Efficient Solutions to the Replicated Log and Dictionary Problems*

Gene T.J. Wuu and Arthur J. Bernstein

Departmemnt of Computer Science

State University of New York at Stony Brook

Long Island, NY 11794

Abstract

We propose efficient algorithms to maintain a replicated dictionary using a log in an unreliable network. A non-serializable approach is used to achieve high concurrency. The solutions are resilient to both node and communication failures. Optimizations are developed for networks which are not completely connected.

1. Introduction

Using serializability as a consistency constraint on data is suitable in most applications since knowledge of the semantics of the data is not required. However, such generality complicates the implementation and degrades performance, especially when considering a replicated database in an unreliable distributed system. Enforcing serializability restricts concurrency [1] [2] [3], and adds considerable communication overhead to support synchronization in a distributed system [6] [4]. Commit protocols are necessary if serialized transactions are to be made failure atomic, thus imposing additional cost [4] [5] [10].

Concurrency and serializability are compatible only when concurrent transactions access disjoint data areas or are mostly read only. Unfortunately, in many applications data objects are highly shared and reliability and fast access are very important. Data replication is encouraged for these purposes. Algorithms [6] [11] [4] have been proposed to keep replicated copies up to date and mutually consistent. However, performance is limited if serializability is used as a consistency constraint. Recently, research [1] [2] [3] has been focused on how to increase concurrency if the semantics of the data is known.

In this paper we use a replicated log to achieve mutual consistency of replicated data in an unreliable network. The log contains records of invocations of operations which access a data object. Each node updates its local copy of the data object by performing the operations contained in its local copy of the log. If the semantics of the data are such that operations are generally commutative, then the order in which operations are performed does not effect the final state. Hence it is not necessary to require serializability in order to guarantee that the copies of the data object are mutually consistent. In this paper, we propose efficient algorithms to maintain a replicated dictionary in an unreliable distributed system using a log. The dictionary problem was first defined and solved by Fischer and Michael [1]. Allchin [8] improves the solution by reducing communication requirements. The algorithms described here reduce communication requirements further and have performance advantages in failure situations. Furthermore, they can be used to solve other similar problems. Logs are generally useful in achieving distributed synchronization [9], and in gathering

*Research sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, USAF, under grant number AFOSR 81-0197. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon.

information about the state of the network.

The paper is organized as follows. The distributed environment is described in section 2. The log and dictionary problems are defined in section 3. Efficient solutions to these problems are developed in section 4. In section 5, the algorithm is compared with other techniques. Optimizations are discussed in section 6. Proofs are given in the appendix.

2. The Model of the Environment

An n node connected network is considered with nodes N_1, N_2, ..., N_n. A node id is an integer in [n], where [n] denotes the set $\{1,...,n\}$. Two kinds of operations are supported -- the communication operations send and receive, and non-communication operations, which vary with the application. We assume a one-to-one communication structure, but the results can easily be extended to multicast communication. Operations may be invoked at different nodes at any instant. The local data at each node is stored in stable storage, which survives node crashes. An event model [7] is used. An event is a particular execution of an operation. Events are locally atomic in the sense that, if a node crashes during the execution of an operation, there will be no effect on the local data. Since distinct events may invoke the same operation with the same parameters, events are distinguished by the time and place at which they occur. We assume that there is a local clock at each node. (Clocks are not necessarily synchronized.) Let e be an event. time(e) and node(e) denote the time and the node, respectively, at which e occurs. op(e) denotes the invoked operation and its parameters. An **execution** of a distributed system consists of a set, E, of events and a partial ordering relation, $\rightarrow$, on E, denoted by $<E,\rightarrow>$. The partial ordering relation is the smallest relation satisfying the following two conditions:

(C1) Events occurring at the same node are totally ordered.

(C2) Let e_1 be a send event and e_2 the corresponding receive event. Then $e_1 \rightarrow e_2$.

Note that these two conditions do not imply that messages arrive in the order in which they were sent. The relation $\rightarrow$ is called **happened before.** e_1 and e_2 are said to be **concurrent** if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$.

The event model is useful in characterizing an unreliable distributed system in which a send event is not necessarily matched by a receive event. The unreliable behavior covered by this model includes lost messages, broken communication links, network partition, and failed nodes. We do not, however, allow messages in transit to be changed in arbitrary ways as in [12].

3. The Log Problem and its Applications

A common technique for synchronizing a distributed system is to construct a log of certain events which have occurred in the network [9]. Each node maintains its own view of the log and a distributed algorithm is employed to keep the views up to date. The algorithm makes use of communication operations and the log records non-communication events.

3.1. The Log Problem

An event in the log is described by a record with the structure :

```
type Event =
    record
        op   :   OperationType;
        time :   TimeType;
        node :   NodeId;
    end
```

The event record describing event e is denoted by eR, and eR.op = op(e), eR.time = time(e), and eR.node = node(e). Let L_i denote the local copy of log L at N_i. Let e be an event. L(e) denotes the content of $L_{node(e)}$ immediately after e completes. **The log problem** is defined to be the problem of finding an algorithm to maintain the log such that, given an execution $<E,\rightarrow>$, for every event e,

(P1) $f \rightarrow e$ iff $fR \in L(e)$.

Nodes exchange messages containing appropriate portions of the log in order to achieve (P1). Consider a network of two nodes, N_1 and N_2. If N_1 only sends the records of those events which have occurred at N_1 since it last sent a message to N_2, then (P1) will not be achieved since message delivery is not guaranteed. A trivial solution to this problem is therefore the following: at each non-communication event, e, occurring at N_i, N_i inserts eR into L_i; at each send event N_i includes L_i in the message. Note that since L_i grows unboundedly, the solution uses an excessive amount of communication bandwidth. We will develop a more efficient solution in section 4.

3.2. The Dictionary Problem

A dictionary is an abstraction of a data object such as a file directory, a database dictionary, a resource management table, a naming table, ..., etc. It consists of a set of entries. A fully replicated dictionary V is considered. V_i denotes the local copy of V at N_i. It represents N_i's view of the dictionary. V(e) denotes the content of $V_{node(e)}$ immediately after e completes. Two non-communication operations are provided in this problem, insert(x), and delete(x), where x is a dictionary entry. (A lookup operation is also allowed, but since it does not change the structure of the solution we will ignore it here.) Two restrictions are imposed :

(R1) delete(x) can be invoked at N_i only if $x \in V_i$ immediately prior to the execution of the operation, and

(R2) for each dictionary entry x there is at most one event, insert(x), in E.

We use c_x to denote the unique event which inserts x. Note that there may be more than one event which deletes x concurrently. We call an event which deletes x an **x-delete event.** (R2) is easily satisfied by tagging each item to be inserted with $<node(c_x),time(c_x)>$, thus forcing the uniqueness of each entry. **The dictionary problem** is defined to be the problem of finding an algorithm for maintaining the dictionary such that, given an execution $<E,\rightarrow>$, for every event, e,

(P2) $x \in V(e)$ iff $c_x \rightarrow e$ and there does not exist an x-delete event, g, such that $g \rightarrow e$.

An obvious solution to this problem using a log is as follows. At each event, e, such that node(e) = i, N_i computes V(e) in the following way :

$V(e) := \{x \mid c_xR \in L(e)$ **and** $(\nexists\ gR \in L(e))gR.op = delete(x)\}$.

The correctness of the solution can be seen from (P1). Its inefficiency is discussed in the next section.

4. Efficient Solutions

Since a log has unbounded size, the above solutions to the log and dictionary problems are not efficient for three reasons.

(O1) The entire log is sent in each message, implying excessive communication costs.

(O2) A new view of the dictionary is repeatedly computed, implying excessive computational costs.

(O3) The entire log is kept at each node, implying excessive storage costs.

We will first develop a more efficient solution to the log problem which overcomes (O1) and then use it as the basis of a solution to the dictionary problem.

4.1. A New Solution to the Log Problem

In the trivial solution of section 3.1, once eR is inserted into L_i it will be included as part of each message sent by N_i. However, once N_i knows that $eR \in L_j$, it does not have to include eR in subsequent messages to N_j. To make such information available a data structure called a **2-Dimensional Time Table** (2DTT) is used by each node so that a node can tell how up to date other nodes are about events occurring in the network.

4.1.1. The algorithm

N_i maintains the following information:

1. A service called $clock_i$.

   type TimeType = integer;

   Each reference to $clock_i$ returns an integer number greater than that returned by the last reference. We do not require the clocks at different nodes to be synchronized.

2. A 2-dimensional time table, T_i.

```
var $T_i$ : array [1..n,1..n] of TimeType;
```

The algorithm to be described operates on T_i in such a way that $T_i[k,u]$ is a value originally obtained from $clock_u$ (and passed to N_i through a sequence of messages) at the time a non-communication event occurred at N_u. If $T_i[k,u] = t$ then N_i knows that N_k has learned of all events which have occurred at N_u up to t. N_k may, in fact, be aware of events which have occurred at u more recently than t, so that t merely serves as a lower bound on k's information about u. Note that $T_i[i,i]$ is the value of $clock_i$ at the occurrence of the most recent non-communication event at N_i and $T_i[i,u]$ is the value of $clock_u$ at the occurrence of the most recent non-communication event which has occurred at N_u of which N_i is aware.

3. A copy of the log.

```
var $L_i$ : SetOf Event;
```

We define the predicate hasrec to be

$$\text{hasrec}(T_i,eR,k) \equiv T_i[k,eR.node] \geq eR.time.$$

The algorithm is designed to guarantee that if hasrec(T_i,eR,k) is true at N_i then N_k has learned of e. Due to the time lag in the exchange of information in the network the converse does not necessarily hold. The efficiency gains of the algorithm are based on the fact that N_i need not send eR to N_k if hasrec(T_i,eR,k) is true. The following is the algorithm executed at N_i.

```
Initialization:
begin
  $L_i := \emptyset$;
  $(\forall I \in [n]) (\forall J \in [n])$ do $T_i[I,J] := 0$
end ;

oper(p):
(* some non-communication operation *)
begin
  $T_i[i,i] := clock_i$;
  $L_i := L_i \cup \{ <$"oper(p)",$T_i[i,i],i> \}$;
  perform the operation oper(p)
end ;

send(m) to $N_k$:
begin
  NP := $\{eR | eR \in L_i$ and $\neg$hasrec$(T_i,eR,k)\}$;        (S1)
  send the message $<NP,T_i>$ to $N_k$
end ;

receive(m) from $N_k$:
begin
  let m = $<NP_k,T_k>$;
  $L_i := L_i \cup NP_k$;
  $(\forall I \in [n])$ do $T_i[i,I] := \max\{T_i[i,I], T_k[k,I]\}$;        (S2)
  $(\forall I \in [n]) (\forall J \in [n])$ do
        $T_i[I,J] := \max\{T_i[I,J], T_k[I,J]\}$
end ;
```

If multicast communication is possible, a node may use a set, S, of nodes as the destination of a send operation. Statement (S1) should then be modified as follows :

$$NP := \{eR | eR \in L_i \textbf{ and } (\exists k \in S) \neg \text{hasrec}(T_i,eR,k)\};$$

A correctness proof for the algorithm is given in the appendix.

4.2. An Efficient Solution to the Dictionary Problem

In this section an algorithm is developed to overcome deficiencies (O2) and (O3). A new view of the dictionary does not have to be computed from the entire log nor does the entire log have to be stored. Instead each node separately maintains its view of the dictionary as well as a partial log which records some events which have happened in the network. At each receive event a node extracts a set, NE, of events, of which it has not yet learned, from NP. (Note that the node may have learned of some of the events in NP.) The node then updates its view using NE; the log is no longer needed for this purpose. Suppose that $(\forall j \in [n])\ T_i[j,eR.node] \geq eR.time$. Then since eR will never be sent by N_i, there is no reason for N_i to keep a copy of it. Hence, only a partial log, PL_i, is kept at each node. The declarations and algorithm executed at N_i are as follows.

```
var
  $V_i$  : SetOf DictionaryEntry;
  $PL_i$ : SetOf Event;
```

```
Initialization:
begin
  $V_i := \emptyset$;
  $PL_i := \emptyset$;
  $(\forall I \in [n])\ (\forall J \in [n])$ do $T_i[I,J] := 0$
end ;

insert(x):
begin
  $T_i[i,i] := clock_i$;
  $PL_i := PL_i \cup \{ <\text{"insert(x)"}, T_i[i,i], i> \}$;
  $V_i := V_i \cup \{ x \}$
end ;

delete(x):
begin
  $T_i[i,i] := clock_i$;
  $PL_i := PL_i \cup \{ <\text{"delete(x)"}, T_i[i,i], i> \}$;
  $V_i := V_i - \{ x \}$
end ;

send(m) to $N_k$:
begin
  $NP := \{eR | eR \in PL_i$ and $\neg hasrec(T_i, eR, k)\}$;
  send the message $<NP, T_i>$ to $N_k$
end ;

receive(m) from $N_k$:
begin
  let $m = <NP_k, T_k>$;
  $NE := \{fR | fR \in NP_k$ and $\neg hasrec(T_i, fR, i)\}$;
  $V_i := \{v | (v \in V_i$ or $c_vR \in NE)$ and
          $(\nexists\ dR \in NE) dR.op = delete(v)\}$;
  $(\forall I \in [n])$ do $T_i[i,I] := \max\{T_i[i,I], T_k[k,I]\}$;        (S3)
  $(\forall I \in [n])\ (\forall J \in [n])$ do
        $T_i[I,J] := \max\{T_i[I,J], T_k[I,J]\}$;                  (S4)
  $PL_i := \{eR | eR \in (PL_i \cup NE)$ and
          $(\exists j \in [n])\ \neg hasrec(T_i, eR, j)\}$
end ;
```

How does this algorithm behave when failures occur? (1) If the communication link between N_i and N_j breaks or a message sent from N_i to N_j is lost, N_j will still learn, indirectly, of new events at N_i using information passed through other nodes. (2) If N_k fails, then $T_i[k,j]$ can never become greater than $T_k[k,j]$ at the time of failure. This follows from the fact that $T_i[k,j]$ increases monotonically as a result of the application of the max function when a message is received. A new (largest) time value makes its initial appearance in $T_j[j,j]$ when a non-communication event occurs at N_j, migrates to $T_k[k,j]$ when N_k receives a message from N_j (S3), and then to $T_i[k,j]$ when N_k communicates with N_i (S4). Thus the record of any event, e, occurring at N_j such that eR.time $> T_k[k,j]$ cannot be deleted from any partial log and will be contained in messages received by N_k when it recovers. Note that we have assumed that the state at N_k, consisting of V_k, PL_k, and T_k, is maintained in stable storage and hence not lost when a crash occurs. The correctness of the algorithm is proven using an argument similar to that shown in the appendix. A complete proof is given in [13].

5. Comparison with Other Work

Fischer and Michael [1] establish an elegant framework for the multiple copy dictionary problem. However the solution requires a node to send its entire copy of the dictionary in each message. In some applications, the average size of the dictionary is large, and therefore this solution is expensive.

In Allchin [8], each node, N_i, maintains a synchronization set, SS_i, and a 1-Dimensional Time Table, T_i' (which is equivalent to row i in the 2DTT), in addition to its view of the dictionary. The elements of SS_i are records of insert and delete events, thus SS_i corresponds to a partial log. A set, called the knownby set, is a part of each event record and contains node ids of nodes which have learned of the event. When the knownby set contains all node ids, the event record can be deleted from the synchronization set. It can be shown [13] that this occurs under the same circumstances which cause an event record to be deleted from the partial log in the algorithm described here. The difference between the approaches is that with a 2DTT, N_k does not send eR to N_i if N_k knows that N_i has learned of the event. Hence only a subset of PL_k is sent in a message. By contrast, in [8], N_k always sends SS_k in its entirety. In other words, N_k sends event record eR to N_i even though N_i's id is in eR's knownby set. This is particularly troublesome in the case of network partition or failed nodes, since no event record can collect all node ids in its knownby set and therefore SS will grow unboundedly. This problem does not arise in the 2DTT scheme.

A deficiency of the approach described here is that the 2DTT, of size $O(n^2)$, is sent as part of each message. By contrast, a 1-dimensional time table of size O(n) and a collection of knownby sets of size O(mn) where m is the size of SS_i and is proportional to the frequency with which events occur, are sent in a message in [8]. Therefore, the

2DTT method is applicable when one or more of the following conditions hold: events are frequent, the number of nodes is not large, and failure is possible. In the next section, strategies are presented to reduce the amount of timing information that must be sent.

6. On the Size of the 2DTT

The fact that the 2DTT has size $O(n^2)$ implies excessive storage and communication overhead. In the following, strategy 0 is the previously described algorithm while strategies 1 - 3 are modifications which deal with these problems.

Strategy 0 -- A node sends the complete 2DTT in each message and stores the complete 2DTT.

Row k of T_i is N_i's estimate of how up to date N_k is concerning events at other nodes. Upon receiving a message from N_j containing T_j, N_i first updates its own row (row i) of T_i using row j of T_j and then updates other rows with corresponding rows of T_j.

Strategy 1 -- Each node stores the complete 2DTT but sends only its own row.

In this case N_j includes only the jth row of T_j in each message it sends and N_i uses the received row to update only the ith row and the jth row of T_i. Other rows of T_i will be updated by direct messages from corresponding nodes. Row i of T_i therefore has the same contents that it would have under Strategy 0, since the way each node's own row is updated remains the same. Entries in other rows, however, may be less than they would be if Strategy 0 were used since now N_i updates a row other than row i only when a message arrives from the corresponding node. Consequently, a node may be forced to include more event records in NP in each message it sends under strategy 1 than it does under strategy 0. Thus the savings made using strategy 1 in reducing the communication cost by sending less timing information is compensated for by the need to send additional event records. The most advantageous strategy will depend on the details of the situation. The correctness of the algorithm as modified by this strategy follows from the fact that the entries of T_j are not larger than they would be if strategy 0 were used. Thus the set, NP, computed when N_j sends a message includes all event records that would be sent if strategy 0 were used.

In the preceding analysis we have not taken into consideration the fact that due to hardware or software limitations it may not be possible for a node to send a message to every other node in the network. We will represent the network by a undirected graph G in which vertices represent nodes and an edge (i,j) indicates that N_i and N_j can send messages to each other. N_i and N_j are called **neighbors** in this case. In the 2DTT technique, N_i uses row k to decide what portion of the log should be included in a message it sends to N_k. This suggests that a node not store those rows of the 2DTT which do not correspond to its neighbors.

Strategy 2 -- Each node stores only its own row and a row for each of its neighbors; it sends only those rows which correspond to neighbors of the target node.

Since the entire 2DTT is not retained, a node can no longer determine when all nodes have learned of a particular event. Instead, a node discards an event record from its log as soon as it knows that all its neighbors have learned of the event. N_i can determine this for an event which has occured at N_k (which need not be a neighbor) by examining $T_i[j,k]$ where j ranges over all neighbors. Because a node discards event records more quickly than in previous strategies, it keeps a smaller partial log. If a node fails before it learns of e then its neighbors keep eR in their partial logs until it recovers. Non-neighboring nodes discard eR.

With the concept of a neighborhood in a network, we have reduced the number of rows of the 2DTT stored at a node. However, all rows still have dimension n. The next goal is to reduce the size of a row by deleting entries in those columns which do not correspond to neighbors. A maximum complete subgraph (i.e., a subgraph in which each node is connected to every node in the subgraph) in G is called an **area**. Two nodes may directly communicate if and only if they are in the same area. A node which is in more than one area is called a **gateway**. Suppose N_i and N_j are in the same area, A, and event e occurs at a node, N_k, not in A. In strategy 2, N_i uses the entry $T_i[j,k]$ to determine whether N_j has learned of e. Thus, one cannot cut off columns which correspond to non-neighbor nodes without modifying the algorithm.

In the following, we modify the log solution so that a node stores only columns which correspond to it neighbors. The intuition underlying strategy 3 can be obtained from the example shown in Figure 1. A network is divided into two areas, A_1 and A_2. Nodes maintain timing information about other nodes in common areas. Thus N_3 cannot call upon $T_3[i,1]$ in deciding whether it should send a record, eR to a neighbor, N_i, where eR.node

= 1. To overcome this problem, a gateway in A_2 whose first knowledge of e comes from a node not in A_2 declares itself to be a representative of e in A_2. In Fig 1 N_2 is such a gateway. N_2 functions as a proxy for N_1 in A_2 with the effect that nodes in A_2 treat e as if it had occurred at N_2. Thus, in deciding whether to send eR to N_i, N_3 appeals to $T_3[i,2]$.

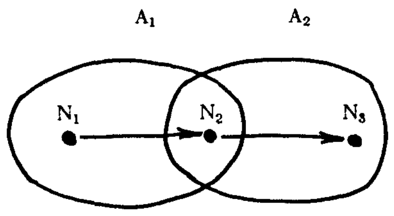

Figure 1

Strategy 3 -- Each node stores only its neighbors' rows and neighbors' columns; it sends only those rows to N_k which correspond to neighbors of N_k.

Let A_i contain n_i nodes. A node maintains a sub-time-table (STT) for each area in which it resides. Thus in Figure 1, N_2 maintains sub-time-tables $T_{2,1}$ and $T_{2,2}$ (having dimension $n_1 \times n_1$ and $n_2 \times n_2$ respectively) for areas A_1 and A_2 respectively while N_1 maintains only $T_{1,1}$. Both rows and columns of $T_{i,j}$ correspond to nodes in A_j. Note that since areas overlap, two STTs stored at a node may have rows and columns corresponding to the same nodes (further optimizations can be used to avoid this). If N_i sends a message to N_j it includes all STTs $T_{i,k}$ such that N_j is a member of A_k.

A **representative** of event e in area A is a node in A which first learns of e from a node which is not in A. A representative must be a gateway; the reverse need not be true. Note that there may be several gateways in A which concurrently learn of e from nodes not in A (see Figure 2), hence representative nodes need not be unique. Because of this lack of uniqueness the contents of an event record may vary depending on its location. Therefore we use the notation eR_i to denote the event record describing e at N_i and eR_i' to denote an event record describing e sent by N_i. Copies of eR can be recognized as describing the same event since they all contain the same unique tag, <time(e),node(e)>. Copies differ in the set of representative tuples they contain. A **representative tuple** in eR, $<A_p$,rtime,rnode> indicates that node rnode, contained in area A_p, first learned of e from a node not in A_p at time rtime as measured on rnode's clock. As a result it declared itself to be a representative of e in A_p. eR_i contains a representative tuple for each area within which N_i resides. This is achieved as follows:

(1) If e occurs at N_i then N_i inserts in eR_i a tuple $<A_j,clock_i,i>$ for each area A_j containing N_i.
(2) If N_i sends a message to N_j it includes in eR_i' only those representative tuples for areas which contain N_j as well.
(3) On receiving eR_i', if N_j does not have a record for e, it creates eR_j containing all representative tuples in eR_i' as well as additional tuples of the form $<A_k,clock_j,j>$ for each area A_k containing N_j but not N_i. eR_j is inserted in PL_j in this case. If N_j already has a record for e it ignores eR_i'.

Note that the effect of the first part of step 3 is that N_j has declared itself to be the representative of e in A_k since e was unknown prior to the message and N_i is not a member of A_k. If $T_{i,u}[j,rnode] \geq$ rtime for some area A_u containing both N_i and N_j and for some representative tuple $<A_u$,rtime,rnode> in eR_i then N_i does not send a record describing e in messages to N_j.

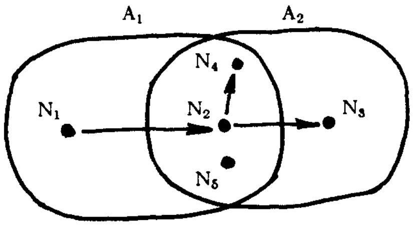

Figure 2

Consider the simple example shown in Figure 2 in which e occurs at N_1 in A_1 at time t_1 read on N_1's clock. eR_1 has one representative tuple, $<A_1,t_1,1>$. N_1 sends a message containing eR_1', where in this case $eR_1' = eR_1$, to N_2 and N_5. When N_2 receives the message, at time t_2 read on its own clock, it becomes a representative of e in A_2 since eR_1' does not have a representative tuple for that area and N_2 has not previously learned of e. N_2 will form eR_2 by adding $<A_2,t_2,2>$ to eR_1', where t_2 is the current reading of $clock_2$ and inserts the result into PL_2. When N_5 receives the message, it will do a similar thing, declaring itself to be the representative in A_2. At a send event to N_3, N_2 includes only the representative tuple for A_2 in eR_2' in the message. At the time N_3 receives the message, it knows it is not a representative and thus inserts eR_3, where $eR_3 = eR_2'$, in PL_3. N_3 will ignore eR_5' on receiving any later messages from N_5. Thus, N_3 takes N_2 as the representative in A_2. A gateway is not necessarily a representative. For example if N_4 learns of e from N_2, it discovers representative tuples for both A_1 and A_2 and therefore does not declare itself to be a representative.

In strategy 2, a node uses space O(kn) for the 2DTT, where k is the number of its neighbors. Note that k depends on the connectivity of the network and does not necessarily grow with the total number of nodes. In many cases it is a constant. In strategy 3 a node uses space $O(k^2)$ for each area to which it belongs. However strategy 3 uses extra space for representative tuples in each event record. Hence strategy 3 is particularly applicable when the network is large, nodes are connected to a small number of other nodes, and events do not occur frequently.

7. Discussion

Algorithms have been developed to maintain a replicated dictionary. The solution is suitable for such applications as distributed directories where entries represent files. Two entries with the same symbolic name, that are created by two concurrent insert events, actually represent two different files. Furthermore, a file with a symbolic name which was used by a previously deleted file is different from its predecessor. The creation time and creator id can distinguish these entries. However, there are applications (e.g. an Access Control List), in which two concurrent events may insert an identical element. Furthermore, reinserting an element x after it has been deleted is not uncommon. Thus multiple occurrences of c_t in E, are allowed and restriction (R2) should be removed. An efficient solution to this problem can be obtained by using a 2DTT and log. An algorithm and its correctness proof are presented in [13].

The log solution can be used in other applications as well. For example:

(1) Replicated numeric data with add-to and subtract-from operations. Since these two operations are commutative, copies of the data will reach the same final state.

(2) Detection of failures. Consider the case that N_i sends a request to a neighboring node N_k and does not receive a reply. To distinguish node failure from communication failure, a log is used to collect records of communication events occurring in the network. If N_i's log shows that none of N_k's neighbors have received messages from N_k for some time, then it may assume N_k has failed. Otherwise N_i knows that a communication failure has occurred. An algorithm is presented in [13].

Efficient algorithms have been proposed in this paper to maintain replicated data whose semantics do not require serializability as a consistency criterion. The efficiency gains which can be achieved when serializability is dropped make it worthwhile to search for other applications having similar properties.

Appendix

The following correctness proof is exhibited with respect to a given execution $<E,\rightarrow>$. For convenience we will use the following notation and definitions.

(N1) $e \overset{+}{\rightarrow} e'$ iff $e \rightarrow e'$ and $e \neq e'$.

(N2) $e \overset{1}{\rightarrow} e'$ iff $e \overset{+}{\rightarrow} e'$ and if node(e) = node(e') then there is no event f at node(e) such that $e \overset{+}{\rightarrow} f \overset{+}{\rightarrow} e'$, or else op(e) = send(m) and op(e') = receive(m) for the same message m.

If $e \overset{1}{\rightarrow} e'$ then e(e') is called the **immediate predecessor** (successor) of e'(e). Only a receive event has more than one immediate predecessor, it has two (except if it is the first event at a node); only a send event may have more than one immediate successor. Assuming only unicast communication it may have two.

(N3) Let e be a receive event. p_e denotes the immediate predecessor of e at node(e). q_e denotes the send event which supplies the message to e.

(N4) T(e) is the value of $T_{node(e)}$ immediately after e completes.

Lemma 1.1: Let e, f $\in$ E. If fR $\in$ L(e) then $f \rightarrow e$.
Proof : We can construct a sequence of immediate predecessors, $h_1 \overset{1}{\rightarrow} \dots \overset{1}{\rightarrow} h_m$ where $h_m = e$, such that ($\forall$ i) $1 \leq i < m$, fR $\in$ L(h_i) and for every immediate predecessor p of h_1, fR $\notin$ L(p). Since only the non-communication event, f, can insert fR in L under these circumstances, $h_1 = f$ and $f \rightarrow e$.

Lemma 1.2: Let e, f $\in$ E. If $f \rightarrow e$, then $\forall$ I, J $\in$ [n], $T(f)[I,J] \leq T(e)[I,J]$.
Proof : This lemma follows by induction on $\rightarrow$. It is true for any event which has no immediate predecessor. Suppose the result is true for all immediate predecessor(s)

of e. The algorithms for the various operations either leave T_i unchanged (send), increase $T_i[i,i]$ (oper) or apply the max function on T_i and T_k (receive). Therefore the lemma holds for e.

Lemma 1.3 : Let e $\in$ E. If T(e)[node(e),k] $>$ 0 then there exists an event, f, at N_k such that T(f)[k,k] $=$ T(e)[node(e),k] and f $\rightarrow$ e.

Proof : The lemma follows by induction on $\rightarrow$. It is true for any event, e, which has no immediate predecessor since if T(e)[node(e),k] $>$ 0 then node(e) $=$ k and e $=$ f. Suppose this result holds for all immediate predecessors of e. The result is obviously true if node(e) $=$ k. If node(e) $\neq$ k, the value of T(e)[node(e),k] is taken from T(e')[node(e'),k], where e' $\xrightarrow{1}$ e. Hence the result holds by the inductive hypothesis.

Lemma 1.4: Let e, f $\in$ E where f is a non-communication event. If T(e)[node(e),node(f)] $\geq$ time(f) then f $\rightarrow$ e.

Proof : By lemma 1.3, there exists an event e' at node(f) such that T(e')[node(f),node(f)] $=$ T(e)[node(e),node(f)] and e' $\rightarrow$ e. By C1 either f $\rightarrow$ e' or e' $\xrightarrow{+}$ f. If e' $\xrightarrow{+}$ f, then T(e')[node(f),node(f)] $<$ T(f)[node(f),node(f)] (since the clock is invoked at f) and we have the following contradiction: time(f) $\leq$ T(e)[node(e),node(f)] $=$ T(e')[node(f),node(f)] $<$ T(f)[node(f),node(f)] $=$ time(f). Therefore we conclude f $\rightarrow$ e.

Lemma 1.5: Let e, f $\in$ E. If hasrec($T_{node(e)}$,fR,k) is true at e then there exists an event, g, at N_k such that f $\rightarrow$ g $\rightarrow$ e.

Proof : Since, by lemma 1.2, each entry of the 2DTT is monotone over $\rightarrow$, we can construct a sequence of immediate predecessors $h_1 \xrightarrow{1} \ldots \xrightarrow{1} h_m$ where h_m = e, such that ($\forall$ i) 1 $\leq$ i $<$ m, T(h_i)[k,node(f)] $=$ T(e)[k,node(f)] and for every immediate predecessor, p, of h_1, T(p)[k,node(f)] $<$ T(h_1)[k,node(f)]. In order to increase an entry in the kth row of $T_{node(h_1)}$, h_1 must satisfy the following. (1) h_1 is a receive event - In this case (S2) can increase the entry only if node(h_1) $=$ k. (2) h_1 is a non-communication event - In this case the entry is increased by a clock invocation and node(h_1) $=$ node(f) $=$ k. Therefore in both cases node(h_1) $=$ k. Since T(h_1)[node(h_1),node(f)] $\geq$ time(f), f $\rightarrow$ h_1 by lemma 1.4. Therefore f $\rightarrow$ h_1 $\rightarrow$ e and node(h_1) $=$ k.

Theorem 1 : Let e, f be 2 events in an execution $<$E,$\rightarrow$$>$ where f is a non-communication event. fR $\in$ L(e) iff f $\rightarrow$ e.

Proof:

($\Rightarrow$) From lemma 1.1.

($\Leftarrow$) This is proven by induction on $\rightarrow$.

Basis. If e has no immediate predecessor then f $=$ e and the result follows from the oper algorithm.

Induction.

Case 1. e $=$ f. From the oper algorithm, fR $\in$ L(e).

Case 2. e $\neq$ f. 2 cases are considered.

Case 2.1. e is a send event or a non-communication event. e has a unique immediate predecessor, g, and f $\rightarrow$ g. Then fR $\in$ L(g) by the inductive hypothesis and L(g) $\subset$ L(e) in this case.

Case 2.2. e is a receive event. Either f $\rightarrow$ p_e or f $\rightarrow$ q_e, or both. If f $\rightarrow$ p_e, then the result follows using the argument of case 2.1. The final case to consider is that f $\rightarrow$ q_e and f $\not\rightarrow$ p_e. By hypothesis, fR $\in$ L(q_e). Let $<$NP,T$>$ be the message sent from q_e to e. Assume fR $\notin$ NP. From the send algorithm, hasrec($T_{node(q_e)}$,fR,node(e)) is true at q_e. By lemma 1.5, there exists an event, h, at node(e) such that f $\rightarrow$ h $\rightarrow$ q_e. From (C1), either h $\xrightarrow{+}$ e or e $\rightarrow$ h. The former cannot be true since f $\rightarrow$ h $\xrightarrow{+}$ e contradicts the assumption that f $\not\rightarrow$ p_e. If e $\rightarrow$ h, we have $q_e \xrightarrow{1}$ e $\rightarrow$ h $\rightarrow$ q_e, which is also a contradiction. Hence fR $\in$ NP. Therefore fR $\in$ L(e).

Reference

[1] M.J. Fischer and A. Michael, "Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network", *ACM SIGACT-SIGMOD Symposium on Principle of Database Systems,* 1982, pp. 70-75.

[2] H. Garcia-Molina, "Using Semantic Knowledge for Transaction Processing in a Distributed Database", *ACM Transaction on Database Systems,* June 1983, Vol. 8, No. 2, pp. 186-213.

[3] L. Sha, E. Jensen, R. Rashid and J. Northcutt, "Distributed Co-operating Processes and Transactions", *ACM SIGCOMM,* March 1983, pp. 188-196.

[4] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database systems", *Computing Surveys,* Vol. 13, No. 2, June 1981, pp. 185-221.

[5] D. Dolev and R. Strong, "Distributed Commit with Bounded Waiting", *IEEE Symposium on Reliability in Distributed Software and Database Systems,* 1982.

[6] R.H. Thomas, "A Majority Consensus Approach to the Concurrency Control Problem for Multiple Copy Databases", *ACM TODS,* June 1979, pp 180-209.

[7] L. Lamport, "Time, Clocks, and the Ordering of events in a Distributed System", *Communication of the ACM,* July 1978, pp 558-565.

[8] J.E. Allchin, "An Architecture for Reliable Decentralized Systems", Ph.D. Thesis. Georgia Institute of Tech., Technical Report GIT-ICS-83/23.

[9] F.B. Schneider, "Synchronization in distributed programs", *ACM TOPLAS,* April 1982, pp 1-24.

[10] B.W. Lampson, "Atomic Transactions", In *Distributed Systems - Architecture and Implementation,* Lecture Notes in Computer Science, vol. 105, Springer-Verlag, 1981.

[11] D.K. Gifford, "Weighted Voting for Replicated Data", *ACM SIGOPS Symposium,* 1979, pp 150-159.

[12] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem", *ACM TOPLAS,* April 1982, pp 382-401.

[13] G.T. Wuu and A.J. Bernstein, "Efficient Solutions to the Replicated Log and Dictionary Problems", Technical Report 84/077, May 1984.