AN EFFICIENT, FAULT-TOLERANT PROTOCOL FOR REPLICATED DATA MANAGEMENT

*Amr El Abbadi**
Computer Science Department
Cornell University

Dale Skeen
IBM Research Laboratory
San Jose

Flaviu Cristian
IBM Research Laboratory
San Jose

ABSTRACT

The objective of data replication is to increase data availability in the presence of processor and link failures and to decrease data retrieval costs by reading local or close copies of data. Moreover, concurrent execution of transactions on replicated data bases must be equivalent to the serial execution of the same transactions on non-replicated databases.

We present a pedagogical derivation of a replicated data management protocol which meets the above requirements. The protocol tolerates any number of component omission and performance failures (even when these lead to network partitioning), and handles any number of (possibly simultaneous) processor and link recoveries. It implements the reading of a logical object efficiently--by reading the nearest, available copy. When reads outnumber writes and failures are rare, the protocol performs better than other known protocols.

1. INTRODUCTION

The objective of data replication in a distributed database system is to increase data availability and decrease data access time. By data replication, we mean maintaining several *physical* copies, usually at distinct locations, of a single *logical* data base object. A *replica control* protocol is responsible for coordinating physical accesses to the copies of a logical data object so that they behave like a *single* copy insofar as users can tell [BGb]. Such a protocol translates a logical write of a data object x into a set of physical writes on copies of x, and translates a logical read of x into a set of reads on one or more physical copies of x. To really increase data availability, a replica control protocol must be *tolerant* of commonly occurring system component *failures*. To minimize the overhead caused by replication, the protocol should minimize the number of physical accesses required for implementing one logical access.

This paper presents a replica control protocol tolerant of a large class of failures, including: processor and communication link crashes, partitioning of the communications network, lost messages, and slow responding processors and communication links. Any number of (possibly simultaneous) processor and link recoveries is also handled. The major strength of the protocol is that it implements the reading of a logical object very *efficiently*: a read of a logical object, when permitted, is accomplished by accessing only the nearest, available physical copy of the object. Since reads outnumber writes in most applications, this strategy is expected to reduce the total cost of accessing replicated data objects.

Our protocol belongs to a class of protocols that *adapt* to detected failures and recoveries. An integral part of the protocol is a subprotocol that maintains at each processor in the system an approximate view of the current communication topology. This view can be used to optimize the transla-

---

*This author's work was partially supported by a grant from the Sperry Corporation.

tion of logical data accesses performed by transactions into physical data accesses.

The protocol compares favorably with other proposed replica control protocols. It tolerates the same fault classes as majority voting [T] and quorum consensus [G], and does so with fewer accesses to copies, assuming that read requests outnumber write requests and that fault occurrences are rare events. It also tolerates the same fault classes as the "missing write" protocol [ES], but, unlike that protocol, uses a "read-one" rule for reading logical data objects even in the presence of failures. Our protocol is also simpler than the "missing write" protocol. In particular, it does not require the extra logging of transaction information that is required by that protocol when failures occur.

2. FAILURE ASSUMPTIONS

System components (processors, links) can fail in many ways, from occasional processor crashes and lost messages to Byzantine failures, where components may act in arbitrary, even malicious, ways. We wish to consider those component failures that have a reasonable chance of occurring in practical systems and that can be handled by algorithms of moderate complexity and cost. The most general failure classes satisfying this criteria are *omission failures* and *performance failures* [CASD].

An *omission failure* occurs when a component never responds to a service request. Typical examples of such failures include processor crashes, occasional message losses due to transmission errors or overflowing buffers, and communication link crashes. A *performance failure* occurs when a system component fails to respond to a service request within the time limit specified for the delivery of that service. Occasional message delays caused by overloaded processors and network congestion are examples of performance faults. An important subclass of omission and performance failures is the class of *partition failures*. A partition failure divides a system into two or more disjoint sets of processors, where no member of one set can communicate in a timely manner with a member of another set. Our objective is to design a replica control protocol that is tolerant of *any number* of omission and performance failures.

3. SYSTEM MODEL

A distributed system consists of a finite set of processors, P={1,2,...,n}, connected by a communication network. In the absence of failures the network provides the service of routing messages between any two processors. Processors or links may fail, leading to an inability to communicate within reasonable delays. Failed processors and links can recover spontaneously or because of system maintenance. Thus, the system of processors that can communicate with each other is a *dynamically evolving system.*

In the following discussion, we will not be concerned with the details of the physical interconnection of the processors (e.g. a point-to-point versus a bus-oriented interconnection) or with the detailed behavior of the message routing algorithm. Instead, we need only consider whether two processors are capable of communicating through messages. We model the current *can-communicate* relation between processors by a *communication graph.* The nodes of the graph represent processors, and a undirected edge between two nodes a,b$\in$P indicates that if a and b send messages to each other, these are received within a specified time limit. We call a connected component of a communication graph a *communication cluster.* A *communication clique* is a communication cluster which is *totally connected,* that is, there is an edge in the communication graph between every pair of processors in the cluster. In the absence of failures, a communication graph is a single clique. The crash of a processor p results in a graph that contains a trivial cluster consisting of the single node p. A partition failure results in a graph containing two or more clusters.

We do not assume that the can-communicate relation is transitive. Thus, it is possible that a and b can communicate, and b and c can communicate, but a and c can*not* communicate. (Note that if the can-communicate relation is transitive, then all communications clusters are cliques.) In a system in which failure occurrences lead quickly to the establishment of new communication routes, which avoid the failed system components, communication clusters can be expected to be cliques most of the time.

For the purpose of adapting to changes in the communication topology, each processor maintains a

local "view" of the can-communicate relation. Each processor's *view* is that processor's current *estimate* of the set of processors with which it believes that communication is possible. The function

$$\text{view: } P \rightarrow \mathcal{P}(P)$$

(where $\mathcal{P}(P)$ denotes the powerset of P) gives the current view of each processor $p \in P$.

A replicated database consists of a set of *logical data objects* L. Each logical object $l \in L$ is implemented by a nonempty set of physical data objects (the *copies* of l) that are stored at different processors. The copy of l stored at processor p is denoted by l_p. The function

$$\text{copies: } L \rightarrow \mathcal{P}(P)$$

gives for each logical object l the set of processors that possess physical copies of l.

Transactions issue read and write operations on logical objects. A *replicated data management protocol* is responsible for implementing logical operations (as they occur in transactions) in terms of physical operations on copies. For a protocol to be correct, the database system must exhibit the same externally observable behavior as a system executing transactions serially in a nonreplicated database system [TGGL]. This property is known as *one-copy serializability* [BGb].

One popular approach for designing a replicated data management protocol is to decompose the algorithm into two parts: a *replica control* protocol that translates each logical operation into one or more physical operations, and a *concurrency control* protocol that synchronizes the execution of physical operations [BGb]. The concurrency control protocol ensures that an execution of the translated transactions (in which logical access operations are replaced by physical operations) is serializable, that is, equivalent to some serial execution. But the concurrency control protocol does not ensure one-copy serializability, since it knows nothing about logical objects. (It may, for example, permit two distinct transactions to update in parallel different copies of the same logical object.) Given this, the replica control protocol ensures that transaction execution is one-copy serializable.

The term *event* is used to denote a primitive atomic action in the system. Among other things, we consider the reading and writing of physical objects and the sending and receiving of messages to be events. An *execution* of a set of transactions is finite set of events partially ordered by the *happens-before* relation studied in [L]. We assume that the set of events restricted to a given processor is totally ordered by the happens-before relation. That is to say, if e and f are events occurring at the same processor, then either e happens-before f or f happens-before e. Consequently, the operations executed on a given physical object are totally ordered. An execution is *serial* if its set of events is totally ordered by the happens-before relation, and if, for every pair of transactions T_1 and T_2, either all physical data operations of T_1 happen-before all physical operations of T_2, or vice versa.

4. REPLICA CONTROL

Following the decomposition outlined above, we now derive a protocol for correctly managing replicated data in the presence of any number of omission and performance failures. In this section, the emphasis is on giving the properties a replica control protocol should possess and on showing that any implementation exhibiting these properties satisfies our correctness criteria. In the next section, we describe in some detail one protocol and show that it exhibits the desired properties.

Ideally, we would like to design a replica control protocol that can be combined with any correct concurrency control protocol. However, this seems to be difficult to achieve given our performance objectives. Consequently, we will restrict the class of allowable concurrency control protocols to those ensuring a stronger property known as conflict-preserving serializability [H]. Two physical operations *conflict* if they operate on the same physical object and at least one of them is a write. An execution E of a set of transactions T is *conflict-preserving (CP) serializable* if there exists an equivalent serial execution E_S of T that preserves the order of execution of conflicting operations (i.e. if op_1 and op_2 are conflicting physical operations and op_1 happens-before op_2 in E, then op_1 happens-before op_2 in E_S) [H]. Henceforth in our discussion, we will assume the existence of a concurrency control protocol ensuring that

(A1) The execution of any set of transactions (viewed as a set of physical operations) is conflict-preserving serializable.

Practically speaking, restricting the class of concurrency control protocols to those enforcing CP-serializability is inconsequential, since all published, general-purpose concurrency control protocols are members of this class. This includes two-phase locking [EGLT], optimistic concurrency control [KR], timestamp ordering [BSR], and all distributed protocols surveyed by Bernstein and Goodman [BGa].

Our performance objective is to provide cheap read access while offering a high level of data availability. In order to understand better what is attainable, let us first consider a "clean" failure environment in which two simplifying assumptions hold. The first assumption is that the can-communicate relation is transitive:

(A2) All communication clusters are cliques.

The second assumption (unrealistically) posits that changes in the communication topology (resulting from failures and recoveries) are instantly detected by all affected processors.

(A3) The view of each processor contains only itself and processors adjacent to it in the current communication graph.

Thus, from A2 and A3 we can conclude that the views of processors in the same communication cluster are *equal* and the views of processors in different clusters are *disjoint*.

Given the above assumptions, the following rules can be used to control access to logical objects. When processor p executes an operation (either a read or a write) on a logical object l, it first checks whether a (possibly weighted) majority of the copies of l reside on processors in its local view. If not, it aborts the operation. Otherwise, for a read, it reads the nearest copy which resides on a processor in its view; and for a write, it writes all copies on processors in its view.

When integrated with an appropriate cluster initialization protocol, which ensures that all copies of a logical object accessible in a newly established cluster have the most up-to-date value assigned to that object, the above rules can form the basis of a correct replica control protocol. The "majority rule" ensures that only one cluster can access a logical object at a time, and the "read-one/write-all rule" ensures that the copies of an object in a cluster act as a single copy. Together, these rules ensure that all executions are one-copy serializable.

The above rules are simple, intuitive, and ensure a high level of data availability, provided the communication information maintained by the processors is *accurate*. Unfortunately, the correctness of the rules depends heavily on assumptions A2 and A3. If either is relaxed, non-one-copy-serializable executions can result.

Example 1. Figure 1 gives a possible communication graph for three processors when assumption (A2) is relaxed. The graph indicates that processors A and B are no longer able to communicate due to, for example, failures that have occurred during a run of the algorithm that establishes the routing between A and B. Both processors however are able to communicate with C, and C with them.

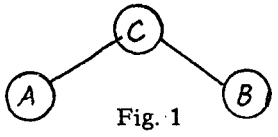

Fig. 1

We thus have: view(A)={A,C}, view(B)={B,C} and view(C)={A,B,C}. Let each processor contain a copy of a logical data object x initialized to 0. Assuming that all copies are weighted equally, each processor will consider x to be accessible, since each has a majority of the copies in its view. Now, let A and then B execute a transaction that increments x by 1. Based on its own view, processor A reads its local copy of x and updates both C's copy and its own. Similarly, B reads its local copy of x (which still contains 0) and updates both C's copy and its own. Observe that after two successive increments, all copies of x contain 1. Clearly, the execution of these transactions is not one-copy serializable. □

Example 2. Consider an initially partitioned system that undergoes re-partitioning as shown in Figure 2.

Two processors (B and D) detect the occurrence of the new partition immediately and update their

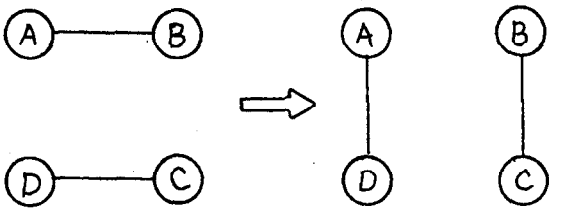

Figure 2.

views. The other two processors (A and C) do not detect it until later. Table 1 shows the intermediate system state after the view updates in B and D and before the updates in A and C.

	old view	new view
A	A,B	A,B
B	A,B	B,C
C	C,D	C,D
D	C,D	A,D

TABLE 1

Assume that, while the views are inconsistent, each processor p executes a transaction T_p. Table 2 gives the transaction executed at each processor, and also the data objects stored there. The superscripts on the objects denote "weights".

	copies	transactions
A	a^2,b	T_A: read(b),write(a)
B	b^2,c	T_B: read(c),write(b)
C	c^2,d	T_C: read(d),write(c)
D	d^2,a	T_D: read(a),write(d)

TABLE 2

Consider the execution of T_A at processor A. Since $B \in view(A)$, A can read its local copy of b. Since A's copy of a has weight 2, A can update it, and, furthermore, A will not attempt to update D's copy since $D \notin view(A)$. Hence, in the execution of the transaction, A accesses its local copies only. The execution of transactions T_B, T_C, and T_D proceed similarly, with each process modifying only its local copies. The result is serializable but not one-copy serializable. □

As example 1 illustrates, the correctness of the simple replica control protocol critically depends on the property that no two processors with different views be able to access a common set of copies. Example 2 illustrates that even in a well-behaved communication network, where transitivity of the can-communicate relation is assured, processors can not independently and asynchronously update their views.

The principal idea in our replica control protocol is to use the majority and the read/write rules mentioned above, but to circumvent the anomalies illustrated in examples 1 and 2 by placing appropriate restrictions on *when* and *how* processors may update their views. Toward this goal, we introduce the notion of a virtual partition. Roughly speaking, a *virtual partition* is a set of communicating processors that have agreed on a common view and on a common way to test for membership in the partition. For the purposes of transaction processing, only processors that are assigned to the same virtual partition may communicate. Hence, a virtual partition can be considered a type of "abstract" communication cluster where processors join and depart in a disciplined manner and communication is limited by mutual consent. In contrast, in a real communication cluster processors join and depart abruptly (and often inopportunely) because of failures and recoveries. It is desirable, of course, for virtual partitions to approximate the real communication capabilities of a system.

The common view of the members in a virtual partition represents a shared estimation of the set of processors with which communication is believed possible. When a processor detects an inconsistency between its view and the can-communicate relation (by not receiving an expected message or by receiving a message from a processor not in its view), it can unilaterally depart from its current virtual partition. (Since the departing processor may no longer be able to communicate with the other members of its virtual partition, it should be able to depart autonomously, without communicating with any other processor.) After departing, the processor can invoke a protocol to establish a new virtual partition. This protocol, which is part of the replica control protocol, creates a new virtual partition, assigns a set of processors to the partition, and updates those processors' views. An objective of the protocol is for the new virtual partition to correspond to a maximal set of communicating processors. However, since failures and recoveries can occur during

the execution of the view update protocol, it is possible that a virtual partition resulting from a protocol execution only partially achieves this objective.

We identify virtual partitions by unique identifiers, and we denote the set of virtual partition identifiers by V. At any time, a processor is assigned to at most one virtual partition. The instantaneous assignment of processors to virtual partitions is given by the partial function

$$vp: P \rightarrow V,$$

where vp is not defined for a processor p if p is not assigned to any virtual partition. We use the total function

$$defview: P \rightarrow \{true,false\}$$

to characterize the domain of vp. That is, defview(p) is true if p is currently assigned to some virtual partition, and is false otherwise.

In order to ensure that the simple "read-one/write-all" rules achieve one-copy serializability, we require the following properties from any protocol managing processor's views and their assignment to virtual partitions. Letting p and q denote arbitrary processors, the first two properties are

(S1) View consistency: If defview(p)&defview(q) and vp(p)=vp(q), then view(p)=view(q).

(S2) Reflexivity: If defview(p) then $p \in$ view(p)

Property S1 states the requirement that processors assigned to the same virtual partition have the same view. With a slight abuse of notation, we let view(v) denote the view common to all members of virtual partition v. Property S2 enforces the requirement that every processor should be able to communicate with itself. From S1 and S2, one can infer that the view of a processor (when defined) is a superset of the processors in its virtual partition, and thereby, a superset of the processors with which it *may* communicate in order to process transactions.

The final property restricts the way a processor may join a new virtual partition. Let p denote any processor and let v and w denote arbitrary virtual partitions. Let *join(p,v)* denote the event where p changes its local state to indicate that it is currently assigned to v. Similarly, let *depart(p,v)* denote the event of p changing its local state to indicate that it is no longer assigned to v. Join and departure events, in addition to physical read and write events, are recorded in the execution of transactions. The function

$$members: V \rightarrow \mathcal{P}(P)$$

yields for each virtual partition v the set of processors that were at some point in their past (but not necessarily contemporaneously) assigned to v. The third property is

(S3) Serializability of virtual partitions: For any execution E produced by the replicated data management protocol, the set of virtual partition identifiers occurring in E can be totally ordered by a relation $<<$ which satisfies the condition:

if $v<<w$ and $p \in$(members(v) $\cap$ view(w)), then depart(p,v) happens-before join(q,w) for any $q \in$ members(w).

Loosely speaking, this property says that before any processor may join a new partition w, every processor p in view(w) must first depart from its current virtual partition. Note that the property does not require p to eventually join w. S3 prevents anomalies of the type illustrated in Example 2, where a processor (B) detects changes in the can-communicate relation, adopts a new view, and begins processing transactions before another processor (C) in the same communication cluster detects the changes. Any ordering over the virtual partitions occurring in a given execution that satisfies condition S3 is called a legal *creation order* for the virtual partitions. In general, there will be many legal creation orders for a given execution.

Comparing requirements S1-S3 to requirements A2-A3, we find that the former ones are considerably weaker than the latter. A3 requires that processors' views reflect the current can-communicate relation. S1-S3, on the other hand, do not imply any relationship between the processors' views and the can-communicate relation. A3 implies that a processor will always know the members of its communication cluster. In general, a processor will never know the set of processors in its virtual partition (but it will know a superset--the processors in its view). Under S1-S3, the views of processors in different virtual partitions may overlap; whereas, under A2, the views of processors in different clusters must be disjoint. Finally and most

significantly, requirements S1-S3 are attainable under realistic failure assumptions (see §5). Requirements A2-A3 are not.

The intuitive replica control rules mentioned previously for the "clean" failure environment can now be reformulated in terms of the weaker notion of a *virtual partition*. The first four rules are

(R1) Majority rule. A logical object l is *accessible* from a processor p assigned to a virtual partition only if a (possibly weighted) majority of copies of l reside on processors in view(p).

(R2) Read rule. Processor p implements the logical read of l by checking if l is accessible from it and, if so, sending a physical read request to any processor $q \in view(p) \cap copy(l)$. (If q does not respond, then the physical read can be retried at another processor or the logical read can be aborted.)

(R3) Write rule. Processor p implements the logical write of l by checking if l is accessible from it and, if so, sending physical write requests to all processors $q \in view(p) \cap copies(l)$ which are accessible and have copies of l. (If any physical write request can not be honored, the logical write is aborted).

(R4) All physical operations carried out on account of a transaction t must be executed by processors of the same virtual partition v. In this case we say that t *executes in* v.

Rules R1-R3 are straightforward interpretations of the simple replica control rules. Rule R4 expresses the communication restriction that is placed on virtual partitions. In particular, a processor p accepts a physical access request from processor q only if p and q are assigned to the same virtual partition (that is, if defview(p) and vp(p) = vp(q)). Observe that R4 requires a transaction to be aborted if any of the processors executing it joins a new partition. (In §6, we will discuss an optimization that avoids this most of the time.)

The final rule concerns propagating the up-to-date values of logical objects to copies on processors that were previously in different partitions.

(R5) Partition initialization rule. Let p be a processor that has joined a new virtual partition v, and let l_p be a copy of a logical object l that is accessible in v. The first operation on l_p must be either a write of l_p performed on behalf of a transaction executing in v, or a *recover(l_p)* operation that writes into l_p the *most recent* value of l written by a transaction executing in some virtual partition u such that $u << v$ for any legal creation order $<<$.

Stated less formally, R5 says that before copy l_p can be read in partition v, it must contain the most recent value assigned to l in the current partition or in a previous partition. Note that the desired value of l can be found at one of the processors in view(v) because a majority of l's copies must reside at processors in view(v) for l to be accessible in v (by R1), and a majority of l's copies is written by each logical write of l (by R3).

The above properties can be used to design a correct replica control protocol.

Theorem 1. Let R be a replica control protocol obeying properties S1-S3 and R1-R5 and let C be a concurrency control protocols that ensures CP-serializability of physical operations. Any execution of transactions produced by R and C is one-copy serializability.

The proof of this theorem, which uses notions introduced in [BGb], is given in [ESC]. The proof actually proves a stronger result.

Theorem 1'. Let R and C be protocols as above. For every execution E produced by R and C, and every legal creation order $<<$ over the virtual partitions in E, there exists a serial execution E_S, equivalent to E, with the property: *if $v << w$, then all transactions executing in v occur in E_S before any transaction executing in w.*

Hence, with regard to serializability, we can consider transactions to execute in an order consistent with a creation order of virtual partitions. That is to say, for an execution with a creation order $<<$, if transaction t_1 executes in v and transaction t_2 executes in w and $v << w$, then we can consider t_1 to "execute before" t_2.

Although a replica control protocol satisfying S1-S3 and R1-R5 produces only one-copy serializable executions, the executions may exhibit a curious and, for some applications, undesirable property. Specifically, transactions may not read the most up-to-date (in real time) copies of logical objects. This can occur because views of different virtual partitions that exist simultaneously can overlap. Consequently, the same logical object can be simultaneously accessible in these partitions. Note that only one of these partitions will be able to write the object, since a logical write requires a majority of the copies to reside on processors actually in the same partition; however, the other partitions will be able to read the object and, therefore, will be able to read out of date values. This phenomena is not detectable by applications executing transactions since, by design, applications can not send messages across partition boundaries. However, this could be detected by a user that moves from a processor in one partition to a processor in another.

The capability of a processor to read stale data indicates that the processor's view no longer accurately reflects the can-communicate relation. This situation arises most often when a processor is slow to detect the occurrence of a failure. For most systems, such situations can be expected to be short-lived. If desired, a distributed systems can periodically send probe messages in order to bound the staleness of the data (the next section discusses probing in more detail). Nonetheless, there appears to be no practical way to completely eliminate the reading of stale data when using the majority and read/write rules.

5. THE REPLICA CONTROL PROTOCOL

We now describe a replica control protocol satisfying the properties presented in the previous section. In the design of the protocol we have assumed that the can-communicate relation will be transitive most of the time, although the correctness of the protocol does not depend on this. We have also chosen to emphasize clarity over performance. Consequently, for any real implementation, a number of significant optimizations are possible. We discuss some of these in the next section.

The replica control protocol consists of two simpler protocols:

1) the *virtual partition management* protocol, which assigns processors to virtual partitions and ensures that all copies of a logical object which are accessible in a newly formed partition have the most up-to-date values,

2) the *logical read/write* protocol that controls logical object accessibility and translates the logical operations issued by transactions into physical read/write operations.

Requirements S1-S3, which constrain the behavior of a virtual partition management protocol, and rules R1-R5 are sufficient for ensuring one-copy serializability. Nevertheless S1-S3 is not by itself a satisfactory specification for a useful virtual partition management protocol, since it does not require that the assignment of processors to virtual partitions mirror the current communication capabilities of a system. In fact, a trivial protocol that never assigns any processor to a virtual partition, or that constantly assigns each processor to its own virtual partition satisfies the specification S1-S3. Clearly, the availability of logical objects is influenced by how closely the views of the processors of a virtual partition mirror the current can-communicate relation. If a processor excludes from its view a processor with which it can communicate, then some logical objects will unnecessarily be deemed inaccessible by rule R1. If, on the hand, a processor p includes in its view a processor q with which it can not communicate, then p will not be be able to write logical objects with copies on q (because of the write-all rule).

To ensure high logical data availability, we introduce a supplementary *liveness* constraint on the discrepancy between processor views and reality. Let us say that a failure or recovery *affects* a processor if it causes in the communication graph the deletion or addition of an edge that is incident to the processor. A failure or recovery affects a set of processors if it affects any processor in the set. The liveness constraint is

(L1) Let C be a nontrivial clique in the communication graph existing at time t. There exists a constant Δ such that, if no failures or recoveries affecting C occur during the time interval $[t,t+\Delta]$, then at $t+\Delta$ the view of each processor in C contains all processors in C.

The replica control protocol is implemented at each processor by several concurrent *tasks* that communicate through shared variables. Figure 3 gives the main task, which declares the shared variables and schedules the tasks of the virtual partition management protocol and the read/write protocol, in that order.

```
1  main task Replica-Control-Protocol;

2  type vp-id = record   n: integer;   p: P;  end;

3  shared var   cur-id, max-id: vp-id init (0,myid);
4       assigned: Boolean init true;
5       lview: set of P init {myid};
6       locked: set of L  init {};
7       local: set of L;

8  schedule(Monitor-VP-Creations);
9  schedule(Send-Probes,Monitor-Probes);
10 schedule(Physical-Access);
11 schedule(Logical-Read,Logical-Write);
```

Figure 3.

A virtual partition identifier (a "vp-id"; see line 2) consists of a sequence number and a processor identifier. Virtual partition identifiers are totally ordered by the relation "$\prec$":

$$id_1 \prec id_2 \equiv (id_1.n < id_2.n) \vee (id_1.n = id_2.n) \& (id_1.p < id_2.p).$$

The shared variable "assigned" is true on a processor p if p is currently assigned to a virtual partition, and is false otherwise. (In what follows, "$assigned_p$" denotes the "assigned" variable on processor p.) The variable "$cur\text{-}id_p$" stores the identifier of the virtual partition to which p was last assigned. Thus, when "$assigned_p$" is true, "$cur\text{-}id_p$" contains the identifier of the virtual partition to which p *is* assigned. The "$max\text{-}id_p$" variable records the largest partition identifier seen so far by processor p, and "$lview_p$" records the local view of p. The variable "locked" contains the set of logical objects whose physical copies must be assigned the same (most up-to-date) values after a new virtual partition is created. If a logical object $l \in L$ is in $locked_p$, l cannot be accessed on processor p. The variable "$local_p$" contains the set of objects that have copies at p. The relation between the concrete variables "assigned," "lview," and "cur-id" and the abstract functions "defview", "view" and "vp" introduced in §4 is:

$$defview(p) = assigned_p$$
$$assigned_p \Rightarrow (vp(p) = cur\text{-}id_p) \;\&\; (view(p) = lview_p).$$

The meaning of the "schedule(T_1,...,T_k)" primitive is "for every $T \in \{T_1,...,T_k\}$, if task T is currently inactive start an execution of T, otherwise do nothing."

The virtual partition management protocol, the first of the two major components of the replica control protocol, enforces the liveness constraint (L1), the restrictions S1 through S3, and the partition initialization rule (R5). The timely detection of changes in communication capability, needed to enforce the liveness constraint, is accomplished through periodic probe messages. When a change in communication capability is detected, the creation of a new virtual partition is attempted.

The process of virtual partition creation requires three phases. In the first phase, a processor that detects the change in communication capabilities, say s, computes an identifier v for a new virtual partition that it will attempt to establish. The identifier v is greater than all sequence numbers seen so far by s, and is guaranteed to be globally unique. Processor s then sends an invitation to join the new virtual partition identified v to all processors in P. A processor q accepts the invitation only if it has not already received another invitation to join a higher numbered partition. After accepting or initiating an invitation and before committing itself to a virtual partition, a processor is not assigned to any virtual partition.

In the second phase, the initiator s defines the view of the new partition v as being the set A of all accepting processors from which it has heard, and it sends the new view to all processors in A. Upon receiving the new view A, an accepting processor q assigns itself to the virtual partition v (and updates its view to A) only if it has not, in the meantime, accepted an invitation to join a higher numbered virtual partition. Hence, at the end of the second phase, only a subset of the processors in A might actually be assigned to the virtual partition v they have accepted to join at an earlier moment.

The third phase starts after a processor is assigned to a new virtual partition. The purpose of this phase is to ensure that all physical copies of a logical object accessible in a newly formed virtual partition have the most up-to-date value before they are made accessible to transactions.

Observe that the above process handles the case where several processors in the same cluster simultaneously attempt to establish new partitions. In the absence of additional failures, only the processor generating the highest numbered partition identifier will succeed.

Let us now examine in some detail the tasks implementing virtual partition creation. The process is initiated by calling a procedure "Create-new-VP" (Figure 4). This procedure first checks to see if the processor knows of an ongoing virtual partition creation (line 2). If so (i.e., "assigned" equals false), then partition creation is not attempted. Otherwise, the procedure departs from the current virtual partition (line 3), computes a new virtual partition identifier, composed from the successor of the largest sequence number seen so far and the local processor name obtained by invoking the predefined function "myid" (line 4), and then schedules the task "Create-VP" (line 5), which actually controls the execution of the first two phases of the virtual partition management protocol. The symbols "<" and ">" are used to delimit critical sections, which provide mutual exclusion on accesses to shared variables.

```
1  procedure Create-new-VP;
2  <if assigned
3  then assigned←false;
4       max-id←(max-id.n+1,myid);
5       schedule(Create-VP(max-id))
6  fi;>
```

Figure 4.

The "Create-VP" task (Figure 5) sends an invitation to join the partition identified "new-id" to all processors (line 3) and records in "A" the identity of all processors that accept to join the new virtual partition (lines 4-13). The δ parameter in line 5 is an upper bound on the message transmission delay between any two processors. After waiting long enough to receive all responses, and after verifying that no invitation to join a higher numbered virtual partition has been received in the meantime (line 14), a commit message is sent to all accepting processors (lines 15-17), the variable "locked" is initialized to the set of all logical objects accessible in view "A" and which have local copies, the task "Update-Copies-in-View" is scheduled, and "Create-VP" ends. The Boolean function (line 18)

accessible: L × $\mathcal{P}$(P) → {true,false},

is true for some logical object $l \in L$ and subset $A \subseteq P$ if a (weighted) majority of copies of l reside on processors in A.

```
1  task Create-VP(in new-id: vp-id);
2  var A: set of P; S: Timer; r: P; v: vp-id;
3  for each p∈P-{myid} do
4     send(p,"newvp",new-id);
5  S.set(2δ); A←{myid};
6  cycle
7   select from
8    receive("OK",v,r) → if v=new-id
9                        then A←A∪{r} fi;
10   □
11   S.timeout        → exit loop;
12  end select
13 endcycle;
14 <if new-id=max-id
15  then cur-id←max-id; lview←A; assigned←true;
16     for each p∈P-{myid} do
17          send(p,"commit",cur-id,A);
18     locked←{l | l∈L&accessible(l,lview)&l∈local};
19     schedule(Update-Copies-in-View);
20 fi;>
```

Figure 5.

The task responsible for generating responses to invitations to join new virtual partitions is "Monitor-VP-Creations" (Figure 6). This task accepts invitations to join new virtual partitions which are higher numbered than the greatest virtual partition identifier ever seen (lines 5-10). In order to avoid an indefinite waiting for a commit message, a timer "T" is set for a duration of 3δ, sufficient for the initiator to compute a view (2δ) and send it to all accepting processors (δ). After accepting an invitation to join a new virtual partition "v", if no other higher numbered invitations are received and a timely commit message for the virtual partition "v" is received, a processor commits itself to "v" (lines 12-20). If no timely commit message is received

```
1  task Monitor-VP-Creations;

2  var A: set of P; T: Timer; v: vp-id;

3  cycle
4   select from
5   receive("newvp",v)   →
6              <if max-id<v
7               then max-id←v; assigned←false;
8                 send(v.p,ack,v,myid);
9                 T.set(3δ);
10             fi>
11   □
12  receive("commit",v,A) →
13             <if v=max-id
14              then cur-id←max-id; lview←A;
15                assigned←true;
16                locked←{l | l∈L& ∈local&
17                        accessible(l,lview)};
18                schedule(Update-Copies-in-View);
19                T.reset;
20             fi>
21   □
22  T.timeout            →
23             <max-id←(max-id.n+1,myid);
24              schedule(Create-VP(max-id))>
25  end select
26 endcycle;
```

Figure 6.

(either because the message notifying the acceptance to join was lost, or the initiator has failed, or the commit message was lost) an attempt to form a new virtual partition is started (lines 22-24).

The periodic probing process is implemented in two tasks. During each probe period (of length π), the "Send-probes" task (Figure 7) of a processor p sends message probes to all other processors. Processor p then waits to receive acknowledgements from all processors in its partition (lines 13-20). After the maximum roundtrip message time (2δ) has elapsed, p compares the set of acknowledging processors to its current view (line 21). Any discrepancy detection triggers the creation of a new virtual partition.

The "Monitor-Probes" task (see Figure 8) for processor p, processes the probe messages received from other processors. A probe message with a partition identifier equal to p's current identifier is acknowledged; one with an identifier less than p's is ignored, because the message may be an old message that was delayed in transmission; and one with an identifier greater than p's identifier triggers the creation of a new virtual partition, because the receipt of such a message unambiguously demonstrates the capability of communication between processors in different virtual partitions.

```
1  task Send-Probes;

2  const π: time; %period of probing%

3  var  T: timer;   R: set of P;
5     n: integer init 0;   %probe sequence no.%
6     m: integer;   q: P;

7  cycle
8    <if assigned
9     then for each p∈P-{myid}
10              do send("probe",myid,cur-id,n)>;
11       T.set(2δ);
12       R← {myid};
13        cycle
14           select from
15             receive("ack",q,m) →
16                    if m=n then R ← R∪{q}
17             □
18             T.timeout  → exit loop;
19           endselect
20        endcycle
21       <if (assigned) & (R≠lview)
22         then Create-New-VP  fi>;
23       n ← n+1;
24       wait(π-2δ);
25    else  wait(π) >
26    fi;
27 endcycle;
```

Figure 7.

```
1  task Monitor-Probes;

2  cycle receive("probe",q,v,m);
3    <if assigned
4     then if v=cur-id → send(q,"ack",myid,m);
5             □
6           v<cur-id →  skip;
7             □
8           cur-id<v →  Create-New-VP;
9          fi
10    fi>;
11 endcycle;
```

Figure 8.

The implementation given for the tasks Send-Probes, Monitor-Probes, Create-VP, and Monitor-VP-Creations satisfies the requirements S1 through S3. Satisfaction of requirement S1 is ensured from the use of a single processor, the initiator, to determine the view of a virtual partition. Satisfaction of requirement S2 is easy to verify. Satisfaction of requirement S3 follows from the fact that the relation $\prec$ defined over virtual partition identifiers is indeed a legal creation order.

The implementation also satisfies the liveness condition (L1) when the parameter Δ is set to $\pi+8\delta$. This value is computed as follows. After a clique C forms, it may take 3δ time units for all ongoing instances of "Create-VP" to finish. It may take another π time units for the processors in C to send probes, and an additional 2δ to receive the acknowledgements. The results of probing may cause additional invocations of "Create-VP", and these will take 3δ to complete. Now if no failures or recoveries affecting C occur during any of this, then within $\pi+8\delta$ time units all processors in C have committed to the invocation of "Create-VP" generating the highest virtual partition identifier.

After a processor is assigned to a new virtual partition, it has to make sure that all physical copies accessible in the new virtual partition possess the most recent value assigned to the logical object they represent (rule R5). To determine the most recent value of a logical object, we will make use of the fact that the relation $\prec$ is a legal creation order. From this and Theorem 1', it follows that the most recent write operation on a logical object l is executed in the highest numbered virtual partition among those partitions containing logical writes to l.

For the purpose of identifying the value produced by the most recent write of l, each processor stores with its local copy of l the virtual partition identifier associated with the latest logical write of l. On each processor p, the partial functions (suitably initialized)

value: L → Value,
date: L → V,

yield the value and the most recent assignment date for the logical objects stored on p. Thus, value(l) denotes the value of the local copy of the logical object l and date(l) denotes the virtual partition identifier (or logical date) current when the local copy of l was last updated.

```
1  task Update-Copies-in-View;
2  var old-id: vp-id;
3  old-id←cur-id;
4  for each l∈locked
5  cobegin
6     var R: set of P; val: Value; d: vp-id init (0,0);
7     <R←copies(l)∩lview>;
8     for each p∈R
9     cobegin
10       var d_p: vp-id;  v_p: Value;
11       send(p,"read",l,cur-id);
12       receive(p,l,v_p,d_p)
           [no-response: Create-new-VP; exit task];
13       if d≺d_p then d←d_p; val←v_p fi;
14    coend;
15    <if assigned & old-id=cur-id
16     then value(l)←val; date(l)←d fi;
17     locked←locked-{l};>
18 coend;
```

Figure 9.

The task "Update-Copies-in-View" for processor p (see Figure 9) brings all accessible copies up-to-date (see requirement R5). Recall that the local copies of accessible logical objects are locked by the virtual partition creation process prior to invoking this task. For each locked logical object l that has a copy on p (line 7) the most recent value "val" assigned to some physical copy of l is retrieved (lines 8-14). This value is assigned to the local copy of l and the copy is unlocked, assuming that p has not joined a new partition since this task was initiated (lines 15-17). The assignment of the most update value to p's copies of logical objects is done in parallel (this is the meaning intended for the construct "for each p∈P cobegin ... coend"). The exception handling notation of [C] is used in line 12 to specify the action to be executed when expected messages fail to be received.

The tasks responsible for enforcing rules R1-R3 are given in Figures 10-12. The tasks Logical-Read and Logical-Write signal an "abort" exception when a logical object l, which has to be read or written by a transaction, is inaccessible on the processor on which the transaction runs. In such a case the transaction has to be aborted.

```
1 task Logical-Read(in l: L, out val: Value)[abort];

2 <if assigned & accessible(l,lview)
3  then p←nearest(copies(l)∩lview);
4       send(p,"read",l,cur-id);
5       receive(p,l,val,d)
          [no-response: Create-new-VP; signal abort];
6  else signal abort;>
```

Figure 10.

```
1 task Logical-Write(in l: L,val: Value)[abort];

2 var A: set of P;

3 <if assigned & accessible(l,lview)
4  then P←copies(l)∩lview;
5     for each p∈P
6     cobegin
7       send(p,"write",l,val,cur-id);
8       receive(p,l,ack)
          [no-response: Create-new-VP; signal abort];
9     coend
10 else signal abort;>
```

Figure 11.

```
1   task Physical-Access;

2   cycle
3    select from
4     receive(p,"read",l,v)      →
                  wait until (l∉locked);
5                 <if assigned & v=cur-id
6                  then send(p,l,value(l),date(l)) fi>;
7     □
8     receive(p,"write",l,val,v) →
                  wait until (l∉locked);
9                 <if assigned & v=cur-id
10                 then value(l)←val; date(l)←cur-id;
11                     send(p,l,ack);
12                 fi>;
13  endselect
14  endcycle;
```

Figure 12.

6. OPTIMIZATIONS

The partition initialization protocol sketched in the previous section can be improved in several ways. This section suggests ways of improving its efficiency in an environment containing a large number of processors and large data objects. We also discuss a modification to rule R4 that reduces the number of aborted transactions when two-phase locking is used as the underlying concurrency control protocol.

Rule R5 requires that a processor p upon joining virtual partition v bring its copy of an accessible object l "up-to-date" by reading a copy of l with the largest *date* less than v. In the simple implementation given, p finds this copy by reading all copies on processors in its view. However, p can optimize its search for an up-to-date copy by making use of the recent partition assignment history of each processor in view(p). Let $previous_v(q)$ denote the largest virtual partition less than v that q was a member of. The optimized search strategy is for p to consider the processors in view(p) in decreasing order of their $previous_v$ values. The desired up-to-date value of l is found at a processor q such that:

(1) $previous_v(q) = \max\{previous_v(r) \mid r \in view(p)$ & l was accessible in $previous_v(r)\}$

Now, if processor p satisfies the role of q in the above condition, then p holds an up-to-date copy of l and no initialization for l_p is necessary.

The values of $previous_v(q)$ for all $q \in view(v)$ can be collected by the initiator in the first phase of the protocol creating v, and this set of values can be distributed to all members of v in the second phase of the protocol at no extra cost in messages or time.

The scenario where a subset of the members of a virtual partition, say v, splits off and forms a new virtual partition w is of practical importance because it occurs frequently. (It occurs, for example, when some members of v detect the failure of another member of v.) In such a scenario, all members of w contain the up-to-date copies of all accessible objects. Consequently, no initialization is required. This special case can be detected using the values of $previous_w$, specifically, in this case $previous_w(p)=v$ for every p that is a member of w.

In the partition initialization protocol of §5, a copy is brought up-to-date by reading another copy, *in its entirety*. If the object is large, a more economical approach is to apply to the out-of-date copy all of the writes that it missed. This, however, requires an efficient procedure for specifying and extracting the values of the missed writes.

Specification of the missing writes is made easy by applying Theorem 1'. Consider a copy of l with date v and on a processor currently assigned to partition w. Roughly speaking, Theorem 1' tells us that the copy missed the writes of transactions executing in virtual partitions with identifiers greater than v and less than or equal to w. Thus, this out-of-date copy can be brought up-to-date efficiently if the system can support a query on an arbitrary copy of the form: *retrieve (the values of) all physical writes on copy c by any transaction executing in u such that* $v<u\leq w$. Such queries can be supported by labelling the records of objects with "dates" in the same way that copies are currently labelled with dates, or by keeping a database log [GMBLLPPT] of all writes and their associated "dates."

One unfortunate consequence of rule R4 is that whenever a processor p joins a new partition, all ongoing transactions that have accessed a copy on p must be aborted. This can be very costly and should be avoided, if possible. It is not easy, though, to find a weaker version of R4--one requiring fewer abortions--without restricting the concurrency control protocol. On the other hand, if a particular concurrency control protocol is assumed, a weaker version of R4 can often be found.

Consider, for example, an implementation using a distributed version of two-phase locking [EGLT]. Assume that copies (rather than objects) are locked and that locks are held until the end of a transaction. In such a system, a transaction can be allowed to execute in a *set* of virtual partitions V_T, without compromising one-copy serializability, if the following conditions hold:

(1) The set of logical objects referenced by T is accessible in every virtual partition in V_T.
(2) The set of processors holding copies that were physically read or written by T are contained in the view of every virtual partition in V_T.
(3) The recover operation (see R5) does not read a copy that is locked for writing.

7. DISCUSSION

We have presented a replica control protocol based on the intuitive ideas that (1) a communication cluster can access a logical object if it contains a (weighted) majority of the object's copies and (2) logical operations are translated into physical operations on the copies *within a cluster* using the "read-one/write-all" rule. Although the basic ideas of the protocol are simple conceptually, its correct implementation is quite subtle because we did not assume that failures are "clean." In addition to providing high fault-tolerance, the proposed protocol implements logical reads very efficiently.

The novelty of our protocol lies in the fact that the virtual partition management subprotocol makes a large class of failures (namely omission and performance failures) look like "clean" communication failures that partition the network. As a result, protocols that have been designed for partition failures can be used in conjunction with our virtual partition management protocol in a more general and realistic processing environment. For example, many proposed data management schemes (e.g. [BGRCK, D, SW]) for partitioned systems require partition detection and, furthermore, assume A2 and A3. Generally, these schemes require nothing stronger than properties S1 through S3. Therefore, these schemes can use the virtual partition management protocol to "detect" virtual partitions and operate on them as if they were real partitions.

ACKNOWLEDGMENTS

We would like to thank Shel Finkelstein, Jim Gray, Bruce Lindsay, and Irv Traiger for a number of useful comments.

REFERENCES

[BGa] Bernstein, P., and Goodman, N., "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys 13,* 2, (June 1981) 185-222.

[BGb] Bernstein, P., and Goodman, N., "The Failure and Recovery Problem for Replicated Databases," *Proc. 2nd ACM Symp. on Princ. of Distributed Computing,* Montreal, Quebec, August 1983, 114-122.

[BGRCK] Blaustein, B.T., Garcia-Molina, H., Ries, D.R., Chilenskas, R.M., and Kaufman, C.W., "Maintaining Replicated Databases Even in the Presence of Network Partitions," *EASCON,* 1983.

[BSR] Bernstein, P., Shipman, D., and Rothnie, Jr., J., "Concurrency Control in a System for Dis-

tributed Databases (SDD-1)," *ACM Transactions on Database Systems 5,* 1 (March 1980), 18-51.

[C] Cristian, F., "Correct and Robust Programs," *IEEE Trans. on Software Engineering SE-10,* 2 (March 1984), 163-174.

[CASD] Cristian F., Aghili H., Strong R., and Dolev D. "Fault-Tolerant Atomic Broadcasts: from Simple Message Diffusion to Byzantine Agreement," Tech. Report, IBM Research San Jose, 1984.

[D] Davidson, S., "Optimism and Consistency in Partitioned Distributed Database Systems," *ACM Transactions on Database Systems 9,* 3 (September 1984), 456-482.

[ES] Eager, D., and Sevcik, K., "Achieving Robustness in Distributed Data Base Systems," *Transactions on Database Systems 8,* 3 (September 83), 354-381.

[EGLT] Eswaran, K., Gray, J., Lorie, R., and Traiger, I., "The Notions of Consistency and Predicate Locks in a Database System," *Comm. of the ACM 19,* 11 (November 1976), 624-633.

[ESC] El Abbadi, A., Skeen, D., and Cristian, F., "An Efficient, Fault-Tolerant Protocol for Replicated Data Management," Tech. Report, IBM Research San Jose, 1985.

[G] Gifford, D., "Weighted Voting for Replicated Data," *Proc. of the 7th Symposium on Operating Systems Principles* Dec. 1979.

[GMBLLPPT] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzulo, F., and Traiger, I., "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys 13,* 2 (June 1981), 223-242.

[H] Hadzilacos, V., "Issues of Fault Tolerance in Concurrent Computations," Tech. Report 11-84, Harvard University, Center for Research in Computing Technology, Cambridge, Massachusetts (June 1984).

[KR] Kung, H., and Robinson, J., "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems 6,* 2 (June 1982), 213-226.

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

[SW] Skeen, D., and Wright, D., "Increasing Availabilty in Partitioned Database Systems". *Proc. 3nd ACM Symp. on Princ. of Database Systems,* Waterloo, Canada, April 1984, 290-299. TR 83-581 Dept. of Computer Science, Cornell University, Ithaca NY 14853.

[TGGL] Traiger, I.L., Gray, J.N., Galtieri, C.A., and Lindsay, B.G., "Transactions and Consistency in Distributed Database Systems," *Transactions on Database Systems Vol. 7,* 3 (September 1982), 323-342.

[T] Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Data Bases," *ACM Transactions on Database Systems 4,* 2 (June 1979) 180-209.