An Efficient Universal Construction for Message-Passing Systems (Preliminary Version)*

Partha Dutta[2] Svend Frølund[1] Rachid Guerraoui[2] Bastian Pochon[2]

[1] Hewlett-Packard Laboratories, Palo Alto, CA 94304

[2] Distributed Programming Laboratory, Swiss Federal Institute of Technology, Lausanne, CH 1015

Abstract

A universal construction is an algorithm that transforms any object with a sequential specification into a wait-free and linearizable implementation of that object.

This paper presents a novel universal construction algorithm for a message-passing system with process crash failures. Our algorithm relies on two fine-grained underlying abstractions: a weak form of leader election, and a one-shot form of register.

Our algorithm is *indulgent*, *efficient* and *generic*. Being indulgent intuitively means that the algorithm preserves consistency even if the underlying system is asynchronous for arbitrary periods of time. Compared to other indulgent universal constructions, our algorithm uses fewer messages and gives rise to less work in steady-state. Our algorithm is generic in two senses: (1) it is initially devised for a crash-stop model and can be easily ported to various crash-recovery models, and (2) it is initially optimized for the steady-state period but can easily be extended to trade-off between steady-state performance and fail-over time.

1 Introduction

A universal construction is an algorithm that provides a wait-free and linearizable implementation of any object that has a sequential specification [13]. In short, being wait-free requires the implemented object to be highly available—any invocation of the object must complete in a finite number of steps, even in the presence of failures. Being linearizable intuitively means that the implemented object must remain consistent—the object must appear to be accessed in a sequential manner [15].

It is very appealing to use the notion of universal construction as the theoretical underpinning of highly-available distributed systems. The notion of universal construction clearly and precisely defines the contractual obligations and guarantees of the various players, such as the objects, the algorithm, and the clients. The object can be implemented in any way that complies with its sequential specification. In particular, objects can be non-deterministic. Clients are given precise safety and liveness guarantees. The universal construction algorithm can be based on any implementation that provides clients with wait-free, linearizable access to the object. From a practical point of view, the object represents an online service. A universal construction algorithm can be viewed as middleware that implements highly-available access to the service from a number of clients.

To be practical as the foundation for high-availability middleware, a universal construction algorithm should have a number of desirable properties:

- It should tolerate arbitrary asynchrony periods of the underlying system (we refer to this property as indulgence [12]). Indulgence is important because the service may be subject to unpredictable workloads, and it may share resources, such as network bandwidth, with other online services.
- The steady-state behavior should be efficient. Steady-state is a period where no

*Technical Report EPFL/IC/2002/28

process fails or is suspected to have failed. In most systems, this is the common case, and thus the case for which we want to optimize.

- The algorithm should minimize the communication between clients and the service. It is indeed common for online services to be accessed via the Internet, and such access typically involves communication over wide-area network links.

Universal construction algorithms are typically based on atomic registers, consensus-like primitives, or strong forms of leader election abstractions. As we explain in the following, these pose several problems in the context of high-availability middleware where practical services communicate through message passing.

Traditional universal construction algorithms were devised in a shared memory (e.g., [13]) where processes communicate through shared registers. One can indeed emulate a register abstraction using message-passing (e.g., assuming a majority of correct processes [3]), but such a solution is not efficient in practice, because it does not take full advantage of the message-passing model. Emulating atomic registers leads to a universal construction that requires $\Omega(n^2)$ messages (instead of $\Omega(n)$ in our algorithm), and emulating more powerful abstractions, such as *compare&swap*, *load-linked* and *store-conditional* [14]), is inherently inefficient with message-passing.

Primary-backup algorithms, such as [2, 4, 6], are universal constructions devised with a message-passing model in mind. These algorithms rely on a strong form of leader election that make them however non-indulgent. More precisely, they rely on the assumption of a single primary, and asynchrony in the underlying system may violate this assumption. The semi-passive replication algorithm [9] can be viewed as an indulgent primary-backup universal construction. Nevertheless, because it relies on an underlying consensus-like abstraction, it increases the number of messages exchanged between clients and a replicated service, with respect to traditional primary-backup algorithms. With these algorithms, the client sends its request to the primary; with semi-passive replication, the client must send its request to all replicas. As we pointed out, this is clearly undesirable when the replicated service is accessed through the Internet.

This paper presents an indulgent an universal construction algorithm for a message-passing model with process crash failures[1], using two fine-grained underlying abstractions: a weak form of leader election, $\Diamond$Leader (denoted Ω in [7]), and a new one-shot form of register, $\triangle$Register. Neither of these can, in isolation, implements consensus, but their combined power is equivalent to consensus.[2]

Our algorithm follows the primary-backup replication pattern, and is more efficient than other indulgent primary-backup algorithms: our algorithm uses fewer messages and gives rise to less work by the backups. The latency of our algorithm matches the lower bound established in [6] (for a non-indulgent solution). The message complexity (number of messages exchanged to process a request) is $2n+2$ in our case, just like traditional primary-backup. The message complexity of semi-passive replication is for instance $5n$.

In our algorithm, only primaries update their state, backups only witness the updates performed by primaries. Thus, only a primary has the current state of the replicated object. A backup "constructs" the current state only if and when it becomes primary. Because backups do not update their state, they play an even more passive role in our algorithm than in traditional primary-backup algorithms that seek to keep backups up-to-date. Because a client only needs to send the request to the primary, we combine at the same time the low message-complexity

[1] In our context of message-passing, we say that an object is *wait-free* if any client always returns from the invocation of an operation on this object within a finite number of its own steps independently of the crash of other clients, despite of the failure of a minority of replicas.

[2] Our weak leader election encapsulates the synchrony assumption needed to implement consensus, whereas our register encapsulates the assumption of a majority of correct processes needed for indulgent consensus.

of the primary-backup approach with a low time-complexity of a lazy replication algorithm.

Using our abstractions leads to a surprisingly simple and comprehensive universal construction algorithm. Furthermore, our modularization makes it easy to extend and adapt the algorithm. Indeed, the main idea behind our algorithm is to move work from steady-state periods to transition periods. Thus, the trade-off of our algorithm is to optimize the performance in steady state at the expense of making fail-over more costly. If failures are rare and the fail-over time not that critical, this is likely to be a good trade-off. However, in certain environments, it is very important to react quickly to failures. The good news is that our algorithm allows us to trade-off between steady-state performance and fail-over time in a modular manner, i.e., by *extending* the algorithm, not *changing* it. Moreover, we build our $\triangle$Register abstraction on top of a more basic notion of register, a *round-based* register, denoted $\mathcal{R}$register, for which we give a precise specification. Interestingly, only the $\mathcal{R}$register needs to be changed for the $\triangle$Register implementation (and the universal construction) to fit in various computation models. In fact, we first present our universal construction in a crash-stop model (assuming that processes that crash do never recover), and we show how to easily port it on various crash-recovery models.

To summarise, this paper helps bridge the gap between universal constructions and high-availability middleware, by introducing a universal construction algorithm for message-passing systems with three nice flavors: indulgence, efficiency and genericity. The key to these flavors is the use of two fine-grained abstractions: a weak form of leader election, $\diamond$Leader, and a new form of register, $\triangle$Register.

The rest of the paper is organized as follows. In Section 2, we define the underlying system model. For presentation simplicity, we assume that processes that crash do not recover and we consider reliable channels. We introduce our abstractions in Section 3, and we give our universal construction algorithm based on these abstractions in Section 4. We analyze some execution scenarios in Section 5. Section 6 considers the genericity of our universal construction. In Section 7, we discuss our abstractions and our algorithm, and discuss how easily it can be extended to shorten the fail-over time or deal with process recovery. Optional Appendix A presents different aspects of register abstractions. We prove the correctness of $\triangle$Register in optional Appendix B and our implementation of consensus using $\triangle$Register in optional Appendix C. The correctness of our universal construction algorithm is presented in optional Appendix D.

2 Model

2.1 Processes, Communication, and Time

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. 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 $\mathsf{receive}$. Message passing is reliable in the following sense: (*validity*) 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.

We assume an asynchronous distributed system. There is no bound on the time it takes for a message to reach its destination, nor do we make any assumptions about the time it takes for a process to execute a step in its algorithm. We indirectly introduce synchrony assumptions through the properties of our $\diamond$Leader abstraction (Section 3), and we use a notion of time to specify the properties of our abstractions. To this end, we assume the presence of a discrete global clock with the set of natural numbers as tick range. The purpose of the clock is to simplify the presentation: processes cannot access this global clock.

We sub-divide the set Π of processes into two disjoint subsets: Client and Server. Processes in

Server (*servers*) collectively implement a wait-free linearizable object that processes in Client (*clients*) can access.

2.2 Objects

An object has an internal state and a number of actions to manipulate this state. An action takes an input value and produces an output value. As a side-effect of producing the output value, the action may also update the internal state. Actions may be non-deterministic. That is, the side-effect and output value of a specific action may not be the same each time we execute it, even if we execute it in the same initial state and the same input value.

The goal of our algorithm is to implement wait-free and linearizable access to any given object. Each server has its own copy of the given object. The algorithm uses two primitive and generic actions in dealing with objects:

- The `execute` primitive action takes a request (action name and input value) and returns an output value and an update value. The `execute` primitive executes the action on the given input. The returned update value captures the state update performed by executing the action. The primitive does not change the internal state of the object.

- The `update` primitive takes an update value, and performs the state update captured by the update value.

These two primitives separate the purely functional aspect of an object (mapping input to output) and the state-update aspect (using an input value to update the internal state).

3 Abstractions

Our universal construction is based on two fundamental abstractions: an eventual leader election, denoted $\Diamond$Leader and a one-shot form of register, denoted $\triangle$Register. $\Diamond$Leader encapsulates the synchrony assumptions needed to ensure wait-freedom. It is in this sense a *liveness* abstraction. $\triangle$Register encapsulates a convenient form of storage to ensure linearizability. It is in this sense a *safety* abstraction.

3.1 $\Diamond$Leader

$\Diamond$Leader eventually elects a unique and correct leader. The abstraction has one operation, called elect(). This operation does not take any input parameters. It returns an output parameter, which is the identity of a process. When p_i invokes elect() 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 $\Diamond$Leader 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 elect(), either the process crashes or it eventually returns from the invocation (wait-free).

Notice that our specification does not preclude the existence of concurrent leaders for arbitrary periods of time: hence the notion of *eventual* leader. In the following, we assume the existence of the $\Diamond$Leader abstraction[3] and refer to [7] for its implementation.

3.2 $\triangle$Register

Roughly speaking, our $\triangle$Register is a one-shot register, in the sense that (1) once a value is successfully written, it remains forever in the register, and (2) a write operation is guaranteed to succeed, however, only if a single process is writing. Our $\triangle$Register is different from Lamport's notion of *regular* register, because if two invocations that are not concurrent return a value, these values are necessarily the same. It is also different

[3] As already pointed out, $\Diamond$Leader corresponds to the failure detector Ω of [7]. In the terminology of [7], this is the weakest failure detector to solve consensus.

from an *atomic* register, because an access to a $\triangle$Register can abort in some cases.

Formally, $\triangle$Register has a single primitive, propose(). When a process p invokes propose() with a single argument $v \in \mathsf{Values}$ such that $\mathsf{abort} \notin \mathsf{Values}$, we say that *$p$ proposes v*. The propose() primitive returns a value in $\mathsf{Values} \cup \{\mathsf{abort}\}$. If p returns from propose() with $v' \neq \mathsf{abort}$, we say that *p decides v'* and that the value v' returned is a *decision*. Otherwise, if propose() returns abort, we say that *p aborts*. If p proposes v and decides v, we say that *p commits v*. We now give the properties of $\triangle$Register:

Validity: If a process decides a value v, then v was proposed by some process.

Agreement: No two processes decide differently.

Termination: If a process proposes, it either crashes or returns (wait-free). If only a single process proposes an infinite number of times without crashing, it eventually decides.

In the case where two or more processes concurrently invoke the propose() primitive an infinite number of times (i.e. the invocation of propose() at a process happens before the invocation of propose() at another process returns), the result returned by $\triangle$Register is only restricted by validity and agreement.

Notice that the validity and agreement properties of $\triangle$Register are similar to the traditional consensus abstraction [13]. Our termination property is strictly weaker than in the traditional consensus object (i.e., *if a process proposes, it either crashes or decides*).

3.3 Implementing $\triangle$Register

Figure 2 gives an implementation of $\triangle$Register using the $\mathcal{R}$register abstraction. The implementation of $\triangle$Register encapsulates the round number at which the $\mathcal{R}$register is accessed. We give the specification of the $\mathcal{R}$register in the next section.

A process p_i owns rounds i, $i+n$, $i+2n$, $\ldots$, and only uses these round numbers within the propose() primitive. It first writes using round i, and increments the round number by n on each subsequent call to the propose() primitive. Round number uniqueness is not necessary and is made here only to accelerate the convergence towards agreement.

3.3.1 $\mathcal{R}$register

Roughly speaking, our $\mathcal{R}$register is a round-based register, where processes read and write the content using a single operation. As long as a process does not return a consistent value from the register, it may keep trying to update the content. If a process returns a value from the register, then no process can ever return a different value from the register.

More formally, the interface of our $\mathcal{R}$register consists in a single primitive, readWrite(). When a process p invokes readWrite() with two arguments, respectively $k \in \mathbb{Z}^+$ and $v \in \mathsf{Values}$, such that $\mathsf{abort} \notin \mathsf{Values}$, we say that *$p$ writes v*. The readWrite() primitive returns a value in $\mathsf{Values} \cup \{\mathsf{abort}\}$. If p returns from readWrite() with $v' \neq \mathsf{abort}$, we say that *p reads v'*. We now give the specification of our $\mathcal{R}$register:

Decide-validity: If a readWrite(k,v) returns $v' \notin \{v, \mathsf{abort}\}$, then there is at least one readWrite(k',v') invocation such that $k' < k$.

Abort-validity: If a readWrite(k,$*$) invocation returns abort, then there is a distinct readWrite(k',$*$) invocation such that (i) $k' \geq k$, and (ii) readWrite(k',$*$) is invoked before readWrite(k,v) returns.

Agreement: If a readWrite(k,$*$) returns $v \neq \mathsf{abort}$, then if any readWrite(k',$*$) with $k' \geq k$ returns, the invocation returns either v or abort.

Termination: If a process invokes readWrite($*$,$*$), then it either crashes or the invocation returns (wait-free).

Note that we do not preclude that two processes use the same round number at the same time.[4] As we will see for $\triangle$Register, agreement will be reached faster though, if processes use different round number.

3.3.2 Implementing $\mathcal{R}$register

Figure 1 presents an implementation of our $\mathcal{R}$register, assuming a majority of correct processes. The key idea behind our implementation is that a process can "lock" a value if it successfully stores it among a majority of processes (these processes are called *witnesses* thereafter). Once a value is locked, no other value can be agreed upon. The idea follows that of [3]: each process keeps its own copy of the current value, and accesses the value by emulating round-based read-like or write-like operations out of message passing.

The readWrite() primitive essentially consists in reading the value from a majority of witnesses, and writing ("locking") this value among a majority. If the value read by a process corresponds to $\perp$, then the process writes its own value.

A round number is associated with every message and permits a witness that receives a message to abort if the round number carried by the message is below its own round number. The readWrite() primitive aborts as soon as it aborts at a witness. In order to lock a value, the round number maintained at a witness is updated to the round number of the message upon processing.

We sketch a correctness proof of our implementation of the $\mathcal{R}$register in Appendix A. For presentation clarity, we have tried to keep the algorithm in Figure 1 as simple as possible. Several optimizations are possible. In particular, following an idea in [16], it is possible to eliminate the read phase in steady-state. In Appendix A, we discuss this optimized version of the $\mathcal{R}$register that allows a process to skip the read phase if it is safe to do so. One round-trip communication is enough for a process to propose and decide in this case (steady-state).

[4] In Appendix A, we consider the crash-recovery model, where a process that crashes and recovers might invoke readWrite() with an old round number. Hence our specification of $\mathcal{R}$register.

type Decision **is** Values $\cup$ {abort} $\quad$ {abort $\notin$ Values}

At process p_i:
object $\mathcal{R}$register
  method $\mathcal{R}$register $\quad$ {*Constructor at* p_i}
    decision $\leftarrow$ abort $\quad$ {*decision* $\in$ *Decision*}
    $v^* \leftarrow \perp$ $\quad$ {$v^* \in$ *Values*}
    $read_i \leftarrow 0$ $\quad$ {*Highest* READ *round handled by* p_i}
    $write_i \leftarrow 0$ $\quad$ {*Highest* WRITE *round handled by* p_i}
    $value_i \leftarrow \perp$ $\quad$ {p_i*'s value for the register*}

  method readWrite(Integer k, Values v)
    send [READ,k] to all processes
    wait until received [ackREAD, v_j, ts_j, k] **or** [nackREAD, k] from $\lceil \frac{n+1}{2} \rceil$ processes
    if received at least one [nackREAD, k] **then**
      decision $\leftarrow$ abort
    else
      select the [ackREAD, v_j, ts_j, k] with the highest ts_j
      $v^* \leftarrow v_j$
      if $v^* = \perp$ **then**
        $v^* \leftarrow v$
      send [WRITE, v^*, k] to all processes
      wait until received [ackWRITE, k] **or** [nackWRITE, k] from $\lceil \frac{n+1}{2} \rceil$ processes
      if received at least one [nackWRITE, k] **then**
        decision $\leftarrow$ abort
      else
        decision $\leftarrow v^*$
    return decision

  upon receive [READ, k] from p_j **do**
    if $write_i \geq k$ **or** $read_i \geq k$ **then**
      send [nackREAD, k] to p_j
    else
      $read_i \leftarrow k$
      send [ackREAD, $value_i, write_i, k$] to p_j

  upon receive [WRITE, v_j, k] from p_j **do**
    if $write_i > k$ **or** $read_i > k$ **then**
      send [nackWRITE, k] to p_j
    else
      $write_i \leftarrow k$
      $value_i \leftarrow v_j$
      send [ackWRITE, k] to p_j

Figure 1: $\mathcal{R}$register Implementation

At process p_i:
object $\triangle$Register
  method $\triangle$Register $\quad$ {*Constructor at* p_i}
    $register \leftarrow$ **new** $\mathcal{R}$register $\quad$ {*Instance of register*}
    $k \leftarrow i - n$ $\quad$ {*Initial round number*}

  method propose(v) $\quad$ {*When* p_i *proposes a value* v}
    $k \leftarrow k + n$
    return($register.readWrite(k, v)$)

Figure 2: $\triangle$Register Implementation

We sketch a correctness proof of our implementation of $\triangle$Register in Appendix B.

4 Universal Construction

We present here our universal construction, i.e. an algorithm that transforms any local and sequential implementation of an object into a wait-free linearizable shared object. Like any replication algorithm that guarantees some form of strong consistency, a key principle behind our algorithm is to implement a total order among the requests, and to ensure that a request only appears once in the total order. Because objects may be non-deterministic, the replicas have to agree not only on the total order of requests but also on the state update and reply associated with a given request. In our algorithm, they do that at the same time.

In this section, we describe the algorithm; we prove the algorithm correct in Appendix D.

4.1 Description of the Algorithm

We start by describing the client-side of the algorithm, given in Figure 3. The client accesses the replicated object using the *submit* function[5] and sends the request to its leader. The client then awaits the reply from the leader before returning. If the client does not receive a reply within a certain time, it re-transmits its request (maybe to a different leader). Note that we use a local timer to make this retransmission possible. The purpose of the timer is only to ensure progress (wait-freedom property), impacting only the blocking time (defined in Section 5) the client is willing to tolerate; safety is guaranteed by the abstractions themselves, not by the timer.[6]

We describe now the server-side of the algorithm. Each replica p_i executes the algorithm presented in Figure 4, and has its own copy of the shared object O. Each copy of the object is in the same initial state Λ. A replica also maintains a variable *num* reflecting its opinion about the first free position in the total order. This variable is hence initialized to 1 for each replica.

[5]Actually, this *submit* function would be implemented as a client stub.

[6]As said before, we ignore possible optimizations, e.g. having an adaptive timer, for the sake of simplicity.

When a replica receives a request, it verifies (line 14) that it did not already execute the same request up to its current position *num*. This verification is done locally, without involving any communication step. It is legal because, by the algorithm, the local state of each replica is guaranteed to be coherent with the total order (a complete proof is given in optional Appendix D).

1. If the replica detects that it already decided this request, it simply returns the reply that was committed together with the request.
2. If the replica does not find the request in its local state, it introduces it in the total order.

To insert a new request in the total order, a replica optimistically assumes that its variable *num* reflects the first free position in the total order. It executes the request on its local object, but without performing any update on its internal state. It then constructs an outcome based on the request, the reply and update values returned from the execution of the request. The constructed outcome is proposed at position *num* using a new instance of $\triangle$Register.

Roughly speaking, an instance of $\triangle$Register corresponds to a single position in the total order of requests in our universal construction. Distinct instances are indexed with their respective position in the total order.[7] The messages sent on behalf of an instance are labeled with its index, to differentiate them from the messages of another instance.

There are three possibilities, whether (1) the replica stops electing itself, (2) $\triangle$Register returns abort, or (3) the replica decides some outcome.

The replica exits in case (1), because only leaders are allowed to propose. In case (2) the replica proposes its request again. By the properties of $\triangle$Register, a replica might need to propose several times before committing. If the replica is eventually single to propose and does not crash,

[7]For the sake of clarity, the algorithm in Figure 4 uses a single instance of $\triangle$Register and the index appears as a subscript of the propose() primitive. This is equivalent to using several instances of $\triangle$Register.

```
◇Leader leader ← new ◇Leader
Reply res ← nil

procedure submit(Request req)
  while true do
    set timer
    send [Request, req] to leader.elect()
    wait until received [Reply, res] or timer expires
    if received [Reply, res] then
      return(res)
```

Figure 3: Client Behavior

```
1:  type Outcome is [Request,Reply,Update]
2:  type Decision is Outcome ∪ {abort}

3:  Object O ← new Object
4:  ◇Leader leader ← new ◇Leader
5:  △Register register ← new △Register
6:  OutcomeStore store ← new OutcomeStore
7:  Integer num ← 1
8:  Outcome prop ← nil
9:  Decision decision ← abort
10: Reply res ← nil
11: Update upd ← nil

12: upon receive [Request,req] from c do
13:   while true do
14:     if store.isCommitted(req,num)=[true,decision] then
15:       send [Reply,decision.res] to c
16:       break
17:     [res,upd] ← O.execute(req)
18:     prop ← [req,res,upd]
19:     decision ← abort
20:     while decision=abort and leader.elect()=p_i do
21:       decision ← register.propose_num(prop)
22:     if decision=abort then break
23:     store.setCommitted(num,decision)
24:     num ← num + 1
25:     O.update(decision.upd)
26:     if decision=prop then
27:       send [Reply,decision.res] to c
28:       break
```

Figure 4: Replica Behavior for Process p_i

it will eventually decide. By the properties of △Register, there will eventually be a single process which proposes. In case (3), the replica stores the decided outcome in its local state, increments its variable *num* to the next position, and updates its object using the update value returned with the decision (line 25).

From case (3), two different sequels are possible, whether (a) the replica committed its outcome, or (b) the replica decided an outcome that it did not propose.

In situation (a), the reply is sent to the client

```
object OutcomeStore
  method OutcomeStore        {Constructor at process p_i}
    Integer i ← 0
    Outcome outcomes[ ] ← {nil, ..., nil}

  method isCommitted(Request req, Integer index)
    for i from 1 to index − 1 do
      if outcomes[i].req = req then
        return(true,outcomes[i])
    return(false,nil)

  method setCommitted(Outcome out, Integer index)
    outcomes[index] ← out
```

Figure 5: Uniqueness Verification for Process p_i

(the test at line 26 evaluates to true), the replica exits and waits for the next request. In situation (b), the replica cannot send the reply belonging to the decided outcome to the client (because the client is awaiting a reply for another request), nor can it send the reply that it locally computed, even though this is the reply awaited (because the reply has not been committed). Hence, the replica restarts the algorithm.

A replica stores a decided outcome within its local state (line 23). This is used to accelerate the verification of whether a request is new when it comes in (line 14). The algorithm used for this verification is shown in Figure 5. It is possible (though very costly) not to verify whether a received request is new, and to systematically propose this request starting from position 1 in the total order.

Finally, consider a replica that decides an outcome that it does not propose. Following the algorithm, the replica restarts the algorithm, and verifies whether the request is actually a new one. As it already executed this verification once, it is sufficient to verify that the request is new among the outcomes committed since the last verification. In Figure 5, we ignore this obvious optimization to make the algorithm simpler.

5 Performance Analysis

We analyze the performance of our universal construction through the following metrics:

- Message complexity: The number of messages it takes to process a request end-to-end.

- Response time: The end-to-end latency observed by clients.

To quantify latency, we assume that message transmission times are bound by some known δ.

5.1 Steady-state

To analyze the steady-state behavior of our algorithm, we consider a nice run in which no process crashes and in which $\diamond$Leader returns the same leader to all processes at every invocation.[8] Moreover, we assume that this leader uses the optimized implementation of the $\mathcal{R}$register given in Appendix A.

In a nice run, our algorithm has the following communication pattern. A client submits a new request to the leader, who handles the request. When the leader has processed a request, it commits the result among the other replicas (using $\triangle$Register) and updates its local object accordingly. The leader finally returns the corresponding reply to the client.

When a client submits a new request, the message complexity is $2n + 2$ and the response time is bounded by 4δ. Following the algorithm, the leader r proposes the request with num set to one more than the previous request. Because we consider a nice run, r is perpetual leader, and no other replica has been leader in the meantime. This means that r will commit the request the first time it proposes it. Committing a request in the optimized version of $\triangle$Register has message complexity $2n$ and latency 2δ (in the optimized implementation, the leader skips the read phase). The communication between client and leader involves a single round-trip message, and has complexity 2 and latency 2δ. In total, we get a message complexity of $2n + 2$, and a total latency of 4δ.

[8] Steady-state performance can be achieved in runs that are not so nice, but where the leader does not crash and does not change (no matter what happens to the other replicas).

5.2 Transitions

Consider the case where a client sends a request to its current leader and this leader crashes before updating any replica. If a replica r_l is then elected leader for the first time, its object is in state Λ and its variables are initialized to their respective initial values. In particular, the variable num_l is set to 1. The client sends the same request to r_l. To commit this request, r_l successively proposes the request from $num_l = 1$ up to the position of that request in the total order (if the request is an old one) or to the first unused position in the total order. As it increments its variable num_l, r_l updates its object with the update values committed in the total order.

The message complexity and response time in a fail-over scenario depends on many factors, such as the number of concurrent leaders that try to take over from the failed leader. If multiple leaders try to take over, it may require multiple rounds inside of $\triangle$Register to commit requests. Furthermore, a leader that takes over has to both read and write inside of $\triangle$Register. In general, this means that each round initiated by a new leader inside of $\triangle$Register requires $4n$ messages and involves a latency of 4δ. If a new leader has to commit k requests to construct its state during take-over, the leader has to at least initiate k rounds—one for each request.

In Section 6.1, we show how to extend our algorithm to reduce the communication during fail-over at the expense of increased communication during steady-state.

6 Genericity

6.1 Trade-offs

In our algorithm, backups play a very passive role: a backup postpones all work until the last possible moment, and "catches up" if and when it becomes a leader. This scheme is clearly efficient during steady-state periods but not during transition periods (i.e., fail-over time). Here, we show how one could easily extend our algorithm (without modifying it) to move work from transi-

```
upon p_i commits decision do
    send [Eager,decision] to all except p_i

upon receive [Eager,decision] where decision.pos = num do
    O.update(decision.upd)
    store.setCommitted(decision.pos,decision)
    num ← num + 1
```

Figure 6: An extension at process p_i that shifts work from transition periods to steady-state

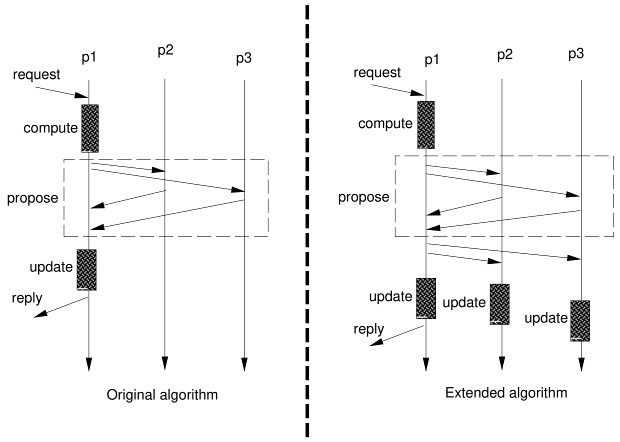

Figure 7: Steady-state behavior for the original and extended scheme

tion periods to steady-state periods, and thereby achieve a faster fail-over time.

In Figure 6, we illustrate such an extension. The two **upon** clauses extend the behavior of Figure 4. In Figure 6, a leader sends each committed update value to all other replicas. The other replicas then process the received update values in order. Notice that the dissemination of update values does not have to be reliable: the basic algorithm ensures consistency and progress, even if a leader fails while sending an update value. We show the different steady-state interaction patterns in Figure 7.

These patterns capture two extremes of a spectrum. We can think of various trade-offs in between these two extremes.

Our scheme also allows for work compression. For example, the leader may gather a number of update messages and send them as a single message. Moreover, some of the "old" updates may become obsolete after "new" updates have been computed. For example if an update message assigns a value to a variable, and if a subsequent update message also assigns a value to the same variable, we only have to apply the most recent update message. Detecting obsolete update values, requires knowledge about the semantics of the state machine, and cannot be performed in a generic manner in the replication algorithm.

6.2 Dealing with Recovery

We have defined our universal construction for a crash-stop model: a process only fails by crashing, and does not recover after a crash. We made this assumption to simplify the presentation. One of the complications of dealing with process recovery is the use of stable storage. To recover consistently, a process has to store parts of its state in stable storage. When recovering, a process can then access its own stable storage to determine its pre-crash state. Stable storage is however expensive and should be minimized.

It turns out that we can encapsulate the manipulation of stable storage within our $\mathcal{R}$register. That is, neither our universal construction nor our $\triangle$Register have to manipulate stable storage. In fact, it would work as is in a crash-stop model. A recovering process would start out as a backup, and if it becomes leader, it would construct its state from scratch, just as if it were a backup that had never failed and never been leader.

In optional Appendix A.3, we give implementations for $\mathcal{R}$register in different crash-recovery models.

7 Discussion

The key to the indulgence, efficiency, and genericity of our algorithm is the use of two fine-grained underlying abstractions: $\Diamond$Leader and $\triangle$Register. Separately, each of these abstractions are strictly weaker than consensus. Together, they can implement consensus, as we show in Figure 8.[9]

[9] Throughout this section, we consider the *uniform* variant of the consensus problem (i.e. we do not restrict agreement to correct processes only).

```
At process p_i:
object Consensus
  method Consensus                    {Constructor at process p_i}
    register ← new △Register
    leader ← new ◇Leader
    decision ← abort                  {decision ∈ Decision}

  method Propose(Values init_i)
    while decision=abort do
      if leader.elect()=p_i then
        decision ← register.propose(init_i)
    send [decision] to all processes
    return decision

  upon receive [value] from p_j for the first time do
    decision ← value
```

Figure 8: A Simple Consensus Algorithm

Consensus defines a single primitive, Propose(). We say that a process *proposes* a value v when it invokes Propose() with v. A process *decides* a value v if it returns from Propose() with v. Only the termination property of consensus (*every correct process eventually decides*) actually differs from $\triangle$Register.

We sketch the correctness proofs in optional Appendix C.

Interestingly, our consensus algorithm in Figure 8 can be viewed as a modular variant of the Synod consensus scheme underlying the Paxos replication algorithm [16]. In fact, our $\diamond$Leader abstraction is identical to the notion of "sloppy" leader in [17], and $\triangle$Register precisely crystallizes Lampson's intuition [17] about the safety of the agreement part of the Synod consensus algorithm [16]: we capture both safety and liveness aspects of that intuition through our $\triangle$Register specification. [10] To our knowledge, Paxos pioneered the idea of combining a notion of leader election and some form of register, although not explicitely identified (more recent variants of the algorithm, e.g. [8, 11], make more explicit use of register-like abstractions).

There are two major differences between the use of $\diamond$Leader and $\triangle$Register in our universal construction and in Paxos. First, Paxos is not a universal construction since it relies on objects to be deterministic. To implement the agreement on total order and on state in a single instance of eventual consensus, our algorithm invokes $\triangle$Register *a posteriori*, after processing a request. This is in contrast to Paxos because, being deterministic, the replicas only need to agree on the total order of requests, and this agreement can be reached a priori before a request is processed. One could extend [16] to non-deterministic replicas using an agreement phase after processing the request, but this would result in two agreements per request (as in [13]). Our algorithm only requires a single agreement phase per request in steady-state. Second, we make use of our two fine-grained abstractions as first class citizens in our universal construction. As we observe in [5], two variants of Paxos are actually given in [16]: a modular algorithm based on a consensus primitive itself built using a sloopy leader and a register-like abstraction, and then an ad-hoc algorithm that opens these abstractions for the sake of performance. By promoting our $\diamond$Leader and $\triangle$Register as first class citizens of the algorithm, we obtain a modular and efficient scheme. Although the combination of these two is as strong as consensus, their use separately leads to a more efficient replication scheme. This is exactly what makes our universal construction more efficient than [9]. Indeed, where a consensus object requires all replicas to propose and decide, $\triangle$Register allows a single replica to propose and decide by itself. With a consensus, it is not possible for backups to play a passive role where they just witness the actions taken by a primary. Moreover, consensus encapsulates $\diamond$Leader whereas $\triangle$Register does not. If we used consensus, and if we had another notion of leader in the replication algorithm (e.g., the primary), these leaders may not coincide (the same process may not be leader at both levels). Having different leaders at different levels would likely result in an additional leader-to-leader communication that is absent in our single-leader scheme. This saves messages between client and replicas.

Indirectly, we argue through this paper that $\diamond$Leader and $\triangle$Register are natural abstractions to build universal constructions in a message-passing model, pretty much like consensus and

[10] $\triangle$Register is also close to the *k-converge* primitive, (with $k = 1$) of [18].

atomic register do in a shared-memory model.

References

[1] M. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the International Workshop on Distributed Algorithms, Springer-Verlag (LNCS)*, April 1998.

[2] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second IEEE International Conference on Software Engineering (ICSE)*, 1976.

[3] H. Attiya, A. Bar-Noy, and D. Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM*, 42(1):124–142, 1995.

[4] J. F. Bartlett. A nonstop kernel. In *Proceedings of the 8th ACM Symposium on Operating System Principles (SOSP)*, 1981.

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

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

[7] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector to solve consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[8] G. Chockler and D. Malkhi. Active disk paxos with infinitely many processes. In *Proceedings of the 21st ACM Symposium on Principles of Distributed Computing (to appear)*, July 2002.

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

[10] S. Frølund and R. Guerraoui. X-ability: A theory of replication. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, 2000.

[11] Eli Gafni and Leslie Lamport. Disk paxos. In *International Symposium on Distributed Computing*, pages 330–344, 2000.

[12] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, 2000.

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, January 1991.

[14] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[15] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[16] L. Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, 1989. Also published in ACM Transactions on Computer Systems (TOCS), Vol. 16, No. 2, 1998.

[17] 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.

[18] Jiong Yang, Gil Neiger, and Eli Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing*, pages 297–306, 1998.

A Aspects of the $\mathcal{R}$register (Section 3.3.1)

A.1 Correctness of the $\mathcal{R}$register

We sketch the proof of correctness for the algorithm in Figure 1.

Decide-Validity: From the algorithm in Figure 1, it is clear that value $value_i$ at a witness p_i can only contains some proposed value or $\bot$. When a process p_i returns a decision value, this value is read from a witness. If the value read turns out to be $\bot$, process p_i decides its own value.

Abort-Validity: To abort, a process that invokes readWrite(k,$*$) must receive either a nackREAD or a nackWRITE. To generate a nackREAD, a witness must have processed a message with $k' \geq k$. To generate a nackWRITE, a witness must have processed a message with $k' > k$. In both cases, there has been an operation readWrite(k',$*$) with $k' \geq k$ which started before readWrite(k,$*$) returns.

Agreement: If no process returns from readWrite() or if all invocations of readWrite() return abort, the property is trivially satisfied. So, let k be the smallest round number in which a process p invokes readWrite(k,$*$) and returns $v \neq$ abort. We claim that for all rounds $k' \geq k$, a process q that invokes readWrite(k',$*$) returns either v or abort.

The proof is by induction on the round number. For $k' = k$, q necessarily aborts, because it receives a nackREAD from at least one witness.[11] Assume that the claim is true for all $k \leq k' < r$, and consider process q that invokes readWrite(r,$*$). We show that the claim holds for $k' = r$ at process q.

Consider that process q reads from a majority of witnesses and does not receive any [nackREAD, $*$] message (otherwise, the claim is trivially satisfied), there exists a witness w such that (i) w sends a message [ackWRITE, k] to p during round k and (ii) w sends a message [ackREAD, v_w, ts_w, r] to q during round r. Because w updated $value_w$ in round k, $ts_w \geq k$. Clearly, we also have $ts_w < r$, otherwise, q would receive a nackREAD from w. Now let t be the largest ts_w that is received in some message [ackREAD, $*, t, k$] by q, at round r, from some witness w. From the remarks above, we have $k \leq t < r$.

Thus, there exists a process s which invoked readWrite(t,$*$), and witness w updated both its value $value_w$ to some value v' and its variable $write_w$ to t. Because $t < r$, by the induction hypothesis, process s reads value v and writes v to a majority of witnesses. Consequently, $v' = v$ for w and process q reads value v from witness w in round r. Then, if q does not receive any [nackWRITE, r] message, it decides v. In both cases, the claim is satisfied. We conclude that agreement is satisfied.

Termination: From the assumption of a majority of correct processes, no operation can block. Thus, if a process does not crash, it returns from the readWrite() primitive.

A.2 Fast$\mathcal{R}$register

The Fast$\mathcal{R}$register is an optimized implementation of the $\mathcal{R}$register used in our universal construction. It is inspired from [16] and is used in [5]. Figure 9 gives the implementation of Fast$\mathcal{R}$register, and Figure 10 gives a modified version of $\triangle$Register that uses the Fast$\mathcal{R}$register.

Compared with $\mathcal{R}$register, the readWrite() primitive of the Fast$\mathcal{R}$register avoids one round-trip of messages between the leader and a majority of witnesses in steady-state: the leader ignores what has been previously written and immediately tries to "lock" the value among a majority.

[11] In our universal construction, processes use different round numbers, which often accelerate the convergence of the algorithm. Nevertheless, our $\mathcal{R}$register implementation is correct even if we do not assume round number uniqueness.

The Fast$\mathcal{R}$register satisfies the following property:

Fast-Write-Uniqueness: In every execution where there is at most one invocation of readWrite(k,$*$) such that $k \leq n$, the Decide-Validity, Abort-Validity, Agreement and Termination properties of the $\mathcal{R}$register apply to the Fast$\mathcal{R}$register.

The readWrite() primitive of our Fast$\mathcal{R}$register implementation also returns an additional boolean variable, which indicates to a process whether it is allowed to invoke the readWrite(k,$*$) primitive with $k \leq n$ on the next instance of Fast$\mathcal{R}$register in the total order.

A.2.1 Description of the Implementation

To allow a process to skip the first read operation, a necessary condition is to guarantee that a process will not overwrite the current value if it directly writes. There are two situations for which such a condition holds:

1. No value has been written. A process can write its value without reading first, as there is no previous value.

2. The value locked is associated with a high round number. If a process directly writes, but with a low round number, its write operation will abort—as a read operation would have done.

We change the implementation of the message handling to automatically increment the associated round number to $n + (1/2)$ whenever it successfully executes an operation carrying a round number $\leq n$. If we allow a process to skip the read operation only if its round number is $\leq n$, the two previous statements holds. Thus, a process can skip the read operation only if it proposes for the first time, i.e. when its round number is $\leq n$.

In fact, it is necessary and sufficient to guarantee that the read operation is skipped exactly once per instance of Fast$\mathcal{R}$register. Imagine the scenario where two processes write two different values without first reading; the first reaches $\lceil \frac{n+1}{2} \rceil$ processes whereas the second reaches $\lfloor \frac{n-1}{2} \rfloor$. If some processes of the first group crash, a process consulting the value written might encounter a situation where it cannot distinguish a majority among the two groups. Hence the following condition: a process is allowed to skip the read operation for Fast$\mathcal{R}$register instance n if it was the first process to propose for instance $n-1$, and no processes had proposed for instance n by that time.

In the implemention of the readWrite() primitive, the algorithm uses a boolean variable $nextFast$ that is true whenever a process is allowed to skip the read operation on the next instance of Fast$\mathcal{R}$register. When a process proposes and receives only [ackWRITE, $k, true$] messages after its write operation, it sets the variable $nextFast$ to $true$, and to $false$ otherwise. Because this variable must be accessed by all instances of the Fast$\mathcal{R}$register, we make it as a global variable. In Figure 9, $value_i^{+1}$ designates the variable $value_i$ of the next instance of Fast$\mathcal{R}$register in the total order. If such instance does not exist (because it has not been created yet), this variable trivially equals to $\perp$.

A.2.2 Correctness

We consider the case where there is indeed exactly one invocation of readWrite(k,$*$) with $k \leq n$. (The case where there is none eactly corresponds to the implemenetation of $\mathcal{R}$register in Figure 1 and the proof of the previous section applies.)

The proofs of the Abort-Validity, Decide-Validity and Termination properties are similar to those for $\mathcal{R}$register. We sketch here the proof for agreement:

Agreement: Assume by contradiction that process p invokes $\mathsf{readWrite}(k,*)$ with $k \leq n$ and returns v, and process q invokes $\mathsf{readWrite}(k',*)$ with $k' \geq n+1$ and returns v'. They both decide differently.

Consider the state of the witnesses after process q successfully returns from its readWrite() invocation. A majority of them now have $write_i \geq n+1$. Upon p's readWrite() invocation, it uses a round number $< n$, and thus aborts. The next times it proposes, it cannot use a round number $\leq n$ by assumption, and thus, executes the same operations as in the readWrite() primitive in Figure 1. Thus, the agreement property of $\mathcal{R}$register applies: a contradiction.

Consider now that process p successfully returns value v from its invocation of readWrite. A majority of witnesses must have $value_i = v$. Because q invokes $\mathsf{readWrite}(k',*)$ with $k' \geq n+1$, the Agreement property of $\mathcal{R}$register applies and q reads v from some witness w, and decides v or aborts: a contradiction.

A.3 Crash-Recovery Model (Section 6.2)

We consider our universal construction in the crash-recovery model. First note that the our universal construction adapts very smoothly to this model. Indeed, we only need to slightly modify the implementation of the $\mathcal{R}$register abstraction. Neither the $\triangle$Register abstraction nor the universal construction need to be modified for this model.

In the crash-recovery model, we consider the following four classes of processes:

Always up: Processes that never crash.

Eventually always up: Processes that crash at least once, but that are always up after a certain time.

Eventually always down: Processes that are permanently down after a certain time.

Unstable: Processes that crash and recover infinitely many times.

The processes of the first two classes are the *correct* processes, whereas the processes of the last two classes are the *faulty* processes.

A.3.1 Retransmission Module

In the crash-stop model, we assumed reliable channels. Here, because we do not consider the same notion of correctness for a process, the validity property (e.g. *if a correct process sends a message to a correct process, then the message is eventually received*) of the channel does not hold any more.

We use the **s-send** and **s-receive** primitives defined in [1, 5]. The implementation is given in Figure 11. In the implementation of $\mathcal{R}$register, we replace the invocations to the **send** and **receive** primitives with invocations to the **s-send** and **s-receive** primitives respectively. Clearly, the following property is satisfied:

Validity: If any process p_i s-sends a message m to a correct process p_j, then if p_i does not crash p_j eventually s-receives m.

Because messages can now be duplicated and because we do not assume FIFO channels, we assume that every message includes an identifier which is unique among the messages from the same process (e.g. an absolute timestamp). Whenever a witness positively acknowledges a message, it stores the

```
At process p_i:
object FastRregister
  method FastRregister                                              {Constructor at p_i}
    decision ← abort                                                {decision ∈ Decision}
    v* ← ⊥                                                          {v* ∈ Values}
    read_i ← 0                                                      {Highest READ round handled by p_i}
    write_i ← 0                                                     {Highest WRITE round handled by p_i}
    value_i ← ⊥                                                     {p_i's estimate value for this instance}

  method readWrite(Integer k, Values v)
    if k > n then
      send [READ, k] to all processes
      wait until received [ackREAD, v_j, ts_j, k] or [nackREAD, k] from ⌈(n+1)/2⌉ processes
      if received at least one [nackREAD, k] then
        return [false, abort]
      else
        select the [ackREAD, v_j, rs_j, k] with the highest ts_j
        v* ← v_j
        if v* = ⊥ then v* ← v endif
    else
      v* ← v
    send [WRITE, v*, k] to all processes
    wait until s-received [ackWRITE, k, permission_j] or [nackWRITE, k] from ⌈(n+1)/2⌉ processes
    if received at least one [nackWRITE, k] then
      decision ← abort
      nextFast ← false
    else
      decision ← v*
      if received only [ackWRITE, k, true] then
        nextFast ← true
      else
        nextFast ← false
    return [nextFast, decision]

  upon receive [READ, k] from p_j do
    if write_i ≥ l or read_i ≥ k then
      send [nackREAD, k] to p_j
    else
      read_i ← k
      send [ackREAD, value_i, write_i, k] to p_j

  upon receive [WRITE, v_j, k] from p_j do
    permission ← false                                              {permission ∈ Boolean}
    if write_i > k or read_i > k then
      send [nackWRITE, k] to p_j
    else
      if k ≤ n then                                                 {Has readWrite() already been called?}
        write_i ← n + (1/2)                                         {No}
      else
        write_i ← k                                                 {Yes}
      permission ← ((value_i = ⊥) ∧ (value_i^{+1} = ⊥))   {value_i^{+1} refers to value_i of next instance of FastRregister}
      value_i ← v_j
      send [ackWRITE, k, permission]
```

Figure 9: Fast$\mathcal{R}$register Implementation

identifier of the message to avoid sending a negative acknowledgement to this process if it receives the same message several times (this is possible because we use the s-send primitive).

Because a process can crash, recover, and then re-use an old round number, it is possible that an ACK generated upon a previous invocation to readWrite() by this process is received instead of a NACK. Because the round number is the same, the process does not detect it and commits, violating agreement. To avoid this possibility, we assume that every ACK (resp. NACK) message also includes the unique

```
At process p_i:
object △Register
  method △Register                                            {Constructor at p_i}
    reg ← new FastRregister                                   {Instance of register}
    k ← i                                                     {Initial round number}

  method propose(v)                                           {When p_i proposes a value v}
    if ((¬nextFast) ∨ (k > n)) then
      k ← k + n
    [nextFast, decision] ← reg.readWrite(k, v)
    return(decision)
```

Figure 10: Optimized △Register Implementation

```
1: procedure Initialization
2:   xmitmsg[] ← ∅; start task{retransmit}
3: procedure s-send(m) to process p_j                         {To s-send m to p_j}
4:   if m ∉ xmitmsg then
5:     xmitmsg ← xmitmsg ∪ {[p_j, m]}
6:   send(m) to p_j
7: upon receive(m) from p_j do
8:   s-receive(m) from p_j
9: task retransmit                                            {Retransmit all messages sent}
10:  while true do
11:    for all [p_j, m] ∈ xmitmsg do
12:      send(m) to p_j
```

Figure 11: Retransmission Module at Process p_i

identifier of the message that generates it. A process that receives an ACK (resp. NACK) message validates it, or rejects if it was in fact a reply for an old invocation. For the sake of clarity, we ommit the represention of this identifier in the algorithms.

In the following, we distinguish the cases when stable storage is available and when it is not.

A.3.2 Stable storage is not available

When stable storage is not available, and if the number of processes that are always up, say n_a, represents a majority of processes, i.e. $n_a > n/2$, then the model is very similar to the model where crashes are permanent and a majority of processes do not crash. In the algorithm, we simply exclude processes that recover from subsequent operations (they do not witness any operation, but may nevertheless receive a decision). For example, a process that recovers may execute a special recovery procedure where it keeps broadcasting to all a special recovery message. Whenever a process receives such a message, it excludes the sender from subsequent operations.

If we call n_f the number of faulty processes, it has been shown [1] that if $n_a > n_f$, stable storage is not needed to achieve agreement. However, the $\Diamond$Leader abstraction has to be changed to take into account the fact that an unstable process might always elect itself as the leader, and thus prevent any progress to be ever accomplished. We refer to [5, 1] for the specific implementation.

The implementation of the $\mathcal{R}$register has to be slightly modified, as a process cannot wait for the answers from a majority anymore. More specifically, a process now awaits $max(n_f+1, n-n_f-|R|)$ acknowledgments (instead of a majority) after either a read or a write operation (R is the set of processes known to have recovered). This corresponds to the maximum number of answers a process might expect to receive without blocking. When a process discovers a new process that recovers, it restarts to send messages from the beginning (this is exactly as described in [1]). The modification of lines 11 and 12 in

```
R ← ∅                                                         {Set of processes known as having recovered}
precR ← ∅

repeat
   s-send [READ, k] to all processes
   precR ← R
until (R = precR) ∧ (s-received [ackREAD, v_j, ts_j, k] or [nackREAD, k] from max(n_f + 1, n − n_f − |R|) processes )
```

Figure 12: Modifications of $\mathcal{R}$register in the Crash-Recovery Model

Figure 1 is shown in Figure 12, and would be similar for lines 20 and 21 of Figure 1.

We refer to [1] for a formal proof, and we give here an intuition of why the $\mathcal{R}$register implementation works in this case. Intuitively, our $\mathcal{R}$register preserves agreement because (i) upon reading among witnesses, a process always waits until at least a process that is always up answers, and (ii) upon writing among witnesses, a process always waits until at least all processes that are potentially always up (not known to have recovered) positively acknowledge the operation. Thus, a process reads always the value associated with the latest round, and from this point, the correctness proofs are the same as in the crash-stop model.

A.3.3 Stable storage is available

It has also be demonstrated in [1] that if the number of correct processes n_c is a majority of processes (note that this is strictly weaker than saying $n_a > n_f$) and $n_a \leq n_f$, then agreement needs stable storage (we refer to [1] for a formal proof of this result). Stable storage is used to maintain consistency at correct processes. Hence, we change the message handling implementation to exploit stable storage. The modification is shown in Figure 13 for $\mathcal{R}$register, but would be similar for the Fast$\mathcal{R}$register.

To give an intuitive idea of why $\mathcal{R}$register is correct in the crash-recovery model using the implementation given in Figure 13, note that witnesses store their copy of the value in stable storage, and retrieve the latest state upon recovery. As we assume a majority of correct processes, at least one witness always knows the value written with the latest round. The correctness proofs are similar to the crash-stop model from this point.

As shown in Figure 14, our universal construction can be further optimized if the replicas are allowed to use stable storage at the level of the universal construction itself. Indeed, a replica can store in stable storage its local state. Upon recovery, a replica loads its local state from stable storage. Consider now that this replica is elected leader. Whenever a request comes in, it does not have to propose it from position 1 in the total order, but it can use the local state retrieved from the stable storage to verify that the request is unique among the requests committed before the replica crashes.

B Correctness of our $\triangle$Register Implementation (Section 3.2)

We sketch the proof of correctness for the algorithm in Figure 2.

Validity: Obvious from the Decide-Validity property of $\mathcal{R}$register.

Agreement: Let k be the smallest round for which a process invokes the readWrite(k,v) primitive and decides a value. The decision value is v, otherwise there exists a round $k' < k$ where readWrite(k',v') is invoked with $v' \neq v$, and k is not the smallest round number (if another process writes and decides a value at round k, by the Agreement property of $\mathcal{R}$register, this value is necessarily v).

```
upon s-receive [READ, k] from p_j do
  if write_i ≥ k or read_i ≥ k then
    s-send [nackREAD, k] to p_j
  else
    read_i ← k
    store(read_i)                                   {Added from Figure 1}
    s-send [ackREAD, value_i, write_i, k] to p_j

upon s-receive [WRITE, v_j, k] from p_j do
  if write_i > k or read_i > k then
    s-send [nackWRITE, k] to p_j
  else
    write_i ← k
    value_i ← v_j
    store(write_i, value_i)                         {Added from Figure 1}
    s-send [ackWRITE, k] to p_j

upon recovery do                                    {Added from Figure 1}
  Initialization                                    {Initialize all local variables}
  load(write_i, read_i, value_i)
```

Figure 13: $\mathcal{R}$register Implementation in the Crash-Recovery Model

```
At process p_i:
object OutcomeStore
  method OutcomeStore                               {Constructor at process p_i}
    Outcome outcomes[ ] ← {nil, ..., nil}

  method isCommitted(Request req, Integer index)
    for i from 1 to index − 1 do
      if outcomes[i].req = req then
        return(true,outcomes[i])
    return(false,nil)

  method setCommitted(Outcome out, Integer index)
    store(outcomes[index],index)
    outcomes[index] ← out

  upon recovery do
    load(outcomes[ ],index)
    num ← index
```

Figure 14: Uniqueness Verification in the Crash-Recovery Model

By the Agreement property of $\mathcal{R}$register, any process that proposes thereafter either aborts or decides the same value v.

Termination: From the Termination property of $\mathcal{R}$register, the readWrite() primitive cannot block. Thus, a process that invokes the propose() primitive either returns or crashes. Assume that a single process invokes readWrite(k,$*$) once and does not crash thereafter. There are two cases to consider: (1) It reads a value, or (2) it reads abort. For (1), by the Agreement property of $\mathcal{R}$register, this value can be decided. In case (2), by the Abort-Validity property of $\mathcal{R}$register, there exists a readWrite(k',$*$) invocation with $k' \geq k$. If the process proposes an infinite number of times and does not crash, it increases its round number for each subsequent call to readWrite(). It will eventually reach round k' to propose a value. As it is single to propose, no other process propose in the meantime, and when it uses round number k', it reads a value from the round number higher that this round, it will be able to lock the value among a majority and thus decides this value.

C Correctness of our Consensus Algorithm (Section 7)

We sketch the correctness proofs of the algorithm presented in Figure 8 implementing Consensus with $\triangle$Register and $\diamond$Leader:

Lemma 1 Validity: *If a process decides v, v was proposed by some process.*

PROOF: A process decides a value when it returns from the Propose() at line 12. Its variable *decision* is assigned at line 10 or when it receives a value from another process, at line 13. In both cases, the value is returned by $\triangle$Register. By the validity property of $\triangle$Register, this value must have been Proposed by some process. From the algorithm, processes only use their input values as an input for $\triangle$Register. □

Lemma 2 Agreement: *No two processes decide differently.*

PROOF: Assume by contradiction that two processes decide differently. By the agreement property of $\triangle$Register, only a single value v can be committed at line 10, so a correct process that subsequently proposes necessarily decides v (by the agreement property of $\triangle$Register). The only possibility left is that one process received a wrong decision value v' at line 13. This means that another process sent the decision value v' at line 11. This process must have successfully exited the loop. By the agreement property of $\triangle$Register, it must have decided v: a contradiction. □

Lemma 3 Termination: *Every correct process eventually decides.*

PROOF: Compared to $\triangle$Register, we make the additional assumption that a majority of correct processes (and among them, the perpetual leader) eventually invoke the Propose() primitive.

Assume by contradiction that a correct process proposes and never decides. Because the perpetual leader eventually invokes the Propose() primitive, some value is eventually committed (because of the properties of $\diamond$Leader, eventually only the perpetual leader proposes; and it eventually decides as it proposes an infinite number of times and never crashes). Any correct process that decides, and at least the leader, sends the decision to all. By the reliability assumption of the channels, every correct process receives the message, and decides thereafter: a contradiction. □

D Correctness of our Universal Construction (Section 4)

D.1 Specification

We define a notion of linearizability that allows us to reason about replicated objects. Because it is defined for a shared-memory model, the original definition of linearizability [13] does not have to deal with replication: there is only a single copy of the shared object, and this copy resides in shared memory. To define linearizability for replicated objects, we use elements of x-ability [10].[12]

We define correctness in terms of how the clients view the shared object. Roughly speaking, clients should be given wait-free access to the shared object, and the invocation history observed by clients should be linearizable. Clients submit requests and receive replies. It should appear to each client as if

[12] Defining complementary notions of linearizability and x-ability, where linearizability deals with concurrency only, and x-ability deals with replication only, is a topic for future work.

object actions (triggered by submitted requests) happen atomically in some total order that is consistent with the real time order seen by clients. To formalize this requirement, we introduce a function, called possibleReply, that captures the request-reply relationship for a given object. The possibleReply function takes a sequence of requests and returns a set of reply sequences. The returned set contains all the sequences that the object may return when presented with the given requests one at a time. In our definition of correctness, we assume that requests and replies are uniquely identified. We capture the relationship between the requests and the states of the object with the function PossibleStates. The PossibleStates function takes two sequences, one of requests and another one of replies, and returns a set of states in which the object can be if, starting from the initial state Λ, we apply to it the requests of the first sequence and receive the replies of the second sequence.

We consider a system with a replicated service and n clients. These n clients each submit a number of requests and receive a number of replies. We use r_i^j to denote the i'th request made by client j, and we use rep_i^j to denote the corresponding reply. We say that the service is linearizable if there exists two sequences, seq_r and seq_{rep}, such that seq_r contains the requests r_i^j and seq_{rep} contains the replies rep_i^j, and the following conditions are satisfied:

1. $seq_{rep}(k) = rep_i^j$ if and only if $seq_r(k) = r_i^j$.
2. If a client receives the reply $rep_i^j = seq_{rep}(k)$ before a client submits the request $r_{i'}^{j'} = seq_r(h)$, then $h > k$.
3. $seq_{rep} \in \mathsf{possibleReply}_S(seq_r)$

The first property requires the two sequences to collectively define a total order for the request-reply interaction. For a given request-reply pair, the request and reply should be at the same position in their respective sequences. The second property ensures that the total order defined by the sequences satisfy the real-time order seen by clients. The third property reflects the requirement that the totally ordered request-reply interaction should obey the sequential, single-copy specification of the object.

Notice that to formalize the real-time requirement of linearizability, we use the notion of a global clock. As in Section 2, the global clock is a purely hypothetical device that we introduce for presentation simplicity.

D.2 Proofs

The proofs are organized as follows. First, we show that the replication algorithm is correct because it implements a total order of requests among the replicas, and because each request appears only once in the total order. Wait-freedom and linearizability of the replicated service are demonstrated thereafter.

D.2.1 Total Ordering and Uniqueness of Outcomes

To show that there exists a total ordering of the outcomes, we show that two outcomes decided at the same position by distinct replicas are indeed identical. Uniqueness is proved by showing that two identical outcomes must be decided at the same position in the total order.

Lemma 4 *An outcome stored locally has been previously decided.*

Proof: Assume not and that a process caches an outcome that has not been decided. By the algorithm, we assign an outcome locally at a process only at line 23. For a process to execute this line, it must have exited successfully the loop at line 20, meaning that $\triangle$Register indeed returned a decision value: a contradiction. □

Lemma 5 *If two replicas decide an outcome at the same position, the outcomes are the same.*

PROOF: Let p (resp. q) decide an outcome o_p (resp. o_q) at position num. By the agreement property of $\triangle$Register, $o_p = o_q$. $\square$

In the following, let H_i be the local state of p_i, corresponding to the sequence of length num_i of outcomes locally stored at p_i and $H_i(k)$ the k^{th} outcome of H_i (outcome at position k). Let H_i' be a sequence of outcomes of length $num' \leq num$. If $(\forall k \leq num')(H_i(k) = H_i'(k))$, we say that $H'i$ is a prefix of H_i and we write $H_i' = prefix(H_i)$.

Lemma 6 *For the local states of any two replicas, at least one is a prefix of the other.*

PROOF: Let p and q be two replicas with respective local states H_p of length num_p and H_q of length H_q. We consider the cases $num_p = num_q$ and $num_p < num_q$. The third one is found by symmetry. In the former case, if $num_p = num_q = 0$, then H_q is a trivial prefix of H_p. So, consider $num_p = num_q > 0$. For each position $k \leq num_p$, p and q must have decided an outcome by Lemma 4, and this outcome must be the same for both, by Lemma 5, so $H_p = H_q$ and each one is a prefix of the other. In the second case, H_q must equal H_p up to num_p for the same reasons, so H_q can be written as $H_p \cdot H_q'$ where $\cdot$ represents the concatenation of sequences. By construction, H_p is a prefix of H_q. $\square$

Proposition 7 *Total ordering of outcomes is ensured among replicas' local state.*

PROOF: Let p and q be two replicas with their respective local state H_p of length num_p and H_q of length num_q. Lemma 6 ensures that no matter num_p and num_q, one of H_p or H_q is a prefix of the other. In turn, this implies that a replica uses a total ordering of outcomes consistent with the total order of another replica. By transitivity, this propagates to all the replicas. $\square$

Lemma 8 *If a replica decides an outcome at some position num, this outcome is unique among the* $num - 1$ *outcomes previously decided.*

PROOF: Let r be a replica that commits an outcome o at position num. We reason by induction on the position num of the outcome.

For $num = 1$, the outcome o_1 is obviously unique, as it is the first outcome to be committed in the total order.

By the induction hypothesis, we now assume that the first $num - 1$ outcomes are unique, and that r commits an outcome o_{num} at position num. Assume by contradiction that outcome o_{num} is not unique and consider i the position in the total order of the similar outcome. For such a situation to be possible, r decides outcome o_{num}, but does not propose it at position num (otherwise, it would detect that the outcome is not unique at line 14). This means that another replica r' proposed outcome o_i for the position num. Proposition 7 implies that $o_i^{r'} = o_i^r = o_i$. Thus r' cannot propose outcome o_i at position num. A contradiction. $\square$

Proposition 9 *If two replicas decide the same outcome at positions num and num′ respectively, then* $num = num'$.

PROOF: Let p and q be two replicas that decide the same outcome o at position $num = num_p$ and $num' = num_q$ respectively. Assume towards a contradiction, and without loss of generality, that

$num_p < num_q$. Because a replica increments its variable num by one every time it decides an outcome, q must already have decided an outcome o' for the position num_p. As q decides o for position $num_q (\neq num_p)$, Lemma 8 implies that $o \neq o'$. Because of the agreement property of eventual consensus, p cannot decide o at num_p: a contradiction. For a symmetric reason, we conclude that $num_p = num_q$. □

Total order of outcomes and consistency among replicas' local state follow, from Proposition 7. Uniqueness of committed outcomes follow from Proposition 9.

D.2.2 Wait-Freedom and Linearizability

Proposition 10 *The submit action is wait free.*

PROOF: Assume that the corresponding request is (eventually) sent to the perpetual leader. We have two cases to consider: (i) the request is an old one, or (ii) the request is a new one. For (i), the replica immediately returns (at line 15). For (ii), as the process is the eventual perpetual leader, it will eventually be the only process to propose. By the termination property of eventual consensus, we know that if eventually a single process proposes, it decides. The replica returns the reply to the client after deciding. □

Lemma 11 *An outcome stored at a position has been decided at this position.*

PROOF: By Lemma 4, an outcome is stored if and only if it has been decided. From line 23 and the algorithm in Figure 5, an outcome decided at a position is stored for this position. □

Lemma 12 *Given a run R with n successful submitted unique requests and n corresponding replies. Then R contains exactly n committed outcomes and outcome i has position i.*

PROOF: For a request to have a corresponding reply, either (i) the reply has been cached and is immediately found (ii) the outcome is committed, cached, and then returned by the replica. By Lemma 4, this means that R will contain at least n outcomes. Moreover, a replica cannot propose an outcome that already appears among the committed outcomes (by Lemma 8), and thus, R will contain at most n outcomes. We conclude that R contains exactly n committed outcomes.

To show that outcome i will appear at position $num = i$, we reason by induction on the outcome number. For outcome o_1, as the position num is initialized to one and is incremented after the replica proposes, we know that $num \geq 1$. Assume now by contradiction that $num > 1$. As we have only a single request so far, it means that another replica must have committed the request at position $num - 1$. The fact that the cache contains the same request at position $num - 1$ and num contradicts Lemma 8.

For an outcome $n > 1$, we assume that there is a sequence of outcomes successfully committed, and we let num_n be the position of the outcome committed for request n. We need to show that $num_n = n$. First, assume by contradiction that $num_n < n$. As any previous outcome k has been successfully committed at position $num_k = k$, this would mean that requests are not unique: a contradiction. If we now assume that $num_n > n$, as we only increments num when a request is proposed (and, by assumption, decided), it means that some outcome would have been committed several times, contradicting Lemma 8. We conclude that $num_n = n$. □

Axiom 13 *Given a state machine S. If $\sigma \in \mathsf{PossibleStates}_S(r_1, \ldots, r_n, rep_1, \ldots, rep_n)$ and if $\mathsf{execute}_S(r, \sigma) = (rep, upd)$, then the following holds:*

- $\mathsf{update}_S(\sigma, upd) \in \mathsf{PossibleStates}_S(r_1, \ldots, r_n, r, rep_1, \ldots, rep_n, rep)$.

- $[rep_1, \ldots, rep_n, rep] \in \mathsf{possibleReply}_S(r_1, \ldots, r_n, r)$.

We want now to relate the notion of committed outcome (exposed in Lemma 12) and the update of the state of the replica.

Lemma 14 *Given a run R with n committed outcomes $out_1 = [req_1, 1, rep_1, upd_1], \ldots, out_n = [req_n, n, rep_n, upd_n]$, then the following holds:*

1. $[rep_1, \ldots, rep_n] \in \mathsf{possibleReply}_S(r_1, \ldots, r_n)$
2. *If a replica updates its copy of the object with parameter upd_n, and if σ is the resulting state, then $\sigma \in \mathsf{PossibleStates}_S(r_1, \ldots, r_n, rep_1, \ldots, rep_n)$.*

PROOF: We reason by induction on the outcome number n.

For $n = 1$, a replica p decides outcome 1. This means that a replica q (maybe p itself) proposed this outcome. For q to propose outcome 1, it must be at position $num_q = 1$ (by Lemma 12). Thus, S_q (q's state machine) is in its initial state Λ (because num_q is initialized at 1 and because a replica updates its state (if and) only if it increments its variable num). When q executes request r_1, it produces rep_1 and upd_1. By definition, $rep_1 \in \mathsf{possibleReply}_S(r_1)$ and $\sigma = update(\Lambda, upd_1) \in \mathsf{PossibleStates}_S(r_1, rep_1)$. By Lemma 11, we know that a process p that decides outcome 1 does so for position 1. So p is in initial state and call update with upd_1, and thus $\sigma \in \mathsf{PossibleStates}_S(r_1, rep_1)$.

For $n = k > 1$, let p be some replica. Replica p must have committed outcome $n-1$: for p to propose a new outcome with $num_p = n$, p must increment its variable num_p from $n-1$ to n, and call $update_S(upd_{n-1})$. This means that the induction hypothesis holds, namely:

- $[rep_1, \ldots, rep_{n-1}] \in \mathsf{possibleReply}_S(r_1, \ldots, r_{n-1})$
- If p calls update with upd_{n-1}, then the resulting state $\sigma \in \mathsf{PossibleStates}_S(r_1, \ldots, r_{n-1}, rep_1, \ldots, rep_{n-1})$.

When p executes r_n, its variable num_p equals n. This happens in state σ (p does not change state because num_p does not change and because a replica updates its state only if it increments its variable num). This means that p's state machine is still in state σ. The call to execute produces the rep_n value and the update upd_n. By Axiom 13, we thus have $[rep_1, \ldots, rep_n] \in \mathsf{possibleReply}_S(r_1, \ldots, r_n)$ and $update(\sigma, upd_n) \in \mathsf{PossibleStates}_S(r_1, \ldots, r_n, rep_1, \ldots, rep_n)$. □

As said previously, we say that a replicated service is linearizable if there exist two sequences, seq_r (requests) and seq_{rep} (replies), where seq_r contains the requests and seq_{rep} contains the replies, such that the following conditions are satisfied:

- If $seq_r(k) = r_i^j$ and $seq_r(h) = r_{i+1}^j$, then $h > k$.
- $seq_{rep}(k) = rep_i^j$ if and only if $seq_r(k) = r_i^j$.
- $seq_{rep} \in \mathsf{possibleReply}_S(seq_r)$

We start by showing a lemma to demonstrate the first condition.

Lemma 15 *Consider two requests req_1 and req_2 that do not appear in the total order yet. If a client C_1 successfully submits req_1 before a client C_2 submits req_2, and if there are two committed outcomes $out_1 = [r_1, -, -]$ and $out_2 = [r_2, -, -]$ respectively at position i and j in the total order, then $i < j$.*

PROOF: First, note that $i = j$ contradicts Proposition 7. Now, assume by contradiction that $i > j$, and C_1 sucessfully submits a request req_1 to replica r. r commits req_1 and returns the reply to C_1. Consider i be the position at which r committed req_1. From Lemma 12, r commits a request at position i, if and only if it already committed $i - 1$ requests, in particular a request at position j. When C_2 successfully submits its request to replica r', r' commits req_2 at position j. But this contradicts Proposition 7. We conclude that $i < j$. □

We can now prove the linearizability of the replicated service.

Proposition 16 *A replicated service using the algorithm presented in Figure 4 is linearizable.*

PROOF: From Lemma 12, consider a run R with n committed outcomes. Take this run and construct the two sequences seq_r and seq_{rep} as follows: for a committed outcome o_i at position i (that is, the i^{th} outcome from Lemma 11), let $seq_r(i)$ be the request contained in o_i and let $seq_{rep}(i)$ be the reply contained in o_i.

The first condition is verified from Lemma 15. By construction, it is obvious that $seq_{rep}(k) = rep_i^j = o_i.res$ if and only if $seq_r(k) = r_i^j = o_i.req$. Lemma 14 also applies by construction and so, $seq_{rep} \in \mathsf{possibleReply}_S(seq_r)$. □