Asynchronous Waitfree Linearizability

1. INTRODUCTION

In this paper we address fundamental questions concerning the possibility of implementing data-structures that can tolerate process failures in an asynchronous distributed setting, while guaranteeing correctness (linearizability with respect to a given sequential specification) and progress (wait-freedom). We consider a class of data-structures whose operations can be classified into two kinds: *update* operations that can modify the data-structure but do not return a value and *read* operations that return a value, but do not modify the data-structure. We show that if the set of all update operations satisfy certain algebraic properties, then waitfree linearizable replication is possible for the data-structure. We also show that under certain conditions waitfree linearizable replication is not possible.

2. THE PROBLEM

State machine replication is a general approach for implementing data-structures that can tolerate process failures by replicating state across multiple processes. The key challenge in state machine replication is to execute data-structure operations on all replicas such that linearizability can be guaranteed.

A state machine $\mathcal{M}$ consists of a set of states $\Sigma_{\mathcal{M}}$ and a set of procedures $Procs_{\mathcal{M}}$. Every procedure has a set of parameters and we assume that the parameters are of primitive type. In the sequel, we will use the term *operation* to refer to a tuple of the form $(p, a_1, \cdots, a_n)$ consisting of the name p of the procedure invoked as well as the actual values $a_1, \cdots, a_n$ of the parameters. In general, the semantics of operations is given by a function that maps an input state to an output state as well as a return value. In this paper, we consider a special class of state machines. We assume that operations of the state machine can be classified into two kinds: *updates* (operations that modify the state) and *reads* (operations that do not modify the state, but return a value). Thus, an operation that modifies the state and returns a value is not permitted. Note that operations on the data-structure are deterministic. Furthermore, we assume that the update operations of the given state-machine either commute or nullify as explained below.

We say that two (update) operations op_1 and op_2 *commute* iff $op_1(op_2(s)) = op_2(op_1(s))$ for every state s. We say that an operation op_1 *nullifies* an operation op_2 iff $op_1(op_2(s)) = op_1(s)$ for all states s. We say that a partial-ordering $<_s$ on a set of operations (of the given state machine) is an NC-ordering if it satisfies the following conditions:

1. if $op_1 <_s op_2$, then op_2 nullifies op_1.
2. if $op_1 \not<_s op_2$ and $op_2 \not<_s op_1$, then op_1 and op_2 commute.

We say that a state machine is a NC state machine if its operations can be classified as updates or reads, and the set of all its update operations has a NC partial ordering $<_s$.

We now show that wait-free replication is possible for any NC state machine. In our replication algorithm, the original replica receiving an operation op augments it with some auxiliary information ts and represents the operation as the pair (ts, op). In the simplest case, the auxiliary information ts is used to serve as an unique id for each operation. This suffices if we just want to guarantee serializability. More generally, to achieve linearizability, we use ts as a timestamp that helps identify operations whose executions do not overlap (i.e., non-concurrent operations). We refer to the ordered pair (ts, op) as an *update command.* We assume that the update commands representing different invocations are distinct (which can be ensured by associating a unique-id with the operation instance or the timestamp).

Examples

Read-Write Register. A read-write register provides operations to write a value to a register and operations to read the current value of the register. Only writes are update operations. In this example, all update operations nullify each other. Let V denote the set of storable values. Then, we have:

$$U = \{\text{write}(v) \mid v \in V\}$$
$$R = \{\text{read}()\}$$

Read-Write Memory. A read-write memory is essentially a collection of read-write registers. The permitted update operations are write operations that write a new value to a given location. In this example, update operations on different memory locations commute with each other, while update operations on the same memory location nullify each other.

Atomic Snapshot Object. An atomic snapshot object is the same as a read-write memory as far as update operations are concerned. It differs from a read-write memory in providing a read operation that can return the values of all memory locations (in one operation).

Counter. Consider a Counter that provides update operations to increment and decrement the value of a single counter. All update operations commute in this case.

Resettable Counter. This extends the Counter interface by providing an additional update operation to reset the value of the counter to zero. (More generally, we can also support a write operation that writes a specific new value to the counter.) In this case, all increment and decrement operations commute with each other. The reset operation nullifies all increment and decrement operations.

Union-Find. In a *Union-Find* data-structure, every element is initially in an equivalence class by itself. A *union* operation (on two elements) merges the equivalence classes to which the two elements belong into one equivalence class. A *find* operation returns an unique representative element of the equivalence class. We make this specification deterministic by assuming that a total-ordering exists on the set of all elements and that the *find* operation returns the minimum element (with respect to this ordering) of the equivalence class to which the parameter belongs. All the update operations of this data-structure commute.

Other Examples

1. Sets (add, remove, member)
2. Maps or key-value stores (update, lookup)
3. Heaps (add, remove, findmin)

3. REPLICATION FOR NC STATE MACHINES

We now describe our algorithm for state machine replication based on generalized lattice agreement. In this algorithm, the lattice L is defined to be the power set of all update commands with the partial order $\sqsubseteq$ defined to be set inclusion. We refer to a set of update commands as a *cset*. We assume that all processes act as proposers, acceptors, and learners, for simplicity.

Updates. A process executes an update operation op by first creating a timestamp ts for the operation (as explained later) and then executing the procedure ReceiveValue($\{(ts, op)\}$), taking the singleton set $\{(op, cmd)\}$ to be a newly proposed value. It then waits until this command has been learnt.

Reads. Reads are processed by computing a state that reflects all commands in the learnt *cset* applied in an order determined as follows. Assume that a partial-order $<_d$ is defined on timestamps. This can be used to capture a *dynamic order* on update commands reflecting execution order

Algorithm 1 ReplicatedStateMachine

procedure ExecuteUpdate(op)
    let ts = get-time-stamp() **in**
    ReceiveValue($\{(ts, op)\}$)
    waituntil $(ts, op) \in$ LearntValue()

procedure State SerializableRead()
    return Apply(LearntValue())

procedure State Apply(S)
    let $cmd_1, \cdots, cmd_k$ = topological-sort(S, $<_c$) **in**
    let $(ts_i, op_i) = cmd_i$ **in**
    let s_0 = initial-state **in**
    let $s_i = op_i(s_{i-1})$ **in**
    return s_k

procedure MMLearntValue()
    Request LearntValue() from all processes
    Let M be a majority of responses
    return max(M)

procedure State LinearizableRead()
    let v = MMLearntValue() **in**
    Broadcast v to all processes and wait for a majority response
    return Apply(MMLearntValue())

between non-overlapping operation instances. We say that $ts_1 \parallel ts_2$ iff $(ts_2 \not<_d ts_1) \wedge (ts_1 \not<_d ts_2)$

Let $<_s$ be a NC partial-order on the set of all update operations. Note that this is a *static ordering* on operations. We define a "combined" partial-order $<_c$ on *cset* as follows. We say that $(ts_1, op_1) <_c (ts_2, op_2)$ iff

$$(ts_1 <_d ts_2) \vee ((ts_1 \parallel ts_2) \wedge (op_1 <_s op_2)).$$

Procedure Apply(S) computes a state corresponding to a set S of update commands by executing them in an order consistent with the partial ordering $<_c$.

A serializable read operation is achieved by simply returning Apply(S) where S is any learnt value. To achieve linearizability, we utilize the procedure MMLearntValue() to find the latest (maximum) learnt value among a majority of processes and compute the state corresponding to this learnt value and return it.

Generating Timestamps. A replica initiating a new operation can generate a timestamp for the operation as follows. Note that the main goal is to ensure that timestamp of the new operation is $>_d$ the timestamp of any operation that has already been completed. We can use MMLearntValue() to serve as this timestamp. The replica sends a request to every acceptor and waits for a response from a majority of the acceptors. Every acceptor responds by sending its current "accepted value". The replica takes the join (max) of the received responses and uses it as the timestamp. (Of course, we can use a more compact representation for the timestamp using a an equivalent vector-clock.)

Note that this ensures that if cmd_1 completes execution before cmd_2 is initiated, then cmd_1's timestamp will be $<_d$ the timestamp of cmd_2. However, the converse does not hold true. In particular, if the timestamp of $cmd_1 <_d$ the timestamp of cmd_2, cmd_1 may be included in a learnt value much later than cmd_2. However, this is not a problem.

Proof Of Linearizability

Let L_i denote the i-th learnt value. Let C_i denote the set of all update commands with a timestamp $< L_i$. We can consider C_i to be the set of all update commands which have been "initiated" before the i-th value has been learnt. Note that $L_i \subseteq C_i$. We define S_i to be the set $\{c \in C_i \mid \exists c' \in L_i.c <_s c'\}$. Thus, we have $L_i \subseteq S_i \subseteq C_i$. We establish linearizability by showing that the S_i forms an increasing chain and that all read values correspond to these sets.

4. IMPOSSIBILITY RESULTS

Consider a state machine with an initial state σ_0. Let op_1 and op_2 be two operations on the state machine. Let σ_i denote the state $op_i(\sigma_0)$ and let $\sigma_{i,j}$ denote the state $op_j(op_i(\sigma_0))$. We say that op_1 and op_2 are *2-distinguishable* in state σ_0 iff $\{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\} \cap \{\sigma_2, \sigma_{2,1}, \sigma_{2,2}\} = \phi$. Note that this essentially says the following: the state produced by execution of op_1, optionally followed by the operation op_1 or op_2, is distinguishable from the state produced by the execution of op_2, optionally followed by the operation op_1 or op_2.

THEOREM 4.1. *A state machine with 2-distinguishable operations op_1 and op_2 in its initial state can be used to solve consensus for 2 processes. Thus, it has a consensus number of at least 2.*

PROOF. Assume that we have a waitfree linearizable implementation of the given state machine. The following reduction shows how we can solve binary consensus for two processes using the state machine implementation.

1: **procedure** Consensus (Boolean b)
2: **if** (b) **then** op_1() **else** op_2() **endif**
3: s = read()
4: **return**($s \in \{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$)

Consider the execution of the above algorithm by two processes p and q. Since the state machine implementation is waitfree, the above algorithm will clearly terminate (unless the executing process fails).

We first show that when neither process fails, both processes will decide on the same value (*agreement*) and that this value must be one of the proposed values (*validity*). Let s_x denote the value read by process x (in line [2]). To establish agreement, we must show that $s_p \in \{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$ iff $s_q \in \{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$.

Let f_x denote the update operation performed by process $x \in \{p, q\}$ (in line [2]). Without loss of generality assume that the update operation f_p executes before f_q (in the linearization order). If f_q executes before the read operation by p, then both processes will read the same value and agreement follows.

Thus, the only non-trivial case (for agreement) is the one where p executes its read operation before q executes its update operation (f_q). Thus, $s_p = f_p(\sigma_0)$ while $s_q = f_q(f_p(\sigma_0))$. Without loss of generality, we can assume that the operation f_p is op_1 (since the other case is symmetric). Operation f_q can, however, be either op_1 or op_2.

Thus, $s_p = \sigma_1$, while s_q is either $\sigma_{1,1}$ or $\sigma_{1,2}$. Hence, agreement holds even in this case.

As for validity: note that this algorithm *decides* on the value proposed by the process that first executes its update operation. Specifically: the value read by either process will belong to $\{\sigma_1, \sigma_{1,1}, \sigma_{1,2}\}$ iff the first update executed is op_1.

This shows that both validity and agreement holds when both processes are correct. If either of the two processes fails, then agreement is trivially satisfied. Validity holds just as explained above.

□

We can extend the above result to n processes as follows. Let $\gamma = [e_1, \cdots, e_k]$ be a sequence where each element e_i is either op_1 or op_2. Define $\gamma(\sigma)$ to be $e_k(\cdots(e_1(\sigma))\cdots)$. Define $\mathit{first}(\gamma)$ to be e_1. Let Γ_k denote the set of all non-empty sequences, of length at most k, where each element is either op_1 or op_2.

We say that op_1 and op_2 are k-distinguishable in state σ_0 if for all $\gamma_1, \gamma_2 \in \Gamma_k$, $\gamma_1(\sigma_0) = \gamma_2(\sigma_0)$ implies $\mathit{first}(\gamma_1) = \mathit{first}(\gamma_2)$. In other words, consider two sequences γ_1 and γ_2 in Γ_k such that $\mathit{first}(\gamma_1) \neq \mathit{first}(\gamma_2)$. Then, the final states produced by executing the sequences of operations γ_1 and γ_2 will be different. Loosely speaking, we can say that the effect of the first operation executed has a "memory effect" that lasts for at least $k-1$ more operations.

Define Σ_i to be $\{\gamma(\sigma_0) \mid \gamma \in \Gamma_k, \mathit{first}(\gamma) = op_i\}$, where $i \in \{1, 2\}$. Note that op_1 and op_2 are k-distinguishable in state σ_0 iff Σ_1 and Σ_2 are disjoint.

THEOREM 4.2. *A state machine with k-distinguishable operations op_1 and op_2 in its initial state can be used to solve consensus for k processes. Thus, it has a consensus number of at least k.*

PROOF. We use the following generalization of our previous reduction scheme:

procedure Consensus (Boolean b)
if (b) **then** op_1() **else** op_2() **endif**
s = read()
return($s \in \Sigma_1$)

The proof follows as before.

□

We now show that the k-distinguishability condition reduces to a simpler non-commutativity property for *idempotent* operations. We say that an operation op is *idempotent* if repeated executions of the operation op have no further effect. We formalize this property as follows. Let γ be a sequence of operations. Define $\gamma!op$ to be the sequence obtained from γ by omitting all occurrences of op except the first one. We say that op is idempotent if: for all sequences γ, $\gamma(\sigma_0) = (\gamma!op)(\sigma_0)$.

Let op_1 and op_2 be two idempotent operations. Then, for any $k \geq 2$, op_1 and op_2 are k-distinguishable in σ_0 iff op_1 and op_2 are 2-distinguishable in σ_0. This condition can be further simplified to: $\{\sigma_1, \sigma_{1,2}\} \cap \{\sigma_2, \sigma_{2,1}\} = \phi$.

Note that the above condition can be equivalently viewed as follows:

1. op_1 and op_2 behave differently in σ_0: $op_1(\sigma_0) \neq op_2(\sigma_0)$.
2. op_1 and op_2 do not commute in σ_0: $op_1(op_2(\sigma_0)) \neq op_1(op_2(\sigma_0))$.
3. op_1 does not nullify op_2 in σ_0: $op_1(op_2(\sigma_0)) \neq op_1(\sigma_0)$.
4. op_2 does not nullify op_1 in σ_0: $op_2(op_1(\sigma_0)) \neq op_2(\sigma_0)$.

Note that the notions of commutativity and nullification used above are with respect to a single initial state.

Note that state machines (or interfaces) in a distributed setting are often designed to be *idempotent* (i.e., all its operations are designed to be idempotent) since a client may need to issue the same operation multiple times (when it does not receive a response back) in the presence of message failures. This may simply require clients to associate a unique identifier to each request they make so that the system can easily identify duplicates of the same request. (Recall that an operation, as defined earlier, includes all the parameters passed to a procedure.)

THEOREM 4.3. *A state machine with 2-distinguishable idempotent operations op_1 and op_2 in its initial state can be used to solve consensus for any number of processes. Thus, it has a consensus number of ∞.*

Extension.

The above theorems immediately tell us that waitfree linearizable implementations of certain data-structures or state-machines are not possible in an asynchronous model of computation (in the presence of process failures). The above theorem requires 2-distinguishable idempotent operations in the initial state. We can generalize this to state-machines where such operations exist in states other than the initial states.

We say that a state σ is a *reachable* state iff there exists a sequence of operations γ such that $\sigma = \gamma(\sigma_0)$.

THEOREM 4.4. *Consider a state machine all of whose operations are idempotent. Suppose the state machine has a reachable state σ and two operations op_1 and op_2 such that op_1 and op_2 are 2-distinguishable in σ. Then, the given state machine can be used to solve consensus for any number of processes. Thus, it has a consensus number of ∞.*

PROOF. Since σ is reachable, there exists a sequence of operations $[f_1, \cdots, f_k]$ such that $[f_1, \cdots, f_k](\sigma_0) = \sigma$. We use the following reduction:

procedure Consensus (Boolean b)
  $f_1(); \cdots; f_k();$
  if (b) **then** $op_1()$ **else** $op_2()$ **endif**
  s = read()
  return$(s \in \{op_1(\sigma), op_2(op_1(\sigma))\})$

The proof follows as before. □