclude $\sim del_x[e']$, so x $\in$ V[e'].

Now suppose x $\in$ V[e']. By Lemma 2, e $\rightarrow$ e' and op(e) = INSERT(x) for some e $\in$ E. Hence, condition V1 holds for e'. Also, $ins_x[e']$ holds by Lemma 3. Let op(e") = DELETE(x). Since $\sim del_x[e']$, we conclude from Lemma 5 that e" $\not\rightarrow$ e'. Thus, condition V2 holds for e', so x $\in$ view(e').

We conclude that view(e') = V[e']. □

6. Remarks and Open Problems

We have not yet addressed the problem of finding a good strategy for the nodes to use in deciding when and how to communicate.

If each message can be received by only a single process, then various strategies can be imagined. At one extreme, a message transmission from i to j could be attempted periodically for all pairs i, j, i $\neq$ j, resulting in a total of $\Omega(N^2)$ messages to propagate information between all pairs of nodes. On the other hand, given a spanning tree in the network and a root, one can propagate information from every node to every other using only O(N) messages by first sending a wave of messages up from the leaves to the root and then back down from the root to the leaves. However, recovering from a network or node failure requires a special recovery procedure since the spanning tree must be rebuilt. We leave as an open problem to find a robust O(N)-message algorithm for propagating data throughout the system.

If a broadcast facility is available, then things are much simpler, for each node need only broadcast a single message. There is still the problem, however, of how often to do so. It is clearly not sufficient for a node to broadcast only when it has new information, for a node restarting after a failure must have some means for being brought up-to-date. Of course, various protocols can be imagined to handle such situations, and we leave that also as an open problem.

Acknowledgement

The authors are grateful to Nancy Lynch for bringing to their attention several of the references listed below.

References

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, Mass., 1974.

2. P.A. Bernstein, N. Goodman, J.B. Rothnie, and C.H. Papadimitriou. "Analysis of Serializability of SDD-1: A System of Distributed Databases (the fully redundant case)." IEEE Trans. on Software Eng. SE-4, 3 (May 1978), 154-168.

3. P.A. Bernstein, D.W. Shipman, and W.S. Wong. "Formal Aspects of Serializability in Database Concurrency Control." IEEE Trans. on Software Eng. SE-5, 3 (May 1979), 203-216.

4. P.A. Bernstein, D.W. Shipman, and J.B. Rothnie. "Concurrency Control in a System for Distributed Databases (SDD-1)." ACM Transactions on Database Systems 5, 1 (March 1980), 18-51.

5. K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. "The Notions of Consistency and Predicate Locks in a Database System." Comm. ACM 19, 11 (Nov. 1976), 624-633.

6. H. Garcia-Molina and G. Wiederhold. Read-Only Transactions in a Distributed Database. Tech. Rept. STAN-CS-80-797, Computer Science Department, Stanford University, April, 1980.

7. D.K. Gifford. Weighted Voting for Replicated Data. Tech. Rept. CSL-79-14, XEROX Palo Alto Research Center, Sept., 1979.

8. M. Hammer and D. Shipman. "Reliability Mechanisms for SDD-1: A System for Distributed Databases." ACM Transactions on Database Systems 5, 4 (Dec. 1980), 431-466.

9. D. Jacobson and W. Chou. Synchronization Strategies for Updating Distributed Data Bases. Tech. Rept. TR 80-17, North Carolina State University, October, 1980.

10. P.R. Johnson and R.H. Thomas. The Maintenance of Duplicate Data Bases. Network Information Center (NIC) Document #31507, Bolt Beranek and Newman, Inc., Jan., 1975. Also referred to as ARPA Network Working Group Request for Comments (RFC) #677.

11. H.-T. Kung and C.H. Papadimitriou. An Optimality Theory of Concurrency Control for Databases. Tech. Rept. MIT/LCS/TM-185, Laboratory for Computer Science, M.I.T., Nov., 1980. Also appeared as Carnegie-Mellon University Technical Report CMU-CS-80-147.

12. L. Lamport. Towards a Theory of Correctness for Multi-user Data Base Systems. Tech. Rept. CA-7610-0712, Massachusetts Computer Associates, Inc., Oct., 1976.

13. L. Lamport. "The Implementation of Reliable Distributed Multiprocess Systems." Computer Networks 2 (1978), 95-114.

14. L. Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System." Comm. ACM 21, 7 (July 1978), 558-565.

15. B.W. Lampson and H.E. Sturgis. Crash Recovery in a Distributed Data Storage System. XEROX Palo Alto Research Center, April, 1979. To be published in Comm. of the ACM.

16. B.G. Lindsay et. al. Notes on Distributed Databases. Research Report RJ2571(33471), IBM Research, July, 1979.

17. C.H. Papadimitriou. "The Serializability of Concurrent Database Updates." J. ACM 26, 4 (Oct. 1979), 631-653.

18. G. Popek et. al. LOCUS: A Network Transparent, High Reliability Distributed System. Proc. Eighth Symp. on Operating Systems Principles, ACM (SIGOPS), Dec., 1981, pp. 169-177.

19. D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II. Consistency and Serializability in Concurrent Database Systems. Tech. Rept. 80-12, Department of Computer Science, SUNY Albany, Aug., 1980.

20. J.B. Rothnie et. al. "Introduction to a System for Distributed Databases (SDD-1)." ACM Transactions on Database Systems 5, 1 (March 1980), 1-17.

21. R.H. Thomas. "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases." ACM Transactions on Database Systems 4, 2 (June 1979), 180-209.