An Algorithm for Replicated Objects with Efficient Reads (Extended Abstract)

Tushar D. Chandra
Google
Mountain View, CA, USA

Vassos Hadzilacos
University of Toronto
Toronto, ON, Canada

Sam Toueg
University of Toronto
Toronto, ON, Canada

1. INTRODUCTION

The problem. We consider the problem of implementing a consistent replicated object in a partially synchronous message passing distributed system susceptible to process and communication failures. The object is a generic shared resource, such as a data structure, a file, or a lock. The processes implementing the replicated object access it by applying operations to it at unpredictable times and potentially concurrently.[1] The object should be linearizable: it should behave as if each operation applied to it takes effect at a distinct instant in time during the interval between its invocation and its response.

The main reason for replicating an object is fault tolerance; if a copy of the object becomes inaccessible, the object can still be accessed through the other copies. Another reason for replicating an object is performance; a process that requires the object can access its local copy. The benefit of accessing a local copy, however, must be weighed against the cost of keeping the copies consistent.

Many objects support *read operations*, i.e., operations that return a function of the object's state without changing that state. In practice, read operations often vastly outnumber *read-modify-write (RMW) operations*, i.e., operations that modify the state and return a value that, in general, depends on the object's old state. It is in such instances that replication can be leveraged for performance, in addition to fault tolerance.

In this paper we present a fault-tolerant algorithm to implement a linearizable replicated object that handles read operations very efficiently and RMW operations about as efficiently as existing implementations of linearizable replicated objects. In our algorithm, every read operation is local, i.e., it does not generate messages. In addition, eventually every read operation is non-blocking, i.e., it can be completed in a constant number of local steps by the process that issued it, unless there is a "pending" RMW operation that conflicts with this read. Furthermore, there is a process for which eventually every read operation is non-blocking, regardless of conflicting RMW operations. We use the "shifting executions" technique [2,13,14,19] to prove that the blocking exhibited by our algorithm is, in some sense, optimal: every algorithm that implements linearizable objects has a run in which read operations of all processes but one block in the presence of conflicting RMW operations.

Model sketch. We assume a distributed system in which a fixed set of n processes communicate by exchanging messages. Messages sent are not corrupted and no spurious messages are generated. Processes are subject to crash failures only, and a majority of them are non-faulty.

The system is *partially synchronous*: it is initially asynchronous and after some unknown point in time, called the *system stabilization time*, it becomes synchronous [7]. In addition we assume that processes are equipped with approximately synchronized clocks. We refer to the values of process clocks as *local time*, to distinguish it from real time. We make the following assumptions on clocks, process speeds, and message delays.

Clocks. Each process p has a local clock denoted $ClockTime_p$. Local clocks are non-negative integers that are monotonically increasing with respect to real time, and they are *always* synchronized within a *known* constant ϵ of each other.[2] (This is satisfied if local clocks are within $\epsilon/2$ of *real* time.)

Processes. There is a known lower bound on the speed of processes (as measured on any local clock) that holds forever after some unknown real time.

Messages. There is a known upper bound δ on message delays (as measured on any local clock) that holds forever after some unknown real time.

Note that the clock properties are perpetual, while the process speed and message delay properties are eventual. Before these eventual properties hold, processes can be arbitrarily slow, and messages can take arbitrarily long to arrive and can even be lost.

Local and non-blocking reads and the properties of our algorithm. An object is defined by specifying a set of states Σ, a set of operations *Ops*, a set of responses *Res*. The effect of an operation $op \in Ops$ applied to state $s \in \Sigma$ is given by a transition function $\Delta : \Sigma \times Ops \to \Sigma \times Res$; if $\Delta(s, op) = (s', v)$ then the new state of the object is s' and the response of the operation is v. An operation op is a

[1] In practice these operations are generated by applications that wish to use the object. We assume that there is a mechanism, outside the scope of our algorithm, that allows applications to locate processes that implement the object.

[2] Many present-day devices are equipped with GPS, which can be used to provide highly synchronized clocks.

PODC'16 July 25-28, 2016, Chicago, IL, USA

ACM ISBN 978-1-4503-3964-3/16/07.

DOI: http://dx.doi.org/10.1145/2933057.2933111

read operation if, for every $s \in \Sigma$, $\Delta(s, op) = (s, v)$ for some $v \in Res$; *op* is a *read-modify-write (RMW)* operation if it is not a read operation.

A read operation R *conflicts with* a RMW operation W if there is an object state s such that if we execute R and W starting from s, R reads different values depending on whether it executes before or after W. More precisely, R and W conflict if there exist $s, s' \in \Sigma$ and $v \neq v' \in Res$ such that $\Delta(s, W) = (s', -)$, $\Delta(s, R) = (s, v)$, and $\Delta(s', R) = (s', v')$.

We say that reads are *local* if read operations do not result in messages being sent. More precisely, reads are local if the number of messages sent during the execution of the algorithm does not depend on the number of reads performed in the execution. We say that an instance of a read operation issued by process p in an execution is *non-blocking* if it completes within a constant number of steps of p, without waiting for a message to arrive or for the process's clock to reach a certain value.

In this paper we present an algorithm for implementing linearizable replicated objects in which read operations are always local and, in certain circumstances, non-blocking. Specifically, after the system stabilizes, reads at one process (the "leader") are always non-blocking; and at any other process p, a read blocks only if p is aware of an on-going RMW operation that conflicts with the read. In many practical settings, this means that with our algorithm most reads are indeed non-blocking. Even a read that blocks because of conflicting RMW operations, blocks only for a short period, namely for at most 3δ time (after the system stabilizes).

Our algorithm is quite robust, in that it retains important properties even when the assumptions of the model do not hold. If a majority of processes crash or the bounds on process speed or message delay never hold, only liveness is compromised — i.e., operations applied to the object may not terminate, but they will never return incorrect results. If clocks are not synchronized, the object remains consistent in the sense that the sub-execution consisting of the RMW operations (i.e., the execution excluding the read operations) is still linearizable, but reads may stall or return stale object states. Once clock synchrony is restored, however, reads will again return the current object state.

Roadmap. The rest of the paper is organized as follows. Our algorithm requires a leader election service, which we describe in Section 2; this service may be of wider interest than its use in our algorithm. In Section 3 we describe the algorithm. In Section 4 we show that in any algorithm reads exhibit blocking similar to that of our algorithm. We conclude in Section 5 with a discussion of related work.

2. ENHANCED LEADER SERVICE

We assume that each process has access to a procedure *leader*(), which returns the name of a process such that there is a nonfaulty process ℓ and a real time after which every call to *leader*() returns ℓ. In other words, the procedure *leader*() is the well-known Ω failure detector [5]. There are efficient implementations of *leader*(), even under weaker assumptions than those required for our algorithm [1, 18].

Note that with *leader*() multiple processes can consider themselves to be leaders at the same (local or real) time. For our implementation of replicated objects we need a leader election algorithm with a stronger property; namely, that at all *local* times, at most one process considers itself to be leader. We present an algorithm that transforms (as a "black box") any simple leader service *leader*() to one that satisfies this stronger property. The resulting service, called *enhanced leader service*, provides to each process a procedure $AmLeader(t_1, t_2)$. Roughly speaking, this procedure returns TRUE if the process that invoked it has been the leader *continuously* at all local times t such that $t_1 \leq t \leq t_2$, and returns FALSE otherwise. More precisely, the procedure $AmLeader(-, -)$ satisfies the following properties:

(EL1) If the calls $AmLeader(t_1, t_2)$ and $AmLeader(t'_1, t'_2)$ by *distinct* processes both return TRUE, where $t_1 \leq t_2$ and $t'_1 \leq t'_2$, then the intervals $[t_1, t_2]$ and $[t'_1, t'_2]$ are disjoint. (Hence, it is not possible for two different processes to be leaders at the same local time.)

(EL2) There is a nonfaulty process ℓ and a local time t^* such that for all local times $t_2 \geq t_1 \geq t^*$, the procedure $AmLeader(t_1, t_2)$ (i) returns TRUE if called by ℓ at any local time $t \geq t_2$; and (ii) returns FALSE if called by any process $p \neq \ell$ at any local time. (Hence, eventually some correct process ℓ is permanently the leader.)

The basic idea of our leader election enhancer is as follows. Each process p periodically calls *leader*() to determine what it believes is the current leader ℓ. It then sends to ℓ a leader-lease message that contains an interval of *local* time during which p "supports" ℓ as the leader. The leader-lease message also contains a counter c of the number of times p has observed the leader changing. Each process p collects all the leader-lease messages sent to it. To determine whether it has been *continuously* leader from local time t_1 to local time t_2, p checks that there is a majority of processes M such that for each $q \in M$, p has received from q a message m_1 with an interval that covers t_1 and a message m_2 (possibly the same as m_1) that covers t_2 *with the same counter c as* m_1. In other words, q was continuously supporting p as the leader throughout an interval that includes both t_1 and t_2. Detailed pseudocode for the enhanced leader service is shown in Appendix B.

3. THE ALGORITHM

We describe the algorithm in English, with comments to provide some intuition why the algorithm is correct. Detailed pseudocode is given in Appendix A. A detailed proof of the algorithm's correctness, and that of the enhanced leader service, will appear in the full paper.

Our algorithm uses the procedure $AmLeader(t_1, t_2)$ to effectively divide local time (i.e., the range of values of local clocks) into a sequence of maximal non-overlapping intervals, during each of which at most one process is continuously the leader, and the last of which is infinite and has a nonfaulty leader ℓ.

The leader has two functions: (i) to linearize the RMW operations using a consensus-like mechanism, and (ii) to issue "read leases", the mechanism the algorithm uses to execute read operations efficiently. These two functions, though not independent of each other, are well separated. To highlight this, we present the pseudocode in Appendix A in two colours: code in black implements the consensus-like mechanism that handles RMW operations; code in red implements the read-lease mechanism that handles read operations. (In black-and-white printing the red code appears in lighter gray.) If we ignore the special property of read operations and submit them as generic RMW operations, the

red code could be simply stripped away, and the resulting algorithm would be a linearizable implementation of replicated objects. The red code can be treated as an add-on, built on top of the black code. This modularity makes reasoning about correctness of the algorithm cleaner, and may make improvements to our algorithm easier to conceive and to prove correct. We consider this clean separation of the two functions an important feature of the algorithm.

For the first function, the leader collects into *batches* the RMW operations submitted by processes (including itself), and then attempts to commit each batch by a two-phase protocol: first, the leader attempts to *notify* all processes about the batch; when it determines that a majority of processes, including itself, have been notified, the leader informs the processes that the batch is *committed*. Each leader ensures that all nonfaulty processes agree on the same sequence of batches, *across all changes of leadership*: It is important to ensure that a batch left incomplete by a leader that was demoted (say because it crashed or was very slow), with some processes notified of the batch and others not, is handled consistently by the demoted leader's successors. Each process applies to its local replica the committed batches in sequence, and applies the operations in each batch in some pre-determined order, the same for all processes. When a process applies one of its own RMW operations to its replica, it determines the response of that operation. Since all processes apply the same sequence of RMW operations in the same order, the execution of RMW operations applied to the object is linearizable.

The second function of the leader is to periodically issue read leases. A read lease is a pair (j, t_s) where j is the sequence number of the latest batch committed by the leader, and t_s is the local time when the leader issued this lease. The end of the lease is $t_s + \mathit{LeasePeriod}$, where $\mathit{LeasePeriod}$ is a suitably defined parameter. Intuitively, the read lease (j, t_s) held by a process p is a promise by the leader to p that no batch $j' > j$ of RMW operations will be committed (by *any* leader, current or future) before local time $t_s + \mathit{LeasePeriod}$ according to p's clock, *unless p has been notified of batch j'*. We will explain later how the leader makes good on this promise and how processes use the leases to read.

Key data structures. If a leader that has notified other processes of batch k receives acknowledgements from $\lfloor n/2 \rfloor$ other processes that they have received the notification, we say that batch k is *committed*. If a batch is committed, its RMW operations have been applied, or are sure to be applied in the future, by all nonfaulty processes to their replicas. Each process p maintains three key pieces of information:

- An array Batch of *committed* batches that p knows about. Initially, $\mathit{Batch}[j] = \emptyset$ for each batch j. If $\mathit{Batch}[j] = O \neq \emptyset$, then p has been informed that the j-th batch has been committed and consists of the operations in set O.
- A triple (Ops, ts, k), which is called p's *estimate* and indicates the "freshest" batch of which p has been *notified* (but which is not necessarily committed). Ops is the set of RMW operations in that batch, and the pair (ts, k) uniquely identifies the batch and indicates its freshness: ts is the local time when the process that sent the notification to p became leader, and k is the sequence number of the batch. The larger this pair (in lexicographic ordering), the fresher the batch.
- An array $\mathit{PendingBatch}$ such that $\mathit{PendingBatch}[j] = O \neq \emptyset$ at p if O is the most recent batch with number j for which p has received a notification.

The algorithm satisfies the following invariants:

(I1) If a process has $\mathit{Batch}[j] = O \neq \emptyset$, then the value of $\mathit{Batch}[j]$ at that process remains O thereafter and if some other process has $\mathit{Batch}[j] = O' \neq \emptyset$ then $O = O'$. Furthermore, if $j \neq j'$, $\mathit{Batch}[j] \cap \mathit{Batch}[j'] = \emptyset$. That is, once assigned, the value of a batch is stable in each process, processes agree on the sequence of committed batches, and no operation is included in two different batches.

(I2) If p's estimate is (O, t, j), then p has $\mathit{Batch}[j-1] \neq \emptyset$; i.e., the previous batch of p's estimate is committed and p knows that batch.

(I3) If process p has $\mathit{Batch}[j] \neq \emptyset$, then for each i, $1 \leq i < j$, a majority of processes have $\mathit{Batch}[i] \neq \emptyset$; i.e., all previous batches are committed and each is known by a majority of processes.

Overall structure of the algorithm. Each process p is structured as three parallel threads:

- Thread 1 handles the RMW operations (lines 2–6) and read operations (lines 7–19) submitted to p.
- Thread 2 (lines 20–23) is an infinite loop in which p repeatedly checks to see if it is the leader at the present local time t. If so, it launches procedure $\mathit{LeaderWork}(t)$, described below, in which it performs the two tasks of a leader outlined above: committing the batches of RMW operations and issuing read leases. The process p remains in this procedure until it discovers that it is no longer the leader. While p is not a leader, its Thread 2 simply handles certain types of messages sent to it.
- Thread 3 (line 24) handles the messages received by p other than those handled in Thread 2.

Procedure *LeaderWork*. A demoted leader may leave its last batch in an indeterminate state, with some processes notified of this batch and others not. Thus, when a process p becomes leader at local time t and launches the procedure $\mathit{LeaderWork}(t)$ (line 22), its first order of business is to (i) commit a batch that may have been left half-done by a previous leader or ensure that this batch will not be committed, even if traces of it remain in the system — e.g., some process still has an estimate with that batch, and (ii) determine the state of the object after all committed batches. This initialization of a new leader (lines 26–36) is done as follows. (We note that throughout the description below the variable t refers to the local time when process p determined that it is the new leader.)

The new leader p first tries to find the most recent estimate that any process has (lines 26–30). It does so by sending $\langle \textsc{EstReq}, t \rangle$ messages to the other processes, requesting their current estimates, and waiting for $\lfloor n/2 \rfloor$ responses (lines 99—101) so that, together with its own, it has estimates from a majority of processes. To tolerate message loss, p sends EstReq messages periodically, all the while ensuring it is still the leader and exiting the procedure $\mathit{LeaderWork}(t)$ if it is not (line 29).

Once p has estimates from a majority of processes, it selects the "freshest" estimate $(\mathit{Ops}^*, ts^*, k^*)$ (line 31). The set of operations Ops^* of estimate $(\mathit{Ops}^*, ts^*, k^*)$ is not necessarily committed as batch k^*, but batch $k^* - 1$ is committed and p knows this batch. To see this recall that, by Invari-

ant (I2) of the algorithm, the process q that sent estimate (Ops^*, ts^*, k^*) to p knew batch k^*-1; q included that batch in its reply to p's EstReq message (line 101), and p assigned it to its $Batch[k^*-1]$ variable when it received q's estimate (line 90). Also, by Invariant (I3) of the algorithm, since q has $Batch[k^*-1] \neq \emptyset$, each of batches $1, 2, \ldots, k^*-2$ is known by a majority of processes, and therefore by at least one nonfaulty process. Thus, p can find any of these batches that it is missing from one of its nonfaulty peers. This is done by calling procedure $FindMissingBatches(k^*-2)$ (line 33).

Once the new leader p has batches $1..k^*-1$, it determines the state of the object after all operations in these batches have been applied. This is done by calling the procedure $ExecuteUpToBatch(k^*-1)$ (line 34). In catching up to the latest state of the object, the leader does not apply all these operations from scratch, but picks up from the most recent state it has computed. The leader now attempts to commit Ops^* as batch number k^* by calling the procedure $DoOps(Ops^*, t, k^*)$ (line 35), which we will describe shortly. If this call fails, p has ceased being leader and exits $LeaderWork(t)$. Otherwise, p initiates a single operation NoOp (that has no effect on the state of the object) as a RMW operation via Thread 1. This ensures the liveness of reads even if no process ever submits a RMW operation again. (Initiating this NoOp is unnecessary if processes keep submitting RMW operations.) At this point, p has successfully completed its initialization as leader, and proceeds with the leader's continuing tasks: granting read leases and committing new batches of RMW operations. To do so, p enters a loop (lines 39–51) in which it records the current local time t', verifies that it has been continuously the leader since local time t by calling $AmLeader(t, t')$ (line 40), and

- if it is time to renew the read leases, it sends a new read lease (k, t') to all other processes, where k is the number of the latest batch committed (line 44),
- if it has received requests for a set $NextOps$ of new RMW operations, it tries to commit this as the next batch of operations by calling the procedure $DoOps(NextOps, t, k+1)$ (line 49), and
- it sends the last committed batch to all other processes (line 51). This can be done "lazily"; its only purpose is to safeguard against message loss.

The leader repeats these actions until it ceases being leader, in which case it exits $LeaderWork(t)$ (lines 41 and 50).

Procedure *DoOps*. Next we describe the procedure $DoOps(O, t, j)$, by which a process p that became leader at local time t attempts to commit O as the j-th batch of RMW operations, and succeeds in doing so unless it crashes or loses its role as leader during the procedure's execution.

Process p first checks whether it has previously sent its estimate to a process that became leader later than it (line 52); if so, it abdandons its attempt to commit batch j and abdicates. Otherwise, p adopts (O, t, j) as its current estimate (line 53). It then notifies other processes of this new batch by sending $\langle$Prepare, $O, t, j, B\rangle$ to all other processes repeatedly (to safeguard against message loss), where B is p's current value of $Batch[j-1]$, until it receives acknowledgments from $\lfloor n/2 \rfloor$ processes (lines 54–58). When a process q receives this $\langle$Prepare, $O, t, j, B\rangle$ message, it sets its $Batch[j-1]$ variable to B, and if (O, t, j) is not outdated, it adopts (O, t, j) as its estimate and sets $PendingBatch[j]$ to O. If q's estimate is (O, t, j), q sends an acknowledgement to p.

Note that, because of the initialization step, and the fact that a leader commits batches in sequence, when p sends the above $\langle$Prepare, $O, t, j, B\rangle$ message, the set B is the previously committed batch $j-1$. So if a process q adopts (O, t, j) it also knows batch $j-1$, ensuring Invariant (I2). Thus, when the leader has received $\lfloor n/2 \rfloor$ acknowledgements to its Prepare messages it knows that a majority of processes (including itself) have adopted estimate (O, t, j), and know batch $j-1$. This ensures Invariant (I3).

Having received $\lfloor n/2 \rfloor$ acknowledgements to its Prepare messages, the leader p next checks that all processes potentially holding read leases have acknowledged; if not, it waits until the last read lease that it (or a previous leader) issued has expired (lines 59–62). This step is critical to ensure that processes do not read stale states, and we will describe it in more detail in the next subsection.

Finally, p verifies that it has continuously remained the leader from local time t (when it became leader) until the current local time t'; if not, p abandons $DoOps$ and abdicates (lines 63–64). If p is still the leader, it commits O as the j-th batch: it stores O in its $Batch[j]$ variable, applies the operations in O to the object's state (this is done by calling procedure $ExecuteBatch(j)$), and sends Commit messages to all other processes to inform them that batch j is now committed (lines 65–70).

The leaseholder mechanism. Suppose a process p with a lease, other than the leader, crashes or becomes temporarily disconnected from the leader. The leader attempting to commit a batch will be delayed by p's failure or disconnection: it will not receive an acknowledgement to the Prepare message it sent to p, and so it must wait for the read lease of p to expire. Our algorithm incorporates a mechanism, described below, that prevents a crashed or disconnected process from delaying RMW operations more than once.

The leader keeps track of a set of processes that it considers to be the current "leaseholders", i.e., the processes potentially holding a valid lease issued by this (or a previous) leader. The leader includes this set in each read-lease message that it sends periodically to all processes (line 44). A process q that receives such a message updates its lease only if it belongs to the set of leaseholders contained in that message (lines 113–116). If q is not in the set of leaseholders, it sends a LeaseRequest message to the leader (lines 114–116); when the leader receives that message, it reintegrates q by adding it to the set of leaseholders (line 46).

The leader p initializes the set of leaseholders to be all processes other than itself (line 25), and it maintains this set as follows. After receiving $\lfloor n/2 \rfloor$ acknowledgements to the Prepare message for some batch j, p waits for one of two conditions: (i) to receive acknowledgements from *all* current leaseholders, or (ii) for 2δ local time units to elapse since it started sending the Prepare messages (line 59).[3] If this waiting ends because of condition (i), it is safe for p to commit batch j right away. If, however, some current leaseholder fails to acknowledge within the maximum round-trip delay of 2δ, p delays committing the new batch until it is certain that all read leases it (or a previous leader) has issued have become invalid. More precisely, in this case, if

[3]Recall that δ is the worst-case delay of a message after the system stabilizes. For simplicity, we assume that the time β to process and reply to a Prepare message is negligible compared to δ and in fact is 0. Alternatively, p should wait for $2\delta + \beta$.

$(-, t_s)$ is the last read lease it issued and t is the local time when p became leader, p waits until its clock shows local time after $\max(t, t_s) + \mathit{LeasePeriod} + \epsilon$ (lines 60–61). The term $\max(t, t_s)$ ensures that p waits until the last lease it *or a previous leader* issued has expired. The term ϵ ensures that the lease has expired even at processes whose clocks are slow compared to p's clock.Thus, p makes sure that any process q that was not notified of the new batch j no longer has a valid lease for any previous batch, and so q will not read a stale state of the object. At this point, p updates the set of leaseholders to be the set of processes that acknowledged the PREPARE messages for the new batch j (line 62), and is ready to commit this batch. Hence, processes that did not acknowledge batch j in a timely manner cease being leaseholders, and p will no longer wait for their acknowledgements until they become leaseholders again.

Processing RMW operations. A process p that wishes to execute a RMW operation o, creates a unique ID (p, i) for that operation consisting of its name and a counter i, and sends $(o, (p, i))$ to the process it believes is the leader. (It does so repeatedly to deal with transient message loss and with the possible initial uncertainty about who the leader is (lines 2–5).) The leader will include this operation in some batch of operations, and will commit that batch using the *DoOps* procedure described earlier (lines 47–51). When p receives the COMMIT message for that batch, say $\langle \text{COMMIT}, O, j \rangle$, it stores O in $\mathit{Batch}[j]$, gets from its peers any batches smaller than j that it is missing (see Invariant (I3)), and applies the operations in O to its replica (lines 109–112). In doing so, it executes the RMW operation $(o, (p, i))$, which is one of the operations in O, and returns its response (line 6).

Processing read operations. A process p that wishes to execute a read operation R, waits until it has a read lease (k^*, t^*) that is valid, i.e., $t' < t^* + \mathit{LeasePeriod}$, where t' is the local time at p (lines 10–13). Process p then determines the batch $\hat{k}$ after which the read operation R should be linearized, i.e., the batch after which R should read the state of the object. A batch j is *pending* at p if p has received a PREPARE but not a COMMIT message for batch j. Process p uses its array *PendingBatch* to compute $\hat{k}$ as follows: if there is no batch $j > k^*$ that is pending at p and contains operations that conflict with R, then p sets $\hat{k} = k^*$ (the batch number for which p holds a valid lease); otherwise, $\hat{k}$ is the maximum number of a batch that is pending at p and contains an operation that conflicts with R (line 15). Note that $\hat{k} \geq k^*$. After determining $\hat{k}$, p waits until it knows all batches up to $\hat{k}$ (line 16), computes the corresponding state of the object if it has not already done so, and applies the read operation to that state.

Non-blocking reads. Consider the behaviour of read operations at a process p after the system stabilizes. After this occurs, p will always hold some valid read lease (k^*, t^*), so p does not wait in lines 10–13. Furthermore, p will already have batches $1, 2, \ldots, k^*$ in its *Batches* array. So if p has no pending batch $j > k^*$ with operations that conflict with R, then $\hat{k} = k^*$ and p does not wait in line 16. Thus p executes a read operation R without blocking unless p has a pending RMW operation that conflicts with R. It turns out that even if R blocks at p, it does so for at most 3δ local time units. Finally, since the permanently elected leader has no pending operations, it can always read without blocking.

Locality of reads. Consider the set of messages sent by the algorithm. If we exclude those sent by (and necessary to) the consensus mechanism used to linearize RMW operations, there are only two types of messages left, namely, those that are sent by the read-lease mechanism (the red code): LEASEGRANT messages to renew read leases and LEASEREQUEST messages to rejoin the set of leaseholders. Some time after the system stabilizes, only LEASEGRANT messages are sent. Furthermore, the frequency with which the permanent leader sends LEASEGRANT messages is fixed and independent of the number of read operations. In fact, the code that a process executes to do a read (lines 7–19) does not trigger the sending of *any* message. Thus, read operations are always local.

Properties of the algorithm. In the full paper we prove that our algorithm satisfies the following properties:

- The execution of the operations applied to the object is linearizable.
- Any operation issued by a correct process eventually terminates.
- Read operations are local.
- There is a time after which every read operation *op* issued by any process p is non-blocking unless p has a pending RMW operation that conflicts with *op*. A read operation that blocks, does so for at most 3δ local time units. Moreover, eventually no read operation by the leader blocks.

4. NECESSITY OF BLOCKING

In our algorithm all read operations are local and, after the system stabilizes, they never block at the leader, and they block at other processes only if there are pending RMW operations that conflict with the reads. We now prove that, in some sense, this is the best that can be done: Even under strong assumptions, any algorithm has runs in which read operations at $n-1$ processes block. Our proof uses the shifting executions technique introduced in [13]. This technique is used in [2,14,19] to determine, for each type of operation, a lower bound on the worst-case time of an instance of that operation in a run. Here we use it to show a lower bound on the *number of distinct processes* with a read operation that blocks in a run (as well as on the time during which these operations block). To do so we exploit the ability of each process to apply multiple operations concurrently. We also discuss a version of our result for the case that each process can apply operations only sequentially. Consider the following distributed system S:

Clocks. Each process p has a local clock denoted $\mathit{ClockTime}_p$. Local clocks are non-negative *real numbers* that are monotonically increasing, and they are *always* synchronized within a *known* constant, $\epsilon/2$ with respect to *real time*. (Hence, clocks are synchronized within ϵ of each other.)

Messages. There is a known upper bound δ on message delays (as measured in *real time*) that *always* holds.

Processes. Processes never crash. Each process can start multiple operations concurrently, and can start successive operations within γ real time of each other.

The parameter γ is a reflection of minimum process speed. In practice γ is much smaller than δ or ϵ; henceforth we assume that γ is negligible.

Consider any object O that has at least two states s_0 and s_1, a RMW operation W that changes the state from s_0 to s_1,

and a read operation that returns v_0 in s_0 and v_1 in s_1, for some $v_0 \neq v_1$.

THEOREM 4.1. *Any algorithm $\mathcal{A}$ that implements a linearizable object O in system S has a run r with a single instance of W in which each of $n-1$ processes executes a read operation that takes at least $\alpha = \min(\epsilon, \delta/2) - 2\gamma \approx \min(\epsilon, \delta/2)$ real time to complete.*

Since α depends on the synchrony of clocks ϵ and the delay of messages δ, and not on the rate at which a process executes its own steps, the $n-1$ read operations in the run r of Theorem 4.1 are blocked.

PROOF OF THEOREM 4.1. If *op* is an operation that appears in a run r, $start_r(op)$ and $end_r(op)$ denote, respectively, the real times when *op* starts and ends in r.

We build a run r of algorithm $\mathcal{A}$ as follows. All processes' clocks run exactly $\epsilon/2$ ahead of real time. All messages take exactly $\delta/2$ real time to be received. The object O is initially in state s_0. Each process invokes multiple read operations on O repeatedly and concurrently *as fast as it can*. We first let every process complete at least one read that returns the initial value s_0. We then let one process complete an instance of a W operation; this changes the state of O to s_1. All processes continue to start read operations until every process completes at least one operation that returns s_1. We now let all ongoing operations complete without starting any new ones. Note that in run r, any two successive reads by the same process start within 2γ real time units of each other (this is because each process starts an operation every γ, and there is at most one non-read operation, namely W, between any two successive reads of that process).

We claim that in r there are $n-1$ processes each of which executed a read operation that took at least $\alpha = \min(\epsilon, \delta/2) - 2\gamma$ real time. Assume, for contradiction, that there are two processes p and q all read operations of which took less than α time in r. Let R_p^0 and R_q^0 be the last read operations (by start time) of p and q, respectively, that returned v_0 in r. Without loss of generality, assume that R_p^0 starts at the same real time or later than R_q^0. Since R_q^0 and R_q^1 are consecutive reads of q, $start_r(R_q^1) \leq start_r(R_q^0) + 2\gamma$. Since R_q^1 completed in less than α real time, $end_r(R_q^1) < start_r(R_q^1) + \alpha \leq start_r(R_q^0) + \alpha + 2\gamma \leq start_r(R_p^0) + \alpha + 2\gamma$; so, $end_r(R_q^1) < start_r(R_p^0) + \alpha + 2\gamma$ $(*)$.

Now consider another run r' of $\mathcal{A}$ that is indistinguishable from r to all processes, in which, intuitively we delay the execution of process p by $\alpha + 2\gamma$ real time units. More precisely, the clocks of all processes other than p run exactly as in r, i.e., $\epsilon/2$ ahead of real time; p's clock is slower by $\alpha+2\gamma$ than in r; all messages to p take $\delta/2+\alpha+2\gamma$ real time, i.e., an additional $\alpha + 2\gamma$ compared to r; and all messages from p take $\delta/2-(\alpha+2\gamma)$ real time, i.e., $\alpha+2\gamma$ less than in r. Since $\alpha+2\gamma = \min(\epsilon, \delta/2)$, in run r', the clock of p is within $\epsilon/2 - \min(\epsilon, \delta/2)$ of real time, i.e., it is at most $\epsilon/2$ behind real time; the messages to p take $\delta/2+\min(\epsilon, \delta/2) \leq \delta$ time; and the messages from p take $\delta/2 - \min(\epsilon, \delta/2) \geq 0$ time. So r' is a legitimate run of $\mathcal{A}$ in system S.

Since $start_{r'}(R_p^0) = start_r(R_p^0) + \alpha + 2\gamma$, and $end_{r'}(R_q^1) = end_r(R_q^1)$, $(*)$ implies $start_{r'}(R_p^0) > end_{r'}(R_q^1)$. So in r', R_p^0, which returns 0, starts after the completion of R_q^1, which returns v_1. This violates linearizability. □

In the preceding proof, the run r we constructed requires each process to invoke multiple concurrent read operations on O. If each process can apply operations on O only sequentially, we can obtain a similar result, using a similar proof, by taking the value of α to be roughly half of that in Theorem 4.1, i.e., $\alpha \approx \min(\epsilon/2, \delta/4)$.

After the system stabilizes, in our algorithm a read may block for at most 3δ. Thus, if δ is within a constant factor of ϵ, our algorithm's blocking behaviour is within a constant factor of the lower bound. In the case when $\epsilon \ll \delta$, ongoing work suggests that there is an extension of our algorithm such that reads block on only $n-1$ processes, only for $\Theta(\epsilon)$ time, and only if there are conflicting RMW operations.

5. RELATED WORK

The problem of implementing linearizable objects was also studied for systems that are *reliable and synchronous* in the following sense: processes do not crash, messages are never lost, and there is a known bound on the maximum message delay (including processing time) that holds *always*, not only eventually [2, 13, 14, 19]. These assumptions greatly simplify the task of implementing linearizable objects. Because large-scale, geo-replicated distributed systems are subject to process and communication failures, and they experience intermittent periods of asynchrony, most of the literature has focused on implementations that tolerate this behaviour.

The literature on fault-tolerant replication algorithms is vast. Those algorithms that implement linearizable replicated objects typically follow Lamport's "state machine approach", whereby processes use an agreement algorithm to order the operations that they apply to their local copies of the object [10]. Our algorithm uses such an agreement algorithm for the RMW operations only, and uses a different mechanism, based on leases, to process reads efficiently. This should lead to better performance in settings where read operations outnumber RMW operations, which is common in many applications. The idea of supplementing a consensus mechanism with read leases was suggested in [4].

Our agreement algorithm that orders the RMW operations uses techniques that are commonly found in agreement algorithms [6, 7, 11, 12, 16], such as electing a leader to coordinate decisions and ensure progress, and intersecting majorities to ensure that later leaders' decisions are consistent with those of earlier ones. It also incorporates novel elements, especially in relation to the role and properties of the leader, that expedite the handling of failures and dovetail with the read-lease mechanism we use to process reads efficiently. We will highlight these elements as we contrast our algorithm with previous related work, specifically the replication algorithms used in Megastore [9], Spanner [8], and Paxos Quorum Leases [15], which come closest to our algorithm's aim of efficient reads; as well as with Viewstamped Replication [12,16] and Raft [17], two replication algorithms that do not make special provisions for reads.

Megastore and Spanner. Megastore [9] and Spanner [8] are systems developed to support globally distributed databases in Google's data centers. Both systems provide many functionalities in addition to replication, such as support for transactions, that are beyond the scope of our algorithm. Here we focus only on these systems' replication mechanisms. These systems rely on various pieces of Google's software infrastructure and are described in the above-cited papers at a high level, and without correctness proofs.

Megastore implements a replicated log that records updates to a distributed database. Processes append new en-

tries to this log and read the most recently committed entry of the log. To agree on the sequence of log entries, processes use a version of the Paxos consensus algorithm [11]. In Megastore, with each agreement on a log entry, processes also agree on the leader that will coordinate the decision for the next entry, namely the process expected to generate the next write operation (this choice expedites the processing of the next write). If the prediction for the next leader is wrong (e.g., because the chosen leader crashes or because the next write is actually generated at another process), Megastore provides a mechanism to change the leader, but this mechanism is vulnerable to livelock. An exponential back-off scheme can provide a probabilistic defence against this vulnerability. In contrast, our algorithm makes a deterministic guarantee of progress.

In Megastore, before the leader commits a write operation, it must receive acknowledgements that all processes have been notified of the write. If the leader does not receive an acknowledgement from some process p, it must delay the write until it is certain that p has marked its replica as out-of-date, preventing it from being read. This looks similar to our read leases, but the mechanism that Megastore uses relies on a specific piece of Google's software infrastructure, the lock service Chubby [3]. If the leader loses contact with Chubby while other processes maintain contact with it, writes can be left blocked forever. The authors of [9] note that they observed this problem "only a handful of times", and that it requires manual intervention by an operator to fix. Our algorithm is not subject to such vulnerabilities.

Spanner, a more recent system, relies on a different piece of Google's software infrastructure, the "TrueTime" service that provides processes with highly accurate clocks. TrueTime clocks expose the (small) uncertainty with which they track real time by returning the endpoints of a short *interval*, instead of an instant; the guarantee is that the real time when a clock reading is taken lies within the interval returned by the clock. Spanner uses TrueTime to implement a leader election service that guarantees that no two processes are leaders at the same *real* time. This is in contrast to our enhanced leader election service, which guarantees *only* that no two processes are leaders at the same *local* time. This seemingly innocuous difference has important consequences, discussed shortly.

Spanner assigns to each write and read operation a timestamp, which is an instant in time (the high endpoint of the interval returned by a clock). Write operations are linearized in timestamp order using the Paxos consensus algorithm [11], and a read operation with timestamp t returns a value based on the state of the object produced by the write operation with the largest timestamp not exceeding t. Write operation timestamps are generated by the leader, which delays the commit of a write until it is certain that the current real time is after the timestamp it generated. Thus, in Spanner all write operations pay the price of clock skew.[4] In contrast, in our algorithm, the real time to commit a batch of RMW operations does not depend on the clock skew ϵ after the system stabilizes.

[4] In the words of [8], "[i]f the [clock] uncertainty is large, Spanner slows down to wait out that uncertainty". To be fair, this waiting is tempered because it can be overlaid with other waiting due to the agreement algorithm. The extent of its effect depends on the clock skew.

To satisfy the timestamp ordering requirement for read operations in Spanner, a non-leader process that issues a read operation has three options: (a) have the read operation executed by the leader, which always has the most up-to-date state of the object; (b) use the current time as its timestamp and wait until it receives a write with a higher timestamp; or (c) use the maximum timestamp of any write operation it knows of. Option (a) means that reads are not local; it also concentrates load on the leader for each read, instead of distributing it among the replicas. Option (b) causes reads to block for an unbounded amount of time, even if there are no conflicting write operations. Option (c) may result in reading stale values, violating linearizability. In contrast, our algorithm ensures that all reads are local, they block only if there are conflicting pending writes (and only for 3δ time), and they never return stale values.

Paxos Quorum Leases. The Paxos Quorum Leases algorithm (PQL) [15] addresses a shortcoming of Megastore that affects the performance of write operations. As we mentioned earlier, in Megastore, before committing a write operation, the leader must receive acknowledgements that *all* processes have been notified of the write. To reduce the waiting time for writes, PQL maintains a set of leaseholders and waits only for the processes in this set to acknowledge notification of the write, as we do in our algorithm. The two algorithms, however, differ in important ways.

First, each lease renewal requires $\Theta(n^2)$ messages in PQL, as compared to $\Theta(n)$ in our algorithm. This is because in our algorithm only the leader renews leases by sending a lease-grant message to each process. In contrast, in PQL, a lease renewal involves a majority of processes, called *grantors*, sending a message to each leaseholder.

Second, in PQL every lease renewal interaction between each of the $\Theta(n^2)$ grantor-leaseholder pairs requires four rounds of communication (two round trips). In our algorithm, each of the $\Theta(n)$ leader-process pairs requires a single message (one way). This is because PQL uses timers (that measure elapsed time) instead of synchronized clocks. As a result, it requires a clever but expensive four-round protocol for grantors to know when a lease has expired at a leaseholder.

A third difference between our algorithm and PQL is that in PQL each change in the set of leaseholders (removing an unresponsive process or adding one that was isolated for a while) triggers the use of a consensus algorithm (Paxos, in PQL) for the processes to agree on the new set. Our algorithm manages the set of leaseholders in a much simpler way (see the discussion of the leaseholder mechanism in Section 3).

Finally, in PQL a pending write will cause all reads to block, even those with which it does not conflict. Furthermore, a steady stream of write operations can cause leases to be perpetually revoked, permanently disabling local reads. In contrast, even when faced with a steady stream of conflicting RMW operations, in our algorithm *all* reads are local, and after the system stabilizes, each read completes within at most 3δ time. This is because our algorithm *never* revokes read leases. Presented with a read operation while there are conflicting pending RMW operations, our algorithm determines the linearization point for the read (see the calculation of $\hat{k}$, line 15) and services the read from the local replica as soon as all RMW operations up to that point are committed.

Viewstamped Replication. Viewstamped Replication (VR) [12, 16] uses a leader-based mechanism for processes to agree on the sequence of operations to apply to their replicas, as in the state machine approach. It differs from our algorithm primarily in (a) the leader election mechanism that it employs, and (b) treating all operations (reads and RMW alike) in the same way. We elaborate on these differences below.

In VR, the period during which the same leader is in charge of committing operations is called a *view*. Processes take turns as leaders of successive views, based on the order of their IDs. In contrast, in our algorithm the leader is determined by the underlying Ω leader service, and that choice can be based on dynamic criteria such as the leader being well-connected to other processes, or being a process where the majority of RMW operations originate (to expedite their processing) [1, 9, 18]. With a static leader election scheme, if the next several processes to become leaders based on the IDs are partitioned away from the majority of processes, the system will cycle through a succession of ineffective views before it reaches one whose leader can commit operations.

VR treats read operations like all others, and it is not clear how to incorporate a read lease mechanism to it. The problem is that in a leader-based read-lease granting mechanism only one leader must issue leases at any time. Otherwise, an "old" leader unaware of its demotion can issue leases "behind the back" of the new leader, allowing processes to read a stale state. In VR, an old leader is unable to commit operations after it ceases being leader. If, however, in addition to trying to commit operations, an isolated old leader is also granting leases, these can cause trouble by allowing an also isolated recipient to read a stale state.

Raft. Raft [17] is an algorithm to implement a replicated log. A design choice that distinguishes it from other leader-based consensus algorithms is that information about the log entries flows only in one direction, *from* the leader *to* the other processes (the "followers"). Thus, a newly elected leader does not try to find out the latest log entries committed (or pre-committed) by its predecessor(s); rather, it imposes its own copy as the current state of the log. To prevent this from undoing previously committed log entries, Raft uses a specially-crafted leader-election algorithm that allows only a process with the most recent committed copy of the log to be elected leader. As noted in our discussion of VR, restricting the choice of leader has significant drawbacks. In addition, Raft's mechanism by which a newly elected leader imposes its copy of the log on the followers can be costly: It takes up to k message exchanges with each follower, where k is the total number of previous leaders. As a result of these message exchanges, a follower may be forced to successively return to its initial state. Raft also differs from our algorithm in the way it handles read operations. In fact, reads are not local and they always block: each read operation is sent to the current leader, and when the leader receives a read request it exchanges "heartbeat messages with a majority of the cluster before responding" to ensure that it is still the leader. The possibility of using a "form of lease" is briefly mentioned in [17], but no details are given. In contrast to our paper, [17] explains how to dynamically change the set of processes in charge of the replication, and it describes an implementation and performance evaluation.

6. REFERENCES

[1] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Dist. Comp.*, 21(4):285–314, 2008.

[2] Hagit Attiya and Jennifer Welch. Sequential consistency versus linearizability. *ACM TOCS*, 12(2):91–122, 1994.

[3] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06*, pages 335–350, 2006.

[4] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *PODC '07*, pages 398–407, 2007.

[5] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *JACM*, 43(4):685–722, 1996.

[6] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, 43(2):225–267, 1996.

[7] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, 35(2):288–323, 1988.

[8] James Corbett et al. Spanner: Google's globally-distributed database. In *OSDI '12*, pages 261–264, 2012.

[9] Jason Baker et al. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR '11*, pages 223–234, 2011.

[10] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

[11] Leslie Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, 1998.

[12] Barbara Liskov and James Cowling. Viewstamped replication revisited. *Technical Report MIT-CSAIL-TR-2012-021*, 2012.

[13] Jennifer Lundelius and Nancy Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984.

[14] Marios Mavronicolas and Dan Roth. Linearizable read/write objects. *TCS*, 220(1):267–319, 1999.

[15] Iulian Moraru, David Anderson, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. In *SoCC '14*, pages 1–13, 2014.

[16] Brian Oki and Barbara Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *PODC '88*, pages 8–17, 1988.

[17] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX ATC'14*, pages 305–320, 2014. An extended version of this paper is in *http://ramcloud.stanford.edu/raft.pdf*.

[18] Nicolas Schiper and Sam Toueg. A robust and lightweight stable leader election service for dynamic systems. In *DSN '08*, pages 207–216, 2008.

[19] Jiaqi Wang, Edward Talmage, Hyunyoung Lee, and Jennifer Welch. Improved time bounds for linearizable implementations of abstract data types. In *IPDPS '14*, pages 691–701, 2014.

APPENDIX

A. REPLICATED OBJECT ALGORITHM

Length and renewal frequency of read leases. To ensure that eventually every correct process has a valid read lease at all times, each LEASEGRANT lease renewal message sent by a leader p must be received before the lease granted by the previous LEASEGRANT message has expired. The leader p is scheduled to send a LEASEGRANT message every *LeaseRenewalPeriod* time units, but this sending may be delayed by up to α local time units, for some small constant α.[5] In addition, the LEASEGRANT message can be delayed by up to δ time units on p's clock, and the recipient's local clock can be up to ϵ time units ahead of p's clock. So, we need $\mathit{LeaseRenewalPeriod}+\alpha+\delta+\epsilon < \mathit{LeasePeriod}$ to ensure that each LEASEGRANT message is received before the read lease granted by the previous LEASEGRANT message has expired.

Simplifying assumptions. We assume that at every process p, p's clock $\mathit{ClockTime}_p$ increases by at least one time unit between any two successive readings of this clock by p. This can be enforced by delaying each clock reading until the clock value exceeds the previously read value. We also assume that the communication links are eventually FIFO. This can be achieved without violating the upper bound δ on message delays that holds after the system stabilizes.

Notation. The two segments of code enclosed by square brackets $[\cdots]$, namely lines 11–12 and 63–66, are uninterruptible — i.e., steps within such a block cannot be interleaved with steps of another thread.

CODE FOR PROCESS p:

variables:
Π = set of all processes
$(\mathit{Ops}, ts, k) = (\emptyset, -1, 0)$ /* current estimate */
$\mathit{Batch}[0, 1, 2, ...] = [\emptyset, \emptyset, \emptyset, \ldots]$ /* currently known batches */
$\mathit{MaxPendingBatch} = 0$ /* max pending batch number */
$\mathit{PendingBatch}[0, 1, 2, \ldots] = [\emptyset, \emptyset, \emptyset, \ldots]$ /* pending batches */
$\mathit{MaxPendingBatch} = 0$ /* max pending batch number */
$\mathit{state}[-1, 0, 1, 2, \ldots] = [s_0, s_0, \perp, \perp, \ldots]$
/* object state after each batch; s_0 = init state */
$\mathit{reply}(op) = \perp$ /* response to RMW operation op, init $\perp$ */
/* for every operation op */
$\mathit{cntr} = 0$ /* number of operations issued by p */
$\mathit{OpsRequested} = \emptyset$ /* RMW operations requested */
$\mathit{OpsDone} = \emptyset$ /* RMW operations executed */
$\mathit{LastBatchDone} = 0$ /* max batch number up to which */
/* all RMW operations are executed */
$t_{max} = -1$ /* max t s.t. p sent $\langle$ESTREPLY$, t, -, -, -\rangle$ */
$\mathit{est\_replied}[t] = \emptyset$ /* responders to $\langle$ESTREQ$, t\rangle$ */
$\mathit{est\_replies}[t] = \emptyset$ /* responses to $\langle$ESTREQ$, t\rangle$ */
$\mathit{P\text{-}acked}[t, j] = \emptyset$ /* responders to $\langle$PREPARE$, -, t, j, -\rangle$ */
$\mathit{LeaseHolders} = \Pi - \{p\}$
/* processes other than p possibly holding valid leases */
$\mathit{LeasePeriod}$ /* length of read lease */
$\mathit{LeaseRenewalPeriod}$ /* frequency of read lease renewal */
$\mathit{lease} = (0, -\infty)$ /* current lease held by p */
/* *lease* has two fields: *lease.batch* and *lease.start* */

[5]This delay is because up to α time units may elapse between two consecutive checks of whether it is time for p to send a LEASEGRANT message; the full paper contains a more detailed explanation and computation of what α is in our algorithm.

cobegin
// THREAD 1: /* issue read or RMW operations */
1 **while** TRUE **do**
2 **if** p wants to execute a RMW operation o **then**
3 $\mathit{cntr} := \mathit{cntr} + 1$
4 $\mathit{op} := (o, (p, \mathit{cntr}))$
5 **while** $\mathit{reply}(\mathit{op}) = \perp$ **do send** $\langle$OPREQ$, \mathit{op}\rangle$ **to** $\mathit{leader}()$
6 **return** $\mathit{reply}(\mathit{op})$
7 **if** p wants to execute a read operation o **then**
8 $\mathit{cntr} := \mathit{cntr} + 1$
9 $\mathit{op} := (o, (p, \mathit{cntr}))$
10 **repeat**
11 [$t' := \mathit{ClockTime}$
12 $(k^*, t^*) := \mathit{lease}$]
13 **until** $t' < t^* + \mathit{LeasePeriod}$
14 $u := \mathit{MaxPendingBatch}$
15 $\hat{k} := \max\{j \mid j = k^*$ **or** $(k^* < j \le u$ **and**
o conflicts with an operation in $\mathit{PendingBatch}[j]\}$
16 **wait for** (for all $j, 1 \le j \le \hat{k}, \mathit{Batch}[j] \neq \emptyset$)
17 $\mathit{ExecuteUpToBatch}(\hat{k})$
18 $(-, \mathit{reply}) := \Delta(\mathit{state}[\hat{k}], o)$
19 **return** *reply*
// THREAD 2:
20 **while** TRUE **do**
21 $t := \mathit{ClockTime}$
22 **if** $\mathit{AmLeader}(t, t) =$ TRUE **then** $\mathit{LeaderWork}(t)$
23 $\mathit{ProcessClientMessages}()$
// THREAD 3:
24 $\mathit{ProcessMessages}()$
coend

procedure $\mathit{LeaderWork}(t)$:
/* New leader initialization: find latest batch and (re)do */
25 $\mathit{LeaseHolders} := \Pi - \{p\}$
26 **repeat**
27 **send** $\langle$ESTREQ$, t\rangle$ **to** $\Pi - \{p\}$
28 $t' := \mathit{ClockTime}$
29 **if** $\mathit{AmLeader}(t, t') =$ FALSE **then return**
30 **until** $|\mathit{est\_replied}[t]| \ge \lfloor n/2 \rfloor$
31 $(\mathit{Ops}^*, ts^*, k^*) :=$ tuple in $\mathit{est\_replies}[t] \cup \{(\mathit{Ops}, ts, k)\}$ with maximum (ts^*, k^*)
32 **if** $ts^* \ge t$ **then return**
33 $\mathit{FindMissingBatches}(k^* - 2)$
34 $\mathit{ExecuteUpToBatch}(k^* - 1)$
35 $\mathit{outcome} := \mathit{DoOps}(\mathit{Ops}^*, t, k^*)$
36 **if** $\mathit{outcome} =$ FAILED **then return**
37 initiate a NOOP as a RMW operation via Thread 1
/* Grant read leases and process new batches */
38 $\mathit{NextSendTime} := \mathit{ClockTime}$
39 **while** TRUE **do**
40 $t' := \mathit{ClockTime}$
41 **if** $\mathit{AmLeader}(t, t') =$ FALSE **then return**
42 **if** $t' \ge \mathit{NextSendTime}$ **then**
43 $\mathit{lease} := (k, t')$
44 **send** $\langle$LEASEGRANT$, \mathit{lease}, \mathit{LeaseHolders}\rangle$ **to** $\Pi - \{p\}$
45 $\mathit{NextSendTime} := t' + \mathit{LeaseRenewalPeriod}$
46 **if received** $\langle$LEASEREQUEST$\rangle$ **from** q **then**
$\mathit{LeaseHolders} := \mathit{LeaseHolders} \cup \{q\}$
47 $\mathit{NextOps} := \mathit{OpsRequested} - \mathit{OpsDone}$
48 **if** $\mathit{NextOps} \neq \emptyset$ **then**
49 $\mathit{outcome} := \mathit{DoOps}(\mathit{NextOps}, t, k + 1)$
50 **if** $\mathit{outcome} =$ FAILED **then return**
51 **send** $\langle$COMMIT$, \mathit{Ops}, k\rangle$ **to** $\Pi - \{p\}$

```
procedure DoOps(O, t, j):
52  if t < t_max then return FAILED
53  (Ops, ts, k) := (O, t, j)
54  repeat
55    send ⟨PREPARE, O, t, j, Batch[j − 1]⟩ to Π − {p}
56    t′ := ClockTime
57    if AmLeader(t, t′) = FALSE then return FAILED
58  until |P-acked[t, j]| ≥ ⌊n/2⌋
59  wait until LeaseHolders ⊆ P-acked[t, j] or
      2δ local time has elapsed since p first executed line 55
60  if ¬(LeaseHolders ⊆ P-acked[t, j]) then
61    wait until ClockTime > max(t, lease.start) +
                             LeasePeriod + ε
62  LeaseHolders := P-acked[t, j]
63  [ t′ := ClockTime
64  if AmLeader(t, t′) = FALSE then return FAILED
65  (Batch[j], lease) := (O, (j, t′))
66  ExecuteBatch(j) ]
67  OpsDone := OpsDone ∪ Batch[j]
68  LastBatchDone := max(LastBatchDone, j)
69  send ⟨COMMIT, O, j⟩ to Π − {p}
70  return DONE

procedure FindMissingBatches(k′):
71  repeat
72    Gaps := {j | 1 ≤ j ≤ k′ and Batch[j] = ∅}
73    if Gaps ≠ ∅ then
        send ⟨MISSINGBATCHES, Gaps⟩ to Π − {p}
74  until Gaps = ∅
75  return

procedure ExecuteBatch(j′):
76  s := state[j′ − 1]
77  let op^1, op^2, ..., op^m be the operations in Batch[j′]
                              listed in operation id order
78  for i = 1 to m do  /* apply operation i of Batch[j′] */
      (s, reply(op^i)) := Δ(s, op^i.type)
79  state[j′] := s
80  return

procedure ExecuteUpToBatch(j′):
81  for j = LastBatchDone + 1 to j′ do
82    ExecuteBatch(j)
83    OpsDone := OpsDone ∪ Batch[j]
84    LastBatchDone := max(LastBatchDone, j)
85  return

procedure ProcessMessages():
86  while TRUE do
87    if received ⟨OPREQ, op⟩ from q then
88      OpsRequested := OpsRequested ∪ {op}
89    if received ⟨ESTREPLY, t, O′, t′, j′, B′⟩ from q then
90      Batch[j′ − 1] := B′
91      est_replied[t] := est_replied[t] ∪ {q}
92      est_replies[t] := est_replies[t] ∪ {(O′, t′, j′)}
93    if received ⟨P-ACK, t, j⟩ from q then
94      P-acked[t, j] := P-acked[t, j] ∪ {q}
95    if received ⟨MISSINGBATCHES, Gaps′⟩ from q then
96      for all j ∈ Gaps′ such that Batch[j] ≠ ∅ do
          send ⟨BATCH, j, Batch[j]⟩ to q
97    if received ⟨BATCH, j, O⟩ from a process q then
98      Batch[j] := O

procedure ProcessClientMessages():
99  if received ⟨ESTREQ, t⟩ from q then
100   t_max := max(t_max, t)
101   send ⟨ESTREPLY, t, Ops, ts, k, Batch[k − 1]⟩ to q
102 if received ⟨PREPARE, O, t, j, B⟩ from q then
103   Batch[j − 1] := B
104   if t ≥ t_max and (t, j) > (ts, k) then
105     (Ops, ts, k) := (O, t, j)
106     PendingBatch[k] := Ops
107     MaxPendingBatch := max(MaxPendingBatch, k)
108   if (Ops, ts, k) = (O, t, j) then send ⟨P-ACK, t, j⟩ to q
109 if received ⟨COMMIT, O, j⟩ from some process then
110   Batch[j] := O
111   FindMissingBatches(j − 1)
112   ExecuteUpToBatch(j)
113 if received ⟨LEASEGRANT, lease′, LeaseHolders′⟩ from q then
114   if p ∈ LeaseHolders′ then
115     if lease′ > lease then lease := lease′
116   else send ⟨LEASEREQUEST⟩ to q
117 return
```

B. LEADER ENHANCER ALGORITHM

Length and renewal frequency of leader leases. To ensure that a permanent leader is eventually elected, each LEADERLEASE renewal message must be received before the lease granted by the previous LEADERLEASE message has expired. So we need $LeaderLeaseRenewalPeriod + \alpha + \delta + \epsilon < LeaderLeasePeriod$ for a small constant α, as explained in connection with read leases in Appendix A.

CODE FOR PROCESS p:

```
variables:
t_s = t_e = 0        /* [t_s, t_e) is lease period for next leader */
L = L_prev = ⊥       /* current and previous leader votes */
c = 0                /* number of times p changed its leader vote */
LeaderLeases = ∅     /* set of leader lease periods received */
Supporters = ∅ /* processes that continuously vote for p */
LeaderLeasePeriod                /* length of leader lease */
LeaderLeaseRenewalPeriod         /* lease renewal frequency */

1   repeat every LeaderLeaseRenewalPeriod
    /* vote for leader of next period */
2     t := ClockTime
3     L := leader()
4     if L ≠ L_prev then                  /* if leader changed */
5       c := c + 1 /* increment the vote change counter */
6       L_prev := L
7     t_s := t_e    /* next lease starts when previous ends */
8     t_e := t + LeaderLeasePeriod   /* end of next lease */
9     P := [t_s, t_e)                     /* next lease period */
10    send ⟨LEADERLEASE, P, c⟩ to L

upon receipt of ⟨LEADERLEASE, P′, c′⟩ from process q:
11  LeaderLeases := LeaderLeases ∪ {(P′, q, c′)}

procedure AmLeader(t_1, t_2):
12  Supporters := {q | LeaderLeases contains tuples
      (P_1, q, i), (P_2, q, i) (not necessarily distinct, but
      with the same third component i) such that
      P_1 covers t_1 and P_2 covers t_2}
13  if |Supporters| > n/2 then return TRUE
14  else return FALSE
```