Democratizing the Parliament*
(Extended Abstract)

Svend Frølund[1] Rachid Guerraoui[2]
[1] Hewlett-Packard Laboratories, Palo Alto, CA 94304
[2] Swiss Federal Institute of Technology, Lausanne, CH 1015

Abstract

This paper presents a replication algorithm that implements a highly-available, non-deterministic state machine. Our algorithm generalizes the Paxos parliament algorithm of Lamport to cope with non-deterministic computations, while preserving its nice resilience and efficiency properties. The algorithm is surprisingly simple, thanks to the use of two powerful underlying abstractions: weak consensus and weak leader election, together with a generic data structure: consensus bag. As a side-effect of our work, we discuss some similarities and differences between replicating deterministic and non-deterministic state machines. Indirectly, we revisit the traditional classification between state-machine replication and primary-backup.

1 Introduction

Motivation. Replication is a well-known software technique to cope with failures in a distributed system. Although intuitive, the idea of replication is not trivial to implement. The difficulty is to give the replicated program's environment the illusion that the program executes on a single, fault-tolerant machine, i.e. the replicas must coordinate their activities to behave consistently as a single highly-available copy. Lamport presented in [Lam89] an algorithm (the Paxos part-time parliament) that ensures replica consistency even if the network is completely asynchronous and an arbitrary number of the processes may crash. Whenever the system stabilizes (communication delays and relative process speeds become bounded) and a majority of the processes remain up for sufficiently long, the protocol ensures fast progress. Thanks to its nice resilience and efficiency properties, the Paxos algorithm was claimed to be very useful in practice [Lam96]. However, Paxos assumes that the replicated program is a deterministic state-machine. In practice, most critical programs that we want to be highly-available are non-deterministic. These include multi-threaded web servers and middle-tier applications that access third-party servers.

The following question naturally comes to mind: can we devise a Paxos-like algorithm that consistently replicates non-deterministic state machines? In fact, Paxos already deals with the non-determinism of the underlying network. Through a *leader-follower* pattern, replicas agree on the same total order for executing requests [Lam89]. The fact that replicated programs are deterministic state machines ensures that, after executing any given request, the replicas end up with the same result (final state and

*Paxons, the citizens of the Paxos island, had a very sophisticated parliament protocol that enabled their legislators to pass decrees despite their frequent forays from the chamber and the forgetfulness of their messengers. Recent archaelogical studies revealed that Paxons could adapt their parliament protocol to a complete democratization of their society. Paxons were able to submit their wishes to the legislators, who had to translate those wishes into decrees and then pass them through the parliament. The legislators maintained copies of the parliamentary records, despite the non-determinism of the translation procedure, i.e., even if every legislator had her own way of translating citizen wishes into decrees.

reply). Obviously, ordering requests would not be enough with non-deterministic state machines. One might think of adding an agreement phase to Paxos, in order to have the replicas agree (a posteriori) on the same result. This agreement phase would however introduce a significant overhead. Alternatively, one might rely on a strict notion of leader to resolve the agreement, assuming an underlying synchronous system model (e.g., [BMST93]). As we will point out below, the resulting algorithm would however not be very resilient.

A more challenging question consequently follows: can we devise an algorithm that copes with non-deterministic actions, while preserving the nice resilience and efficiency properties of Paxos?

Contribution. This paper shows that the answer to this question is *yes*. We present a surprisingly simple algorithm that generalizes Paxos to non-deterministic state machines, while preserving its nice resilience and efficiency flavors.

Roughly speaking, our algorithm addresses, at the *same* time and using the *same* leader-follower pattern, the non-determinism due to concurrent requests and the non-determinism of the state machine. We use an optimistic scheme to agree on the total order of requests: a replica makes an informed guess about where a request fits in the total order, and uses an agreement phase *after* computing a result to both certify the estimated order and enforce the result. If this agreement fails, we simply retry the computation.

The simplicity of our algorithm lies in the use of two powerful underlying abstractions: *weak leader election* and *weak consensus*. Interestingly, each abstraction factors out a specific typical assumption in replication algorithms. Weak consensus factors out the assumption of a majority of correct processes (or more generally the assumption of a correct quorum) and weak leader election factors out the assumption of a minimal level of synchrony needed to solve any form of agreement (or more abstractly the assumption of a minimal knowledge about failures [CHT96]). We manipulate instances of the weak-consensus abstraction through a generic data structure: a *consensus bag*. This data structure allows us to simultaneously agree on the request ordering and on the results.

We prove that our algorithm implements an x-able service [FG00] and ensures one-copy semantics even in the presence of multiple clients. That is, the algorithm implements a service that, besides its high-availability, behaves with respect to its environment as if it were executing on a single process.

By focusing on *what* to replicate, instead of *how* to do it, we revisit the traditional classification between *primary-backup* (also called *passive replication*) [BMST93] and *deterministic state-machine replication* (also called *active* replication) [Sch93]. The difficulty in replication indeed has to do with non-determinism, and this is typically dealt with through a leader-follower pattern. Primary-backup makes this pattern explicit. State-machine replication usually hides this pattern in an underlying total-order broadcast abstraction [Sch93]. By using *finer grained* abstractions (*weak leader election* and *weak consensus*), we point out the very fact that *underlying every replication scheme, there is a leader-follower pattern waiting to come out.*

Related Work. To our knowledge, the only precise and formally proven algorithm that replicates a non-deterministic state machine was given in [BMST93]. The correctness specification relies on a synchronous model and explicitly contains the assumption that there can only be at most one single leader at any point in time. If synchrony assumptions are violated (e.g., during instability periods of the system where multiple leaders are elected), the algorithm might violate replica consistency. The very same issue holds with the *semi-active* replication algorithm given in [Pow91].

In [SM94], a "simulation" of the single-leader notion is suggested, with the main motivation of building a replication protocol that deals with non-deterministic computation in a non-synchronous model. The proposed solution relies on a group membership abstraction with, as

pointed out by the authors, the risk of ending up with an empty group during instability periods of the system. This undesirable behavior, detailed in [ACBMT95, VKCD99], may also occur in the *coordinator-cohort* scheme of [Bir86]. In [DSS98], the group membership abstraction is replaced with a consensus abstraction, precisely to circumvent that behavior. Roughly speaking, consensus is used to prevent any disagreement on the result of a request, in case multiple leaders are elected. Nevertheless, building a leader-follower scheme on top of consensus hampers its performance for two main reasons: (1) the duplication of the leader election mechanism and (2) the obligation for all correct processes to propose an initial consensus value. The first situation happens whenever the replication leader does not coincide with the leader (coordinator) of the underlying consensus protocol: this typically adds communication steps and messages. The second situation is related to the specification of consensus [FLP85]: all processes are supposed to propose a value, which means that even if the leader of the replication scheme is correct (and not suspected to have crashed), all followers must also compute the request in order to propose the result to the consensus abstraction. These issues are circumvented in [DS00] through a new specification of the consensus abstraction that exposes the underlying failure detection mechanism and relies on higher-order functions.

Rather than specify correctness in terms of what goes on inside a replicated service (for example, that the service has a single leader as in [BMST93]), we use x-ability [FG00] as a correctness condition for our replication algorithm. X-ability specifies correctness in terms of how the service interacts with its environment (e.g., clients and third-party entities). This external view of replication correctness allows us to reason about correctness of a replication algorithm that deals with non-deterministic computation, without inherently requiring a strict notion of single leader. Our leader election notion is strictly weaker than the traditional one [SM95] in the sense that it does not preclude the existence of concurrent leaders for an arbitrary period of time. This is precisely what enables us to preserve the resilience flavor of Paxos: replica consistency is always ensured, even during instability periods of the system (e.g., even if multiple leaders are elected), and progress is fastly achieved whenever the system stabilizes. Our consensus notion is also strictly weaker than the traditional one [FLP85] in the sense that it only ensures agreement in the absence of concurrency. In a sense, we extract leader election from consensus to make it a first class citizen, along the lines of Paxos [PLL97]: this is precisely what enables us to preserve its efficiency flavor.

Roadmap. Section 2 defines our system model, Section 3 introduces our abstractions, Section 4 presents our replication algorithm and Section 5 concludes the paper with some final remarks. For space limitations, we do not recall the details of the x-ability theory [FG00], and we only sketch the correctness proofs of our algorithms (more details are given in the optional appendix).

2 Model

We represent a distributed system as a finite set of processes Π. Processes fail by crashing—we do not deal with Byzantine failures, nor do we assume that processes recover after a crash. (We come back to this in Section 5.) A process is correct if it does not fail.

Processes communicate by message passing. A message can be sent by the primitive send and received by the primitive receive. Message passing is one-way, asynchronous, and reliable in the following sense: (*termination*) if a correct process sends a message to a correct process, the message is eventually received, (*no duplication*) each message is received at most once, and (*integrity*) the network does not create nor corrupt messages.

3 Abstractions

We describe below two abstractions that underly our replication algorithm. The first abstraction is *weak leader election* and the second is *weak consensus*. We also introduce a (local) data structure, called a *consensus bag*, which processes use to manage multiple instances of the weak consensus abstraction.

3.1 Weak Leader Election

We describe here the abstraction of a shared object that provides the guarantee of eventually electing a *unique* and *correct* leader. The leader-election abstraction has one operation *leader()*. This operation does not take any input parameter. It returns an output parameter, which is a process identity. When p_i invokes *leader()* and gets p_j as an output at some time t, we say that p_i *elects* p_j at t. We also say that p_j is *leader* (for p_i) at time t. We define the semantics of our leader election abstraction through the following properties.

- AGREEMENT: There is a time after which no two correct processes elect two different leaders.
- VALIDITY: There is a time after which every leader is correct.
- TERMINATION: After a process invokes *leader()*, either the process crashes or it eventually returns from the invocation.

It is easy to see that our specification does not preclude the existence of concurrent leaders for arbitrary periods of time: hence the notion of *eventual* leader election. Our abstraction corresponds to the failure detector Ω introduced in [CHT96]. This failure detector outputs, at very process, a "trusted" process (i.e., a process that is trusted to be up) and ensures that, eventually, all correct processes trust the very same correct process.[1]

[1]Since our abstractions are inspired by Paxos, we follow the terminology of [Lam86], i.e, we use the term leader election instead of failure detector.

3.2 Weak Consensus

Basically, whereas consensus [FLP85] always ensures agreement, weak consensus ensures agreement only in the absence of "concurrent proposals." Roughly speaking, weak consensus looks like consensus when there is no concurrency (just like a regular register looks like an atomic register where there is no concurrency [Lam86]).

Our weak consensus object has one operation: *propose()*. The operation takes as an input parameter a value v (we say that the process proposes v) and returns as an output parameter a value v' (we say that the process decides v'). We define the semantics of our weak consensus abstraction through the following properties.

- AGREEMENT: If a process p_i proposes a value v_i and decides v_i, and a process p_j proposes a value v_j and decides v_j, then $v_i = v_j$.
- VALIDITY: If a process decides a value v, then some process has proposed v.
- TERMINATION: After a process invokes *propose()*, either the process crashes or it eventually returns from the invocation.

We say that a process *commits* a value v when the process proposes and decides v. Agreement means here that no two processes can commit different values. Just like in traditional consensus, validity means that a value that is decided must have been proposed.

In Figure 1, we give a simple implementation of weak consensus with a majority of correct processes.[2] We actually implement here a *generic* weak consensus: the type of the values proposed (and decided) is denoted by $ConsT$ and we assume that $nil \notin ConsT$. The idea of the algorithm is the following (we prove its correctness in

[2]Obviously, consensus is strictly harder than weak consensus in the sense that the assumption of a majority of correct processes is not sufficient to implement consensus in an asynchronous system [FLP85]. Intuitively, consensus = weak consensus + weak leader election. Just like in [Lam86], we extract the leader election of consensus to make it a first class abstraction in the replication algorithm.

```
class WeakCons<ConsT> {
  estimate := nil;

  // Process p_i proposes a value:
  ConsT propose(v) {
    send [Propose,v] to all;
    wait until received |(n+1)/2| [Reply,r-val];
    if any r-val != v then
      return r-val;
    else return v;
  }

  // Process p_j receives a proposition
  when receive [Propose,r-val] from p_j:
    if estimate == nil then estimate := r-val;
    send [Reply,estimate] to p_j;
  }
}
```

Figure 1: Implementing weak consensus

optional Appendix D). Every process p_i maintains a copy of the weak consensus object value, i.e., a local variable `estimate`. The initial value of that variable at every process is set to *nil*. When a process p_i proposes a value v, p_i sends to all processes the message `[Propose,v]`. Process p_i waits to receive a majority of replies. If p_i receives a value v' that is not equal to v, then p_i decides v', i.e., returns v'. (Process p_i is free to choose arbitrarily from any value v' different than v, if it indeed receives such a value.) Otherwise, p_i decides v (i.e., p_i commits v).

When a process p_i receives a message `[Propose,v]` from some process p_j, p_i updates `estimate` with v if `estimate` was equal to *nil* (we say that p_i adopts p_j's proposition). Otherwise, p_i simply ignores v. In both cases, p_i sends back `estimate` to p_j.

3.3 Consensus Bags

Our replication algorithm uses series of weak consensus instances. In stable periods of the system, only one weak consensus is typically needed to ensure agreement on both the order and the result for a given request.[3] Otherwise (if the system is not stable), the replicas might need several weak consensus instances to commit a specific result for a given request. To gather weak consensus instances, we introduce a specific data structure: *consensus bag*.

[3] And this can be performed in a very effective manner as discussed in Section 5.

```
type RequestNumber: IndexType {
  Int num; Request req;

  Boolean operator==(RequestNumber r1,r2) {
    return (r1.num == r2.num) or
      (r1.req == r2.req);
    }
  ... // Constructors, etc.
}
```

Figure 2: The index type for our algorithm

A consensus bag has an operation `lookup` that takes the "name" of a consensus instance, and returns that instance. Processes can then access the same consensus instance and use it to agree on a result. A name is a pair, which contains an index type and an integer. The index type identifies a given request. The integer component of a name allows us to create a series of consensus instances that are all related to the same request. Having a series instead of a single consensus instance per request is necessary to handle concurrent proposals from different processes and still end up with a unique result.

In our algorithm, we use the `RequestNumber` index type shown in Figure 2. The `RequestNumber` index type not only identifies a request, it also identifies a number for this request. The index type reflects the need to establish a global total order of requests, and the number reflects the sequence number in this total order. We define equality for request numbers in such a way that a consensus bag returns the same series of consensus instances for two request-number pairs if *either* the requests are the same *or* the numbers are the same. With this notion of equality, we ensure that we get a unique result for a given "slot" in the total order of requests *and* for a given request—since we want such two-dimensional uniqueness, we need two-dimensional equality inside of the index

```
class ConsBag<IndT,ConsT> {

  ConsInst lookup(IndT i,Int j) {
    // If the pair (i,j) has not been accessed,
    // create a weak consensus instance for
    // that name;
    // Return the instance for (i,j);
  }

  ConsT clean(IndT ind,Int inx,Int l-inx) {
    ConsT l-c, c := ⊥;
    for Int i := inx to l-inx do {
      c := lookup(ind,i).propose(c);
      if  c ≠ ⊥ then l-c := c;
      }
    return(l-c);
    }
}
```

Figure 3: Consensus Bag

type.

We illustrate the basic functionality of consensus bags in Figure 3. A consensus bag is a generic class with two parameters: `IndT` and `ConsT`. `IndT` is the index type whereas `ConsT` is the value type for the weak consensus instances stored in the bag (we assume that the type `ConsT` contains the special element $\perp$). The `lookup` method returns a consensus instance (an object of type `ConsInst`). The method creates these instances in a lazy manner, i.e., when they are first accessed. The `clean` method captures a pattern that we use in our algorithm: processes try to prevent concurrent processes from committing a value by proposing a specific value $\perp$ in a number of consensus instances. The `clean` method takes a value `ind` of the index type. This value gives rise to a series of consensus instances. The clean method iterates over an interval in this series—the interval is defined by the parameters `inx` and `l-inx`. The method proposes $\perp$ in every consensus instance in that interval, and returns either $\perp$, or the "latest" value different from $\perp$. In a sense, a process that invokes that function tries to clean a portion of the consensus bag.

4 Replication Algorithm

We distinguish here between two categories of processes: clients and replicas. We assume a bag of weak consensus instances used by the replicas. We assume that both clients and replicas have access to the weak leader election object. The function *leader*() of this object outputs the identity of a replica (with the properties given in Section 3.1).

We first informally define below the kind of actions executed by our state machines. Then we give an overview of our algorithm and we describe its pseudo-code. Finally we discuss its correctness.[4]

4.1 State Machines

Each replica has a copy of the same state machine. A state machine exports a number of actions. An action takes an input value and produces an output value. In addition, an action may modify the internal state of its state machine, and it may communicate with external entities. We assume that actions are idempotent: although we invoke an action n times, its effect (e.g. state update) appears to happen only once (e.g. because the action only updates local state and performs duplicate elimination based on request identifiers). Moreover, actions may be nondeterministic: the effect and output value of an action may not be the same each time we execute it, even if we execute it in the same initial state and with the same input value. An action that updates local state only, and performs this update under (local) duplicate elimination, is idempotent and may be non-deterministic. The action is idempotent because no individual state-machine copy is updated more than once; the action can be non-deterministic because the duplicate elimination scheme does not prevent multiple state-machine copies from being updated once each.

[4]Due to space limitations, we relegate many details to the optional appendix. In particular, the appendix (1) uses the theory of [FG00] to formally define the type of actions we consider, (2) formally recalls what it takes for a service to be x-able [FG00], and (3) we proves the correctness of our algorithm.

A state machine has a function, called `execute` that takes a request and executes the corresponding action. The `execute` function returns the reply of the action execution. The execution of an action may fail (for example if the action manipulates a remote database and the database crashes). If the action fails, the `execute` function returns an error. Otherwise, we say that the action executes successfully. We assume that there is a time after which `execute` never returns an error. A state machine also has an attribute, called `state`, that allows us to read and update the state of the state machine.

4.2 Overview

To provide clients with the illusion of a single state machine copy that does not fail, our replication algorithm addresses the following issues:

- *Total order.* State updates are coordinated so that subsequent requests are processed in the context of previous requests. That is, if a client submits a request req_1 after a request req_2, then the state update performed by req_1 should be visible to req_2 even if the two requests are processed by different replicas. The requirement of single-copy semantics implies that the replication algorithm has to establish a global total order for the requests.

- *Deterministic replies.* To handle failures, the same request may be executed multiple times by different replicas. However, to give clients the illusion that no failures occur, the replicas agree on the state produced by any request in the total order.

The idea of the algorithm is the following. Every request has a "row" of consensus instances in a consensus bag. The replicas have to agree on which request has which row (the total order of requests) and the replicas have to agree on the result for a given request. To facilitate these agreements, each replica owns a "column" of consensus instances in a consensus bag. With n replicas, a replica process p_i owns column number i, $i+n$, $i+2n$, and so on. Here, ownership means the following: if process p_i is leader, and has a request r to process, p_i will try to find r's row and commit a result in that row. However, p_i will only propose a result value for the instances it owns, e.g., at column i. To prevent the situation where different replicas are leaders and commit different results, p_i tries first to "clean" every consensus instance in a candidate row with column index smaller than i. Replica p_i cleans a consensus instance by proposing a distinct value $\perp$.

Assume that a replica p_i believes row num to be the row for a request r, and assume that p_i is trying to clean a consensus instance in row number num. There are two reasons why replica p_i may not be able to clean the consensus instance: (1) row num is already used for another request or (2) row num is in fact the row for r, but some other replica has already proposed a result for r in row num. In case (1), replica p_i tries another row (e.g. $num+1$). Before continuing with the next row however, replica p_i installs the state from row num into its state-machine copy. In case (2), replica p_i considers the proposed result as its own. If p_i proposes a result in consensus i but does not commit it (e.g. because some other replica has already committed $\perp$ in consensus i), p_i tries again with a consensus instance at row $i+n$, and so forth. If p_i commits a result at some row num, then num is *the* row for the request, and the computed result is *the* result for the request.

The stylized use of consensus bags prevents any disagreement, even in the presence of concurrency (which weak consensus by itself does not ensure). The leader election protocol abstraction that eventually, only one replica keeps trying, and if no result has been committed, this replica will reach an "empty" weak consensus instance it owns and will commit a result.

4.3 Pseudo-Code

Our replication algorithm has a client part and a replica part, which are shown in Figure 4.

A client process has a `submit` function that it calls to send a request to the replicated service. The `submit` function takes a request (the name of

an action and an input value) and returns a reply (an output value). From the client's point of view, the `submit` function is a "wrapper" of the server-side state machine that encapsulates the fact that the state machine is replicated. The `submit` function embeds the given request in a message, and sends this message to the leader replica. The function then waits for a reply message. If the leader changes and no reply was returned from the former leader, the request message is simply sent to the new leader.

The server-side behavior is triggered by the reception of a request message. If a replica is not a leader (as determined by the `leader()` function), it simply ignores the request message. If a replica is a leader, it enters a `while` loop to process the request. The variable `num` contains the first row that replica p_i believes to be empty.

The consensus bag at each replica is parameterized by two types: `RequestNumber` and `Outcome`. We already introduced the `RequestNumber` type in Section 3.3. The `Outcome` type is a product type: values of type `Outcome` are tuples that contain a request, a reply, and a state. Each replica will have a consensus bag as illustrated in Figure 5. The consensus bag in the figure could be the bag of replica number 3 in a system of 3 replicas. A consensus bag maps values of the index type to a series of consensus instances. Each row in the figure shows the mapping for a particular value of the index type; in our case these values are request-number pairs. In the figure, `req1` is a request and `v1` is a value (or result) that is committed to a consensus instance.

4.4 Correctness

Our replication algorithm implements an x-able service [FG00]: i.e., a service that provides the illusion to its environment that it is executing every action (request) *exactly-once*. Here, we give some intuition about about the properties that characterize an x-able service.

In short, X-ability covers (1) state consistency, (2) action execution and (3) reply validity. (1) State consistency is concerned with the internal state of state machines (as opposed to the external state of third-party entities, such as databases

```
behavior Client {
  Replica p_i;

  Reply submit(Request req) {
    Reply rep;
    while true {
      p_i := leader();
      send [Request,req] to p_i;
      await (receive [Reply,rep]) or
        p_i != leader();
      if(received [Reply,rep]) then
        return rep;
      }
    }
}

behavior Replica {// Algorithm for p_i
  type [Request,State,Reply] Outcome;
  ConsBag<RequestNumber,Outcome> bag;
  Int l-inx, inx, num := 1;
  Request req;
  State-machine S;
  Outcome out;
  ConsInst obj;

  when receive [Request,req] from client:
    inx := 0; l-inx := i;
    while(leader() == p_i) {
      out := bag.clean([num,req],inx,l-inx);
      if out != ⊥ then
        if req == out.req then
          send [Reply,out.rep] to client
          return;
        else
          S.state := out.state;
          num++;
          continue;
      else
        while true {
          try rep := S.execute(req);
          catch(error) continue;
          }
      obj := bag.lookup([num,req],l-inx);
      out := obj.propose(req,S.state,rep);
      if out != ⊥ then
        send [Reply,out.rep] to client;
        num++;
        return;
      else
        l-inx := l-inx + n;
        inx := l-inx + 1;
      }
}
```

Figure 4: Our replication algorithm

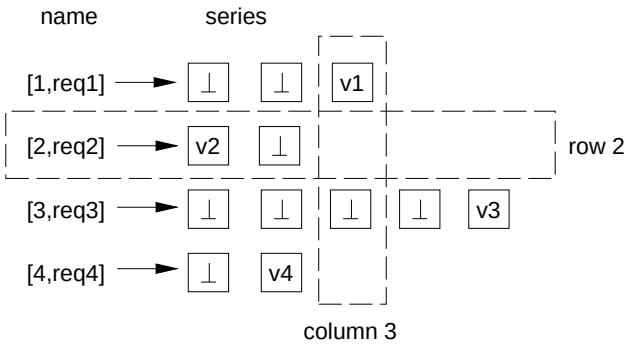

Figure 5: A possible consensus bag for replica 3 in an ensemble of 3 replicas

or other services). With x-ability, we characterize state consistency in terms of what clients observe, which are the replies returned to clients. We capture state consistency by insisting that replies should appear as if they were produced by a single state machine copy. Proving state consistency for our algorithm is not straightforward. We essentially have to prove that our scheme for checkpointing a state, storing it in the consensus bag as part of a result, and installing it in other state-machine copies preserves the illusion of a single-copy state machine. (2) The actions executed for each request should have a combined effect that is exactly once. More precisely, when multiple executions of the same action take place, these should appear as if only a single execution took place. For the kind of actions we consider (e.g., updating a state), this requirement is relatively straightforward to satisfy. Finally, (3) reply validity in general prevents the algorithm from "inventing" replies, and requires the algorithm to eventually terminate. For our algorithm, the termination property is primarily due to the termination property of the leader election abstraction.

5 Discussion

We discuss here the resilience and performance of our algorithm, with respect to Paxos [Lam89]. We also point out some fundamental similarities and differences between replicating deterministic and non-deterministic state machines.

5.1 Resilience Assumptions

The correctness of our algorithm relies on the assumption of a crash-stop system model where channels are reliable, a majority of the processes are correct (do not crash), and there is a time after which there is exactly one leader process (that is furthermore correct). This model makes it easy to describe our abstraction implementations. In a companion paper [BDFG01], we give optimized implementations of our abstractions in various practical crash-recovery models.[5] Thanks to the modularity of our approach, one can easily use those implementations to adapt our replication algorithm to various crash-recovery system models.

Basically, the processes would need to (1) retransmit messages in order to cope with a temporary crash of the channels and (2) log (in stable storage) the values of the weak consensus abstraction in order to cope with their own crash and recovery. Just like in Paxos, (a) safety would always be preserved, [6] and (b) liveness would be achieved as soon as a majority of the processes remain up, can reliably communicate, and elect a unique correct leader for "sufficiently long" [Lam89].

5.2 Performance Issues

In practice, most runs of a distributed system are *nice*: processes and channels do not crash and are completely synchronous. In these runs, the leader process, initially elected by default, does not change. Typically, the default leader is p_0 and p_0 can directly compute a request and propose it to weak consensus 0, i.e., p_0 does not need first to go through a "cleaning" phase precisely because there is no consensus to clean. In this scenario, after p_0 receives a request from a client, p_0 needs only one round trip communication step with other replicas before returning the reply to the client. In other words, our algorithm has the same communication pattern as Paxos in nice

[5] The weak consensus specification we consider in this paper is actually simpler and even easier to implement than the consensus specification we considered in [BDFG01].

[6] Remember that processes do not behave maliciously and channels do not create or corrupt messages.

runs. A crash-recovery variant of our algorithm would also have the same number of logs as Paxos in these nice runs.

Lamport has informally described in [Lam89] a clever way to apply the optimal case above whenever the system stabilises, i.e., even in runs that are not nice but simply *eventually* nice. The eventual leader p_k would also only need one round trip communication step with other replicas before returning the result to the client. In our terminology, this basically means that, in stable periods, the leader process p_k would not need to go through a cleaning phase. Intuitively, this is made possible after the leader process p_k asks the processes not to use any consensus with a row lower than k.[7]

5.3 Similarities and Differences

To sum up, we point out here the very fact that there is nothing inherent to non-deterministic state machines that make their consistent replication harder or significantly different than replicating deterministic ones. This somehow contradicts the belief underlying early work on replicating non-deterministic state machines [BMST93], which might have given for instance the impression that a strong notion of leader election (and an underlying synchronous system model) is necessary to replicate non-deterministic state machines.

Roughly speaking, replicating a deterministic state machine requires agreement on some order for the requests, whereas replicating a non-deterministic state machine requires agreement on the order plus agreement on the results of these requests (final states and replies): this paper shows that the agreement can however be the same one. Basically, we combine an optimistic form of agreement on the order with the order on the result. As a consequence, we end up with the same communication pattern for both kinds of replication.

The fact that the nature of the agreements are different have however some important consequences. Indeed, while talking about efficiency, we have focused here on the number of messages, communication steps, and logs needed for the replicas to compute a result corresponding to a given request. Obviously, the size of the messages is not the same: our replication algorithm involves agreement on states and this more expensive than an agreement on integers (on an order). Some optimisations could to be considered however. Typically, one should avoid having a new leader always start from row 0 but periodically log the current state or contact other processes for updates. Similarly, one should employ specific semantic-based techniques to avoid sending out states but only state differences. Nevertheless, there are also some optimisations that one can achieve with deterministic state machines but not with non-deterministic ones. In particular, if the agreement is only on the order, one can gather several requests together and have only one agreement for these requests. This does not seem to be possible with non-deterministic state machines since the agreement should hold for every state.

[7] Roughly speaking, p_k reconfigures the system so that it can play the role of p_0 in a nice run.

References

[ACBMT95] E. Anceaume, B. Charron-Bost, Pascale Minet, and S. Toueg. On the formal specification of group membership services. Technical Report 95-1534, Department of Computer Science, Cornell University, August 1995.

[BDFG01] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing the parliament. Technical Report EPFL-2001, Swiss Federal Institute of Technology, January 2001.

[Bir86] K. P. Birman. Isis: A system for fault-tolerant distributed computing. Technical Report TR86-744, Department of Computer Science, Cornell University, April 1986.

[BMST93] N. Budhiraja, K. Marzullo, F. B. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, 1993.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector

to solve consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[DS00] X. Défago and A. Schiper. Semi-passive replication and lazy consensus. Technical Report DSC/2000/012, Swiss Federal Institute of Technology, February 2000.

[DSS98] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, October 1998.

[FG00] S. Frølund and R. Guerraoui. X-ability: A theory of replication. In *Proceedings of the Symposium on Principles of Distributed Computing*. ACM, 2000.

[FLP85] M. Fisher, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[Lam86] L. Lamport. On interprocess communication. *Distributed computing*, 1(1):77–101, 1986.

[Lam89] L. Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, 1989.

[Lam96] B. Lampson. How to build a highly available system using consensus. In *Proceedings of the International Workshop on Distributed Algorithms, Springer-Verlag, LNCS (WDAG)*, September 1996.

[PLL97] R. De Prisco, B. Lampson, and N. Lynch. Revisiting paxos. In *Proceedings of the International Workshop on Distributed Algorithms, Springer-Verlag, LNCS (WDAG)*, September 1997.

[Pow91] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer Verlag, ESPRIT Research Reports, volume 1 edition, 1991.

[Sch93] F. B. Schneider. Replication management using the state machine approach. In S. Mullender, editor, *Distributed Systems*. Addison-Wesley, 1993.

[SM94] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 138–147, October 1994.

[SM95] L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. Technical Report TR95-1488, Dept. of Computer Science, Cornell University, 1995.

[VKCD99] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group communication specifications: A comprehensive study. Technical Report MIT-LCS-TR-790, Laboratory for Computer Science, Massachusetts Institute of Technology, April 1999.

A X-ability

We give in this appendix a brief reminder of the main ideas behind x-ability [FG00], which we extend with a notion of *atomic actions*.

A.1 Basic Sets

We first introduce the basic elements that we consider in defining x-ability. The set Action contains the name of actions; the sets Input and Output contain input and output values for actions. Both input and output values are structured entities: besides an "application-level" value (an element in the set Value), they also contain an identifier (an element in Identifier), which distinguishes different invocations of the same action on the same logical application-level value. We use iv to refer to an input value in Input, and we use ov to refer to an output value in Output. Furthermore, we identify two sets, Request and Reply. A reply is an output whereas a request is a pair that contains an action name and an input value. We write pairs as "(a, iv)" (this pair contains the action name a and the value iv). These sets are more precisely defined as follows.

$$\begin{aligned} \mathsf{Input} &= (\mathsf{Identifier} \times \mathsf{Value}) \\ \mathsf{Output} &= (\mathsf{Identifier} \times \mathsf{Value}) \\ \mathsf{Request} &= (\mathsf{Action} \times \mathsf{Input}) \\ \mathsf{Reply} &= \mathsf{Output} \end{aligned}$$

Usually, we treat input and output values as opaque values and do not explicitly refer to constituent values from Identifier and Value. In particular, we assume that a state machine action takes a value in Input and produces a value in Output. That is, the action ignores the Identifier component in input values during execution and returns an output value with the same identifier as the input value.

We use the following expressions to explicitly refer to the constituent of input and output values: $iv.\mathsf{val}$ returns the value component and $iv.\mathsf{id}$ returns the identifier component (we use similar operators for output values).

A.2 Events and Histories

We use event histories to specify x-ability. We consider two kinds of events: start events and completion events. A start event denotes the invocation of a state machine action on a particular input value. A completion event captures the successful completion of an action with a particular output value. We use the following notation for events:

$$e \quad ::= \quad S(a, iv) \mid C(a, ov)$$

The event $S(a, iv)$ captures the start of executing the action a with iv as argument. The event $C(a, ov)$ captures the completion of executing the action a, and ov is the output value produced by the action.

A history is a sequence of events. We use the following notation for histories:

$$h \ ::= \ \Lambda \mid e_1 \ldots e_n \mid h_1 \bullet \ldots \bullet h_n$$

The symbol Λ denotes the empty history—a history with no events. The history $e_1 \ldots e_n$ contains the events e_1 through e_n. The history $h_1 \bullet \ldots \bullet h_n$ is the concatenation of histories h_1 through h_n. The semantics of concatenating histories is to concatenate the corresponding event sequences.

We say that an event e *appears* in a history h if h contains e. We write this as $e \in h$. We refer to the complement of $\in$ as $\notin$.

A.3 History Equivalence

With x-ability, we define what it means for a history of action invocations to have exactly-once effect. The idea is to (1) define a notion of history equivalence (based on equivalent effects), (2) define a notion of failure-free history (histories that have exactly-once effect per definition), and (3) define x-able histories as those that are equivalent to a failure-free history.

Our equivalence relation for histories is based on so-called re-write rules for histories. We write this as $h \Rightarrow_x h'$ (h can be re-written to h', which also means that h and h' are equivalent).

The algorithm in Figure 4 replicates state machines with idempotent actions (though they are non-deterministic). In [FG00], we give re-write rules for idempotent actions. Essentially, a history with n invocations of an idempotent action is equivalent to a history with a single invocation, as long as both histories contain a successful invocation of the action.

The actions we consider here are also atomic: their effect either happens completely or not at all. This means that executing an action will never leave the system in an inconsistent state. Even if the action fails during its execution, no partial updates will be visible. An action that only updates the internal state of its state machine is typically both idempotent and atomic. Updating the internal state is idempotent: the action can perform duplicate elimination based on request identifiers. Moreover, an internal state update fails only if the process itself fails (after all, the process is the only entity involved in the state update). This property ensures atomicity: if the state update fails during the action execution, the process itself failed and the partial state update will never be visible to other processes.

In Figure 6, we use re-write rules to define a notion of atomic action. The rules capture the following idea: although we only observe the start event for an action a (we do not observe the actions completion event), the effect is equivalent to one of two possible failure-free histories. One failure-free history is one in which the action did not occur at all. That is, we can remove the start event. The other failure-free history is one in which the action happened successfully. To obtain this latter action, we add the completion event.

A.4 X-able Histories

We define x-able histories as the ones that are equivalent (under $\Rightarrow_x$) to a failure-free history. A failure-free history is a history that could have been produced by a failure-free execution of a single state-machine action. To define the notion of failure-free history, we define a function, called eventsof, on actions and their values. The eventsof function returns the failure-free history associated with an action and its values.

$$\mathsf{eventsof}(a^i, iv, ov) = S(a^i, iv)C(a^i, ov)$$

Due to non-determinism, there are multiple failure-free histories which are possible for a given action a and a given input value iv. We define the set of all possible histories, $\mathsf{FailureFree}_{(a,iv)}$, as follows:

$$\begin{aligned}&\mathsf{FailureFree}_{(a,iv)} = \{h \in \mathsf{History} \mid \\ &\quad \exists\, ov \in \mathsf{Result}, \exists\, iv' \in \mathsf{Input} : \\ &\qquad h = \mathsf{eventsof}(a, iv', ov) \wedge iv.\mathsf{val} = iv'.\mathsf{val}\}\end{aligned}$$

An x-able history is one that satisfies the predicate x-able on histories:

$$\mathsf{x\text{-}able}_{(a,iv)}(h) = \begin{cases}\mathsf{true} & \text{if } \exists\, h' \in \mathsf{FailureFree}_{(a,iv)} : h \Rightarrow_x h' \\ \mathsf{false} & \text{otherwise}\end{cases}$$

We use x-able histories to define the contract a service has with its environment. Insisting on histories being x-able forces the service to have a side-effect that appears to be exactly-once. In defining x-ability, we also address the relationship between clients and services. Intuitively, clients should see the service as a single-copy state machine. We recapitulate our formalization of single-copy semantics in the following sub-section.

A.5 Single-Copy Consistency

The reply value given to a client in response to a request must be the value returned from the server-side state machine when the service processes the request. In other words, the service should not be allowed to "invent" a reply value and pass it back to the client. Moreover, the service should not be allowed to invent requests, it should only process the requests sent by clients.

We use the server-side history to define the constraints for requests and replies. The history contains a request value as part of start events and

$$\frac{\forall\, C(a, ov) \in h_2 : iv.\mathsf{id} \neq ov.\mathsf{id}}{h_1 \bullet (S(a, iv)) \bullet h_2 \;\Rightarrow_x\; h_1 \bullet h_2} \quad (1)$$

$$\frac{\forall\, C(a, ov) \in h_2 : iv.\mathsf{id} \neq ov.\mathsf{id} \qquad ov' \in \{v \in \mathsf{Result} \mid ov'.\mathsf{id} = iv.\mathsf{id}\}}{h_1 \bullet (S(a, iv)) \bullet h_2 \;\Rightarrow_x\; h_1 \bullet (S(a, iv)C(a, ov')) \bullet h_2} \quad (2)$$

Figure 6: Definition of atomic actions

reply values as part of completion events. We introduce the notion of a history signature, which defines the client-side information (request and result) that is legal relative to a given server-side history. Because of non-determinism and server-side retry, a history can have multiple signatures.

The set Signature is the set of all possible history signatures. We define the set of signatures for a given history h as follows:

$$\mathsf{Signature} = (\mathsf{Action} \times \mathsf{Input} \times \mathsf{Output})$$

$$\begin{aligned}\mathsf{signature}(h) = \{(a, iv, ov) \in \mathsf{Signature} \mid \\ \exists\, ov' \in \mathsf{Result}, \exists\, iv' \in \mathsf{Input} : \\ h \Rightarrow_x \mathsf{eventsof}(a, iv', ov') \text{ where} \\ iv.\mathsf{val} = iv'.\mathsf{val} \wedge ov.\mathsf{val} = ov'.\mathsf{val}\}\end{aligned}$$

The basic idea is that the client's view should correspond to the actual server-side effect. However, we have to define the server-side effect in terms of parameter values and ignore the identifiers associated with input and output values.

If a client submits a sequence of requests, one after the other, later requests should be processed in the context of earlier requests. This consistency requirement prevents a service from "forgetting" updates to its state.

To prevent a service from forgetting the effect of previous requests, we want to characterize the set of possible reply values for a given request. Since we do not know what state-machine actions do, we cannot describe which specific values are possible. Instead, we assume the existence of a set PossibleReply that contains the possible reply values for a given request. To capture the history-sensitive nature of the set of possible replies, we define PossibleReply in the context of a request sequence $r_1 \ldots r_n$. The interpretation of PossibleReply in the context of a sequence is the set of possible replies to request r_n after the state machine has executed the requests $r_1 \ldots r_{n-1}$ one after the other. Thus, we write the set as: $\mathsf{PossibleReply}_{(r_1 \ldots r_n)}$.

Notice that the set PossibleReply is defined for state machines, not replicated services. Thus, there is no notion of failures or replication involved in its definition. The set is well-defined for state machines in general.

A.6 X-able Services

We can now describe the requirements for a service to be x-able. Given a set of replicas that all have a copy of a state machine S. Given a client with a *submit* action that takes a request and returns a reply. The replicas constitute an x-able service if the following properties are satisfied:

- CONSISTENCY: The action *submit* is idempotent.
- TERMINATION: There is a time after which *submit* always executes successfully.
- EFFECT: If the client submits a request (a, iv), then the server-side history for (a, iv) is either empty or it satisfies $\mathsf{x\text{-}able}_{(a,iv)}$.
- RESULT: If the client receives a reply ov in response to a request (a, iv), and if the server-side history for executing this request is h, then $(a, iv, ov) \in \mathsf{signature}(h)$.

- STATE: If the client successfully submits a sequence of requests, $r_1 \ldots r_n$, and receives the reply ov in response to r_n, then ov is in $\mathsf{PossibleReply}_{(r_1 \ldots r_n)}$.

B Proof of X-ability

We prove that the algorithm in Figure 4 implements an x-able service for state machines with idempotent atomic actions.

Proposition 1 (CONSISTENCY) *The action submit is idempotent.*

PROOF (SKETCH): Consider a client-side history from invoking the *submit* action on a request *req* n times. We have to show that, if one of these n invocations is successful, then the effect of this n-invocation history is equivalent to the effect of a history with a single successful invocation. The effect we consider is the effect of executing state-machine actions.

First, observe from the algorithm that the *submit* action only returns successfully for a request *req* if a state-machine copy executed *req* successfully. Because they both contain a successful invocation of the *submit* action on *req*, both the n-invocation and single-invocation histories contain a successful invocation of a state-machine action on *req*. Because state machine actions are idempotent, the resulting server-side histories are both equivalent to a history with a single successful invocation of the state-machine action. □

Proposition 2 (TERMINATION) *There is a time after which* `submit` *always executes successfully.*

PROOF: Assume by contradiction, that the proposition does not hold. By the leader abstraction, there eventually is a unique, correct and perpetual leader. By the reliable channels assumption, the eventual perpetual leader receives the request, and keeps trying to commit that request. Since we assume that eventually, `execute` is always successful, the leader eventually proposes a successful result in an empty weak consensus that was not cleaned by any process and succeeds in deciding that reply and returning it to the client. □

Proposition 3 (EFFECT) *If the client submits a request (a, iv), then the server-side history for (a, iv) is either empty or it satisfies* $\mathsf{x\text{-}able}_{(a,iv)}$.

PROOF (SKETCH): Given a run R of the algorithm on the request (a, iv), and let h be h_R. We show that if h is non-empty, then h satisfies $\mathsf{x\text{-}able}_{(a,iv)}$.

Assume that h is non-empty. Then h contains at least one start event for (a, iv). Consider the last start event for (a, iv) in h. If h also contains a completion event for (a, iv) after this last start event, then we can use the idempotence rules to reduce h to a single successful invocation of (a, iv). If h does not contain a completion event after the last start event, we can use rule (2) to add such a completion event. After adding this event, we can again use the idempotence rules to reduce h to a failure-free history. □

Proposition 4 (RESULT) *If the client receives a reply ov in response to a request (a, iv), and if the server-side history for executing this request is h, then $(a, iv, ov) \in \mathsf{signature}(h)$.*

PROOF: Since the client receives a reply, which is different from $\perp$, we can deduce from the algorithm that h is not the empty history. Thus, by Proposition 3, h satisfies $\mathsf{x\text{-}able}_{(a,iv)}$. This again means that there exists values iv' and ov' such that $h \Rightarrow_x \mathsf{eventsof}(a, iv', ov')$. From the definition of failure-free histories, we know that iv has the same parameter value as iv'. We have to prove that ov has the same parameter value as ov'. This is the case because the algorithm never changes parameter values. Thus, we can now conclude that the signature (a, iv, ov) is in $\mathsf{signature}(h)$. □

To prove the STATE property (Proposition 9), we first need to establish some new terminology and concepts. For a given state machine, we have

already introduced the set of possible replies after executing a sequence of requests—we refer to this set as $\mathsf{PossibleReply}_{(r_1 \ldots r_n)}$. Along the same lines, we can refer to the set of possible states that a state machine can be in after executing a sequence of requests. We write this set of states as $\mathsf{PossibleStates}_{(r_1 \ldots r_n)}$. A state is an element in the set State, and we use the symbol σ to refer to a state.

We want to formalize the idea that if we execute a request in a "legal" state, then we obtain a "legal" reply. In other words, we want to use a notion of action execution to connect the concept of possible states with the concept of possible replies. We formalize action execution through the function execute:

$$\mathsf{execute} : (\mathsf{State} \times \mathsf{State} \times \mathsf{Request}) \rightarrow \mathsf{Result} \quad (3)$$

The execute function satisfies the following axiom:

Axiom 5 *Given a state machine S. If $\sigma \in \mathsf{PossibleStates}_{(r_1 \ldots r_{n-1})}$ for S, then $\mathsf{execute}(S, \sigma, r_n) \in \mathsf{PossibleReply}_{(r_1 \ldots r_n)}$.*

Recall that a process p is said to *commit* a value for a weak consensus instance if p proposes and decides that value. Based on this definition, we also introduce the notion that a replica in our algorithm can commit a result for a given name. Remember that a name is a request-number pair. We say that a replica p commits a result *res* for a name *nam*, if p commits *res* in a weak consensus instance in the series associated with *nam*.

With this definition, we can now prove the following lemma:

Lemma 6 *At most one replica commits a result for a given name.*

PROOF: For a contradiction, assume that two replicas p_1 and p_2 commit two different results res_1 and res_2 for a given name. Because of the agreement property of weak consensus, the two results must be committed by two different consensus instances. Assume that res_1 is committed by instance number i and that res_2 is committed by instance number j. Furthermore, assume without loss of generality that $i < j$.

According to the algorithm, any replica p will only propose a result for consensus instance k if p has successfully cleaned all instances with index smaller than k. Thus, for p_2 to commit res_2, r_2 must clean all instances with index smaller than j. This is a contradiction with the assumption that p_1 commits res_1 for instance i with $i < j$. □

Lemma 7 *Given a run R in which a client successfully submits the requests $r_1 \ldots r_n$. If a replica commits a result for a name nam in R, then nam has the following format: $[i, r_i]$, where $1 \leq i \leq n$.*

PROOF: Replicas construct names from a received request `req` and the variable `num`. The lemma follows from the following properties and the fact that requests are submitted one after the other by a single client:

- A replica only increments the variable `num` to the value $k + 1$ if some replica has committed a value for a name of the form $[k, -]$.
- Given two names $[k, r]$ and $[k, r']$. No replica commits a value for both names.

□

We say that a replica p commits a result for a request *req*, if p commits the result for any name with *req* as component (i.e. names of the form $[-, r]$.

Lemma 8 *Given a run R in which a client successfully submits the requests $r_1 \ldots r_n$. Then a replica commits a result $[r_n, \sigma, rep]$ for r_n in R, and the following holds:*

1. $rep \in \mathsf{PossibleReply}_{(r_1 \ldots r_n)}$.
2. $\sigma \in \mathsf{PossibleStates}_{(r_1 \ldots r_n)}$.

PROOF: If the client successfully submits r_n, the client receives a reply for r_n. A client only

receives a reply for request, if some replica commits a result for that request. Moreover, any result committed for r_n will have r_n as its request component.

We prove (1) and (2) by induction on n (the number of submitted requests).

- $n = 1$: Let p be the replica that commits a result for r_1. Because no other replica commits a result for r_1, p will not execute the state assignment `S.state := out.state`. Moreover, since p commits a result for r_1, p will execute its state machine copy (which is in its initial state) on r_1. Per definition, the reply produced by the execution is in $\mathsf{PossibleReply}_{(r_1)}$ and the state produced is in $\mathsf{PossibleStates}_{(r_1)}$.

- $n = k, k > 1$: Let p be the replica that commits a result for r_n. There are two cases: (a) p commits a result for r_{n-1} or (b) p does not commit a result for r_{n-1}.

  Consider first case (a). When p commits a result for r_{n-1}, its state-machine copy has a state σ that belongs to $\mathsf{PossibleStates}_{(r_1 \ldots r_{n-1})}$. It then increments its `num` variable to n, and waits for another request. If p receives any request in the set $r_1 \ldots r_{n-1}$, then p will obtain the result for this request from the bag, and return the reply to the client. Some replica will have committed a result for any such request, and p will not execute the state assignment "`S.state := out.state`." Thus, when p receives r_n, p's state machine is still in state σ, and p will compute the reply for r_n in this state. By Axiom 5, the computation will produce a result that complies with (1) and (2) above.

  Consider next case (b). Since p commits a result for r_n, p's `num` variable will eventually have the value n. Moreover, since p does not commit a result for r_{n-1}, `num` is incremented from $n-1$ to n as part of the code that contains the state assignment "`S.state := out.state`." In this assignment, the state `out.state` is returned from the `clean` method, which means that some replica committed this state as part of a result. This result was committed for r_{n-1}, and the state in `out.state` belongs to $\mathsf{PossibleStates}_{(r_1 \ldots r_{n-1})}$ by the induction hypothesis. As in case (a), we can now show that replica p computes a result for r_n in this state, and the result complies with (1) and (2) above.

□

Proposition 9 (STATE) *If the client successfully submits $r_1 \ldots r_n$ and receives the reply value ov in response to r_n, then ov is in* $\mathsf{PossibleReply}_{(r_1 \ldots r_n)}$.

PROOF: ¿From Lemma 8, we know that some replica p commits a result for r_n. Moreover, the reply portion of this result belongs to $\mathsf{PossibleReply}_{(r_1 \ldots r_n)}$. The proposition follows from the fact that replicas only return replies from committed results to clients. □

C Handling Multiple Clients

X-ability deliberately does not encompass the issue of concurrent access by multiple clients.[8] We prove that the algorithm in Figure 4 ensures that a multi-client system is equivalent to a single-client system.

To characterize correct handling of multiple clients, we introduce the concept of *delivering* a request. We say that a replica *delivers* a request if either it executes the request, or it installs a corresponding state.

Proposition 10 *Let req_1 and req_2 be any two different requests. If any replica delivers req_1 before req_2, then no replica delivers req_2 without having delivered req_1.*

[8] One of our motivations in defining x-ability [FG00] was precisely to separate correct handling of failures from correct handling of concurrent accesses. By defining x-ability in terms of a single-client system, it is simpler to express the notion that it should appear as if there is a single copy only of the server-side state machine. The specification does not have to account for the possibility that multiple clients share this single copy.

PROOF (SKETCH): Assume by contradiction that some replica p_i delivers req_1 before req_2, whereas some replica p_j delivers req_2 without having delivered req_1. For p_i to deliver req_1 before req_2, the consensus bag must have a number num_1 associated to req_1 and a number num_2 associated to req_1 such that $num_1 < num_2$. Since replicas scour the consensus bag sequentially, starting from number 0, process p_j reaches num_1 before num_2. If num_1 is not associated with any request, then p_j associates req_2 with num_1: a contradiction. Otherwise, p_j computes req_1 before req_2: a contradiction. □

D Weak Consensus Algorithm: Correctness

Proposition 11 *The algorithm of Figure 1 implements weak consensus with a majority of correct processes.*

PROOF (SKETCH): Validity is satisfied since we assume that processes can only fail by crashing and channels do no invent or corrupt messages; any value decided is a value proposed by some process. Termination is satisfied with the assumption of reliable channels and a majority of correct processes. Consider now Agreement. Assume that some process p commits some value v_i and some process q commits some value v_j. Since none of the processes can decide without having a majority adopts its value, and no process adopts more than one value, then $v_i = v_j$. □