ACM SIGACT News Distributed Computing Column 10

Sergio Rajsbaum*

Abstract

The Distributed Computing Column covers the theory of systems that are composed of a number of interacting computing elements. These include problems of communication and networking, databases, distributed shared memory, multiprocessor architectures, operating systems, verification, internet, and the web.

This issue consists of the paper "Deconstructing Paxos" by Romain Boichat, Partha Dutta, Svend Frølund, and Rachid Guerraoui. Many thanks to them for contributing to this issue.

Request for Collaborations: Please send me any suggestions for material I should be including in this column, including news and communications, open problems, and authors willing to write a guest column or to review an event related to theory of distributed computing.

Deconstructing Paxos [1]

Romain Boichat, Partha Dutta
Distributed Programming Laboratory
Swiss Federal Institute of Technology
Lausanne, Switzerland

Svend Frølund
Hewlett-Packard Labs
1501 Page Mill Rd
Palo Alto, USA

Rachid Guerraoui
Distributed Programming Laboratory
Swiss Federal Institute of Technology
Lausanne, Switzerland

Abstract

The celebrated Paxos algorithm of Lamport implements a fault-tolerant deterministic service by replicating it over a distributed message-passing system. This paper presents a deconstruction of the algorithm by factoring out its fundamental algorithmic principles within two abstractions: an eventual leader election and an eventual register abstractions. In short, the leader election abstraction encapsulates the liveness property of Paxos whereas the register abstraction encapsulates its safety property. Our deconstruction is faithful in that it preserves the resilience and efficiency of the original Paxos algorithm in terms of stable storage logs, message complexity, and communication steps. In a companion paper, we show how to use our abstractions to reconstruct powerful variants of Paxos.

*Instituto de Matemáticas, UNAM. Ciudad Universitaria, Mexico City, D.F. 04510 `rajsbaum@math.unam.mx`.

[1]This work is partially supported by the Swiss National Science Foundation (project number 510-207).

The Island of Paxos used to host a great civilisation, which was unfortunately destroyed by a foreign invasion. A famous archaeologist reported on interesting parts of the history of Paxons and particularly described their sophisticated part-time parliament protocol [15]. Paxos legislators maintained consistent copies of the parliamentary records, despite their frequent forays from the chamber and the forgetfulness of their messengers. Although recent studies led to new ways to describe the parliament algorithm [16, 17], as well as powerful tools to reason about its correctness [20], our desire to better understand the Paxon civilisation motivated us to revisit the Island and spend some time deciphering the ancient manuscripts of the legislative system. We discovered that Paxons had precisely codified various aspects of their parliament protocol within specific sub-protocols: one sub-protocol used to ensure the progress of the parliament and one sub-protocol used to ensure its consistency. The precise codification of these sub-protocols helped Paxons adapt their algorithm to various seasons of their parliament.

1 Introduction

The Paxos part-time parliament algorithm of Lamport [15] provides a very practical way to implement a highly-available deterministic service by replicating it over a system of (non-malicious) processes communicating through message passing. Replicas follow the *state-machine* pattern, also called *active replication* [14, 22]. Each replica is supposed to compute every request and return the result to the corresponding client, which selects the first returned result. Paxos maintains safety (replica consistency) by ensuring total order delivery of requests. It does so even during unstable periods of the system, e.g., even if messages are delayed or lost and processes crash and recover. Thus, Paxos is indulgent in the sense of [11]. Paxos ensures liveness (result delivery) whenever the system stabilizes, and it does so in a very efficient way. Paxos involves however many tricky details and it is difficult to factor out the abstractions that comprise the algorithm. Deconstructing Paxos and identifying those abstractions is an appealing objective towards practical implementations of the algorithm, as well as effective reconstructions and adaptations of it in specific settings.

In [17, 20, 16], Paxos was described as a sequence of *consensus* instances. The focus was on consensus which, from a solvability point of view, is indeed equivalent to ensuring total order request delivery [6]. However, providing the efficiency of the original Paxos algorithm goes through breaking the consensus abstraction and combining some of its underlying algorithmic principles with non-trivial techniques such as log piggy-backing and leasing. In particular, this means that the modular correctness proof given in [20] does not apply to the actual (optimized) Paxos algorithm. The goal of our paper is to describe a deconstruction of the Paxos that is *faithful* in that the underlying abstractions do no need to be broken in order to preserve the efficiency of the original Paxos replication scheme.

The key to our faithful deconstruction is the identification of the new notion of $\Diamond$*Register*, which can be implemented with a majority of correct processes in an asynchronous system and does not fall within the FLP impossibility of consensus [10]. We use $\Diamond$Register in conjunction with a $\Diamond$*Leader* abstraction, which captures the exact amount of synchrony needed to ensure agreement among replicas [5]. In short, the leader election abstraction encapsulates the *progress* procedure of Paxos that ensures liveness (i.e., service availability), whereas the register abstraction encapsulates the *state management* procedure of Paxos that ensures safety (i.e., consistency of service).

Our abstractions are both first class citizens at the level of the replication algorithm. Hence, instead of considering Paxos as a sequence of consensus instances [16, 17, 20], we consider it as a sequence of compositions of finer-grained $\diamond$Register and $\diamond$Leader instances. Our deconstruction of Paxos is modular, and yet it preserves the resilience and efficiency of the original algorithm in terms of stable storage logs, message complexity and communication steps. In a companion paper, we show how to use these abstractions to reconstruct powerful variants of Paxos that are customized to specific settings of the distributed environment [3].

The rest of the paper is organised as follows. Section 2 describes the model. Section 3 describes the problem solved by Paxos. Section 4 gives the specification of our abstractions. We show how to implement these specifications in a crash-stop model and get a simple variant of Paxos in Section 5. We then show how to transpose the implementation in a more general crash-recovery model and get the faithful deconstruction of Paxos in Section 6. Section 7 discusses related work and concludes the paper.

2 Model

2.1 Processes

We consider a set of processes $\Pi = \{p_1, p_2, ..., p_n\}$. At any given time, a process is either *up* or *down*. When it is *up*, a process progresses at its own speed behaving according to its specification, i.e., it correctly executes its program. While being up, a process can fail by crashing; it then stops executing its program and becomes *down*. A process that is down can later recover; it then becomes up again and restarts by executing a recovery procedure. The occurrence of a *crash* (resp. *recovery*) event makes a process transit from up to down (resp. from down to up). A process p_i is said to be *unstable* if it crashes and recovers infinitely many times. We define an *always-up* process as a process that never crashes. We say that a process p_i is *correct* if there is a time after which the process is permanently up.[2] A process is *faulty* if it is not *correct*, i.e., either *eventually always-down* or *unstable*.

A process is equipped with two local memories: a volatile memory and a stable storage. The primitives *store* and *retrieve* allow a process that is up to access its stable storage. When it crashes, a process loses the content of its volatile memory; the content of its stable storage is however not affected by the crash and can be retrieved by the process upon recovery.

2.2 Link Properties

Processes exchange information by *sending* and *receiving* messages through channels. We assume the existence of a bidirectional channel between every pair of processes. We assume that every message m includes the following fields: the identity of its sender, denoted *sender(m)*, and a local identification number, denoted *id(m)*. These fields make every message unique throughout the whole life of the process, i.e., two distinct messages sent by the same process cannot have the same id even after the sender crash and recover between the two send events. Channels can lose messages and there is no upper bound on message transmission delays. We assume channels that ensure the following properties between every pair of processes p_i and p_j:

- **No creation:** If p_j receives a message m from p_i, then p_i has sent m to p_j.

[2] The validity period of this definition is the duration of an execution, i.e., in practice, a process is correct if it eventually remains up long enough for the algorithm to terminate.

- **Fair loss:** If p_i sends a message m to p_j an infinite number of times and p_j is correct, then p_j receives m from p_i an infinite number of times.

The last property is sometimes called *weak loss* [18]. It reflects the usefulness of the communication channel. Without this property, no interesting distributed problem would be solvable.

We assume the presence of a discrete global clock whose tick range $\mathcal{T}$ is the set of natural numbers. This clock is used to simplify presentation and not to introduce time synchrony, since processes cannot access the global clock. We will indeed introduce some partial synchrony assumptions (otherwise, fault-tolerant agreement, and hence, total order are impossible [10]), but these assumptions are encapsulated inside our $\diamond$*Leader* abstraction and used only to ensure progress (liveness) of the replication algorithm.

3 Problem

The Paxos algorithm coordinates a set of replica processes so that they behave in a consistent way. More precisely, the main problem solved by Paxos is to ensure total order delivery of messages, i.e., total ordering of requests broadcast to replicas. This problem can be precisely defined by two primitives: *TO-Broadcast* and *TO-Deliver*. We say that a process TO-Broadcasts a message m when it invokes *TO-Broadcast* with m as an input parameter. We say that a process TO-Delivers a message m when it returns from the invocation of *TO-Deliver* with m as an output parameter. Total order broadcast satisfies the following properties (inspired by [12]):

- **Termination:** If a process p_i TO-Broadcasts a message m and then p_i does not crash, then p_i eventually TO-Delivers m.

- **Agreement:** If a process TO-Delivers a message m, then every correct process eventually TO-Delivers m.

- **Validity:** For any message m, (1) every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process, and (2) every process p_i TO-Delivers m at most once.

- **Total order:** Let p_i and p_j be any two processes that TO-Deliver some message m. If p_i TO-Delivers some message m' before m, then p_j also TO-Delivers m' before m.[3]

4 Abstractions: Specifications

Our deconstruction of the Paxos replication algorithm is based on two main building blocks: a $\diamond$*Register* and a $\diamond$*Leader* abstractions. Roughly speaking, Paxos ensures that all replica processes agree on the order associated with every request message. The $\diamond$Register abstraction encapsulates the algorithm used to "store" and "lock" the agreement value (i.e., the order), whereas $\diamond$Leader encapsulates the algorithm used to eventually choose a unique leader that succeeds in storing and locking a final agreement value in the register.

[3]The total order property we consider here is slightly stronger than the one introduced in [12]. In [12], it is stated that, if any two processes p_i and p_j *both TO-Deliver messages* m *and* m', then p_i TO-Delivers m before m' if and only if p_j TO-Delivers m before m'. With this property, nothing prevents a process p_i from TO-Delivering the sequence of messages, say $m_1; m_2; m_3$, whereas another (faulty) process TO-Delivers $m_1; m_3$ without ever delivering m_2. Our specification clearly excludes that scenario and more faithfully captures the (uniform) guarantee offered by Paxos.

$\diamond$Register and $\diamond$Leader are "shared memory" like abstractions that export operations invoked by the processes implementing the replicated service. We give here the specifications of these abstractions, and we illustrate their use through a simple consensus algorithm using these abstractions. (Implementations of these abstractions are given in Section 5 and 6.)

As in [13], we say that an operation invocation inv_2 *follows* (is *subsequent* to) an operation invocation inv_1, if inv_2 was invoked after inv_1 has returned. Otherwise, the invocations are *concurrent*.

4.1 $\diamond$Register

The $\diamond$*Register* provides a distributed storage facility with a *write-once* semantics; it encapsulates the act of locking a value in Paxos. The act of storing a value in the register might however fail: if several processes try to store some value in the register, none of them might succeed.

Processes access $\diamond$Register through a single primitive *propose*(). A process invokes *propose*() with a single argument $v \in Values$, where $Values$ is the set of possible values that can be stored in the register. The primitive returns a value in $Values \cup \{abort\}$ ($abort \notin Values$). If a process p_i invokes *propose*(v) we say that p_i *proposes* v; if *propose*() at p_i returns $v' \neq abort$, we say that p_i *decides* v'; if *propose*() at p_i returns *abort*, we say that p_i *aborts*.

$\diamond$Register satisfies the following three properties:

- **Validity:** If a process decides a value v, then v was proposed by some process.
- **Agreement:** No two processes decide differently.
- **Termination:** (1) If a process proposes, it either crashes or returns from the invocation, and (2) if a single correct process proposes an infinite number of times, it eventually decides.

Notice that the validity and agreement properties of $\diamond$Register are identical to those of consensus [10]. The termination property is strictly weaker: if more than one correct processes keeps on proposing values, none of them is ever guaranteed to decide. As we will see in Section 5.1, this weaker termination property makes it possible to implement $\diamond$Register in an asynchronous system (with a majority of correct processes), in spite of consensus being impossible in such a model [10].

To illustrate the behaviour of $\diamond$Register, consider the example depicted in Figure 1. Three processes p_1, p_2 and p_3 access the same $\diamond$Register. Process p_1 invokes *propose*(X) and process p_2 invokes *propose*(Y) concurrently. Both invocations return *abort*. Then p_3 invokes *propose*(Z) which returns Y: p_3 decides Y. Then p_1 invokes *propose*(X): this invocation returns Y but it could also have returned *abort*.

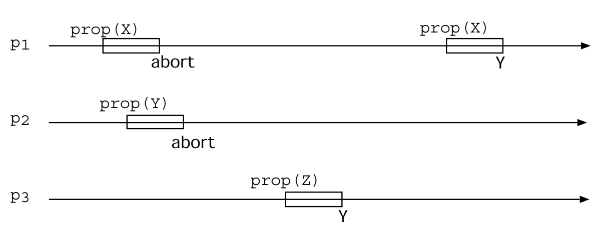


Figure 1: $\diamond$Register scenario

4.2 ◇Leader

Intuitively, the ◇*Leader* abstraction is a shared object that elects a leader among a set of processes. It encapsulates the sub-protocol used in Paxos to choose a process that decides on the ordering of messages. The ◇Leader abstraction has one operation, named *leader*(), which returns a process identifier, denoting the current leader. When the operation returns p_j at time t and at process p_i, we say that p_j is leader for p_i at time t (or p_i elects p_j at time t). ◇Leader satisfies the following three properties:

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

It is important to notice that the properties above do not prevent the case where, for an arbitrary period of time, various processes are simultaneously leaders.[4] However, there must be a time after which the processes agree on some unique correct leader. Figure 2 depicts a scenario where every process elects process p_1, and then p_1 crashes; eventually every process elects process p_2.

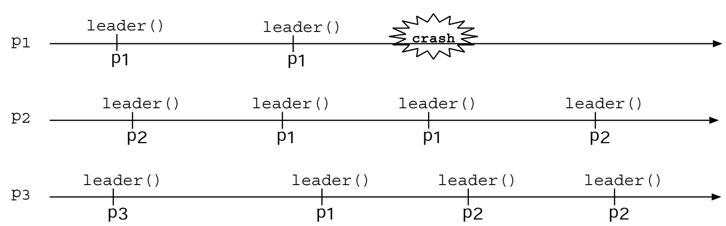


Figure 2: ◇Leader scenario

4.3 Illustration: Consensus

To illustrate the semantics of our abstractions, we show in Figure 3 how they can be composed to build a *simple* consensus algorithm. In a sense, this also shows that, together, and from a computational point of view, our abstractions are at least as powerful as consensus.

Every process is supposed to propose a value and some process eventually decides some value. A process proposes a value v through invoking *propose*(v) and is said to have decided v', if the invocation returns v'. A consensus algorithm satisfies the following properties: (1) (*Validity*) if a process decides v then some process has proposed v, (2) (*Uniform Agreement*) no two processes decide differently, and (3) (*Termination*) some correct process eventually decides.

The algorithm uses a single ◇Register instance for ensuring agreement. After proposing a value v, any process that is elected leader invokes *propose*(v) (on the ◇Register) and keeps on doing so, unless it stops being the leader or it decides (i.e., the *propose*(v) invocation on the ◇Register returns a non-*abort* value, or p_i receives the decision value from some other process).

We assume here that some process is correct. The agreement and the validity properties of consensus follow directly from those of ◇Register. To see how the termination property of consensus is ensured, recall that the ◇Leader eventually elects the same correct process p_l at all processes.

[4] In this sense our leader election specification is strictly weaker then the notion of leader election introduced in [21].

Suppose by contradiction that no correct process ever decides. It follows that p_l never decides. Thus, process p_l invokes *propose*() on the $\diamond$Register an infinite number of times (without deciding) and all other processes invoke *propose*() only a finite number of times: a violation of the termination property of $\diamond$Register .

```
For each process p_i:
procedure initialization:
   register ← new ◇Register
   ld ← new ◇Leader
   decision ← abort
procedure propose(v)
   while decision = abort do
      if ld.leader() = p_i then
         decision ← register.propose(v)
   return(decision)
```

Figure 3: Consensus

5 Abstractions: Implementations

In the following, we discuss implementations of our two abstractions, $\diamond$Register and $\diamond$Leader, and then use these abstractions to implement a simple variant of Paxos in the crash-stop model (i.e., solve the total-order problem in the crash-stop model). The architecture of this implementation is shown in Figure 4. Section 6 extends this implementation to the crash-recovery model to obtain a faithful deconstruction of Paxos.

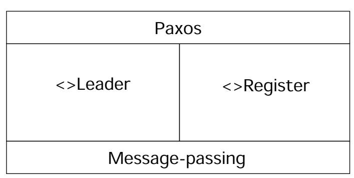


Figure 4: Architecture

In the crash-stop model, we assume that messages are not lost or duplicated (although they may be arbitrarily delayed) and processes that crash halt their activities and never recover. We also assume that a majority of the processes never crash, and for the implementation of our $\diamond$Leader abstraction, we assume the failure detector Ω introduced in [5].

5.1 $\diamond$Register

5.1.1 Overview

The algorithm of Figure 5 implements the abstraction of $\diamond$Register. The algorithm works intuitively as follows. Every process plays two distinct roles: initiator (when it has some value to propose) and witness (when some process proposes some value). The *propose*() procedure captures the initiator

```
procedure ◇Register()                    {Constructor, for each process p_i}
   read_i ← 0                            {Highest READ round number accepted by p_i}
   write_i ← 0                           {Highest WRITE round number accepted by p_i}
   val_i ← ⊥                             {p_i's estimate of the register value}
   v* ← ⊥                                {value with which WRITE message is sent}
   k ← i − n                             {round number, initialized such that the first round number sent by p_i is i}
procedure propose(v)
   k ← k + n
   send [READ,k] to all processes
   wait until received [ackREAD,k,*,*] or [nackREAD,k] from ⌈(n+1)/2⌉ processes
   if received at least one [nackREAD,k] then
      return(abort)                      {propose() aborts after READ}
   else
      select the [ackREAD,k,k′,val′] with the highest k′
      if val′ ≠ ⊥ then
         v* ← val′
      else
         v* ← v
      send [WRITE,k,v*] to all processes
      wait until received [ackWRITE,k] or [nackWRITE,k] from ⌈(n+1)/2⌉ processes
      if received at least one [nackWRITE,k] then
         return(abort)                   {propose() aborts after WRITE}
      else
         return(v*)                      {p_i decides v*}
task wait until receive [READ,k] from p_j
   if write_i ≥ k or read_i ≥ k then
      send [nackREAD,k] to p_j
   else
      read_i ← k
      send [ackREAD,k,write_i,val_i] to p_j
task wait until receive [WRITE,k,v′] from p_j
   if write_i > k or read_i > k then
      send [nackWRITE,k] to p_j
   else
      write_i ← k
      val_i ← v′                         {A new value is “adopted”}
      send [ackWRITE,k] to p_j
```

Figure 5: A ◇Register implementation in a crash-stop model

role. The witness role is performed by the two message-reception tasks which reply to messages generated by *propose*() invocations. The message-reception tasks at each process p_j maintain an estimate of the register's value, denoted by val_j and intialized to $\perp$. Every process p_j maintains the current round number k. The first *propose*() invocation at process p_j has round number j, and in every subsequent invocation, the round number is incremented by n. Thus, the round number associated with invocations at process p_j are: $j, j+n, j+2n$, and so on.[5]

Procedure *propose*() at a process p_j consists of two phases: *read* and *write*. Each phase can succeed or abort, and the write phase is initiated only if the read phase succeeds. Procedure *propose*() returns a decision value if both phases succeed, otherwise, it aborts.

- The purpose of the read phase is to detect any value which has already been decided (i.e., has been successfully written) and to get a promise from the witnesses that no subsequent read or write will succeed with a lower round number [16]. Process p_j initiates the read phase by incrementing its round number k, and sending a READ message with round number k to all processes. On receiving a READ message, a witness p_i replies nackREAD if it has already seen a READ or WRITE message with the same or a higher round number. Otherwise, p_i sets $read_i$ to k, and sends an ackREAD containing val_i (estimate of the written value) and $write_i$ (round number when val_i was last updated): we say that the witness accepts the READ message. On receiving replies from the witness tasks at majority of processes, the initiator p_j aborts if it receives any nackREAD message. Otherwise, p_j selects the val ($\neq \perp$) with the highest $write$ and proceeds to the write phase. However, if all received replies have $val = \perp$, p_j selects the invocation value of *propose*(). We denote the value selected for the write phase by v^*.

- The write phase tries to update val and $write$ at the witness tasks, to v^* and k, respectively. In the write phase, the initiator p_j sends a WRITE message to all processes with v^*. On receiving a WRITE message, a witness p_i replies nackWRITE if it has seen a READ or WRITE with a higher round number. Otherwise, p_i updates val_i to v^* and $write_i$ to k, and sends ackWRITE to p_j: we say that p_i accepts the WRITE message and adopts v^*. On receiving replies from a majority of processes, p_j aborts if it receives any nackWRITE. Otherwise, p_j returns v^* as the decision value.

5.1.2 Correctness

The validity property of $\Diamond$Register follows from (1) the observation that a process only tries to write (and decide on) a value which it has either proposed or has read from other processes and (2) the no creation property of the channels. The termination property follows from the assumptions that messages are not lost and that a majority of processes never crash, as well as the use of an increasing round number. Now we discuss agreement.

A process decides a value v in a given invocation only if both the read and the write phases of the invocation succeed and the most recent value seen in the read phase is v. Let us call an invocation *writer* if it sends at least one WRITE message. We say that a writer tries to write v at round k, if it sends a [WRITE, k, v] message. It is easy to see that (1) every writer invocation has a successful read phase, (2) any invocation in which some process decides is a writer invocation, and (3) only writer invocations can change the estimates val at the message-reception tasks.

[5] We would like to point out that incrementing the round number by n is an optimization; processes may increment the round number by any finite positive integer, and its value can vary between invocations and from process to process.

Notice that two successful read phases always overlap at the witness task of some process, and therefore, no two successful read phases have the same round number. It immediately follows that two writer invocations cannot have the same round number. Let k' be the lowest round number in which some process decides. Let v' and W1 be the corresponding decision value and the writer invocation,respectively. Therefore, there is a set of processes S which contain a majority of processes, and every process in S replies ackWRITE to W1. We claim that every writer with a higher round number tries to write v'. This claim implies agreement, because only writer invocations can decide, and a writer invocation decides on the value it tries to write.

Suppose by contradiction that there is a writer invocation which tries to write a value different from v'. Consider the lowest round number $k'' > k'$ such that some writer invocation W2 at round k'' tries to write $v'' \neq v'$. Thus, every writer invocation with round k such that $k' \leq k < k''$, tries to write v'. Consider the read phase of W2. Since set S contains a majority of processes, W2 receives [ackREAD , k'', $write'_i$, val] from some process p_i in S. As the message is an ackREAD, from line 26 we have $write'_i < k''$.

Let $t1$ be time when p_i replied ackWRITE to W1 and let $t2$ be the time when p_i replied ackREAD to W2. We claim that $t1 < t2$. Suppose by contradiction that $t1 > t2$. ($t1 \neq t2$ because we assume that the message reception tasks at a process treats one message at a time.) Then, p_i sets $read_i$ to k'' at $t2$. Since, $read_i$ can never decrease with time (lines 26 and 29), $read_i \geq k'' > k$ at $t1$. Thus, from line 32, it follows that p_i sends a nackWRITE to W1; a contradiction.

As p_i sends a ackWRITE to $W1$, so $write_i$ is k' at $t1$. Also, from the message sends by p_i to W2 we know that $write_i$ is $write'_i$ at $t2$. As $t1 < t2$ and $write_i$ can never decrease with time (lines 32 and 35), we have $k' \leq write'_i$.

Thus, $k' \leq write'_i < k''$. Consider the ackREAD message from which W2 choose the value for the write phase: [ackREAD , k'', l, v'']. (As W2 receives non-$\perp$ value from p_i, W2 cannot choose its own invocation value as the value for the write phase.) Since a writer chooses the latest value seen in the READ phase, $write'_i \leq l$. Furthermore, since it is an ackREAD message, from line 26 we have $l < k''$. Thus there is a writer which tried to write v'' at round l and $k' \leq write'_i \leq l < k''$. Recall that every writer at round k such that $k' \leq k < k''$, tries to write v'. Thus, $v' = v''$; a contradiction.

5.2 $\Diamond$Leader

In the presence of at least one correct process, $\Diamond$Leader corresponds to the failure detector Ω introduced in [5]: Ω outputs (at each process) exactly one process called a *trusted* process, i.e., a process that is trusted to be up. Failure detector Ω satisfies the following property: *There is a time after which all correct processes trust the same correct process.* It was shown in [5] that Ω is the weakest failure detector to solve consensus and hence total order broadcast in a crash-stop system model with a majority of correct processes. The Ω failure detector can be implemented in a message passing system with partial synchrony assumptions [6].

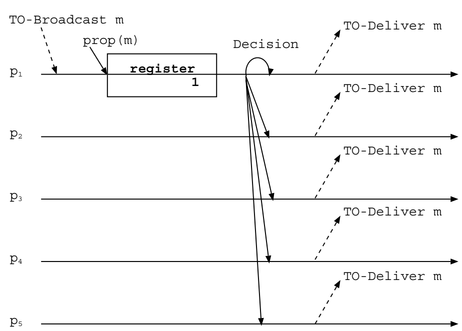
5.3 A Simple Variant of Paxos

The algorithm of Figure 7 is a simple and modular variant of Paxos in a crash-stop model (whereas the original Paxos algorithm considers a crash-recovery model - see Section 6).

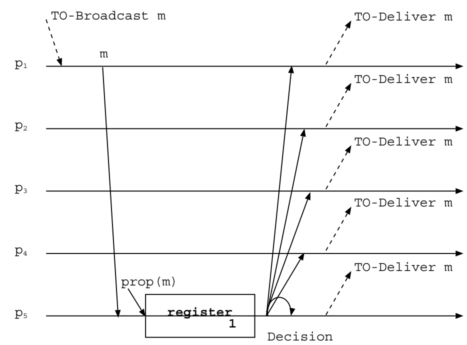
5.3.1 Overview

Processes deliver messages in batches, and in increasing order of their batch numbers: messages in batch L are delivered before messages in batch $L+1$. Inside a batch, messages are delivered in a deterministic order (e.g., lexicographically). For each batch, if processes can agree on the set of messages which constitutes that batch, then the ordering of batches immediately implies the total-ordering of messages.

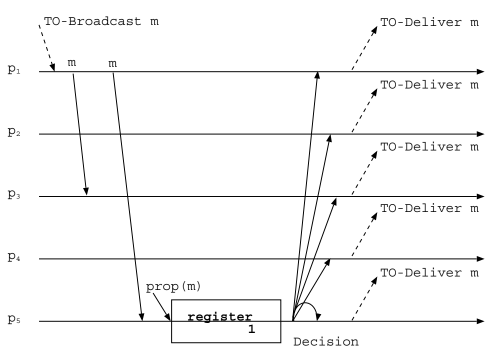
The algorithm uses a series of $\Diamond$Registers (we simply say registers) indexed by batch numbers to agree on the set of messages for each batch: register$_L$ is used to agree on the set of messages which constitute batch L, and we call this agreed set of messages the *decided message set* for batch L.



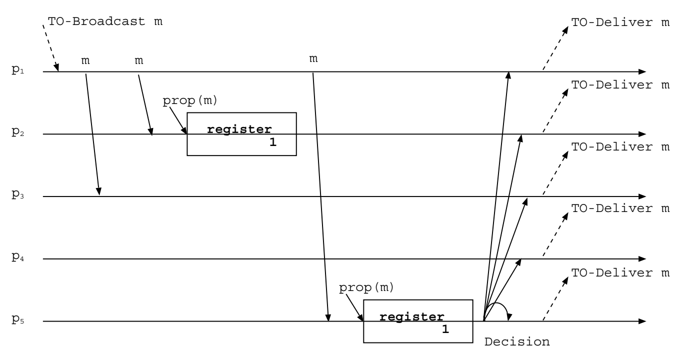
(a) p_1 is leader



(b) p_5 is leader



(c) p_1 first elects p_3 and then p_5



(d) p_1 first elects p_3, then p_2 and finally p_5

Figure 6: Execution schemes

We give here an intuitive description of the algorithm. When a process p_i TO-Broadcasts a message m (i.e., m is supposed to contain a request to the replicated service), p_i consults $\Diamond$Leader and sends m to the leader, say p_j. When p_j receives a TO-Broadcast message m, p_j checks whether the local $\Diamond$Leader module elects itself as the leader. If p_j indeed elects itself as the leader, p_j triggers a *Converge* task to decide on the next higher batch of messages, say batch L. The *Converge* task repeatedly invokes propose() on register$_L$, until it decides on some set of messages,[6] or p_j stops

[6]Recall that a propose() invocation decides when it returns a non-*abort* value.

being leader. The proposal value for each invocation is the set of messages which have been received by p_j but not yet delivered. If p_j decides on a message set for batch L, it sends the decision value to all processes. On receiving a decided message set for batch L, a process delivers the message set as batch L if it has already delivered batch L-1. If it has not yet delivered batch L-1, the process delays the delivery of batch L until batch L-1 is delivered to respect the total order. Within a batch, processes deliver messages in the decided message set using a deterministic ordering function.

5.3.2 Examples

Figure 6 depicts four possible execution schemes of the algorithm. We assume for all cases that (1) process p_1 TO-Broadcasts a message m, (2) process p_5 is the eventual perpetual leader, and (3) L =1. (*prop*() stands in the figures for *propose*(), and since $L = 1$, all *propose*() invocations are on the register$_1$.) In Figure 6(a), p_1 elects itself, triggers the task $Converge(1, m)$ (which in turn invoked *propose*(m) on register$_1$), decides m, and sends the decision to all. In Figure 6(b), p_1 elects p_5 and sends m to p_5. Process p_5 then invokes *propose*(m) on register$_1$, decides m, then sends the decision to all. In Figure 6(c), p_1 first elects p_3 and sends m to p_3. In this case however, p_3 does not elect itself and therefore does nothing. Later on, p_1 elects p_5 and then sends m to p_5. As for case (b), p_5 decides m and sends the decision to every process. Note that p_3 could have sent m to p_5 if p_3 had elected p_5. Finally, in Figure 6(d), p_1 elects p_3 (which does not elect itself), then p_1 elects p_2, which elects itself and invokes *propose*(m) but aborts. Finally, p_1 elects p_5, and, as for case (c), p_5 decides m and sends the decision to all.

5.3.3 Detailed description

We give here a detailed description of the algorithm. We first describe the primary data structure, and then the main parts of the algorithm. Each process p_i maintains a variable *TO_delivered[]* which is an array of message sets which have already been TO-Delivered, and indexed according to their batch number. For ease of description, we denote the set of all messages in *TO_delivered[]* by *TO_delivered*. When p_i receives a message m, p_i adds m to the set *Received* which keeps track of all messages that need to be TO-Delivered. Thus *Received* - *TO_delivered*, denoted *TO_undelivered*, contains the set of messages that were submitted for total order broadcast, but are not yet TO-Delivered. The batches that have been decided but not yet TO-Delivered are put in the array *AwaitingToBeDelivered[]* indexed by their batch number. The variable *nextBatch* keeps track of the next expected batch in order to respect the total order property.

There are four main parts in the algorithm: (a) when a process p_i receives some message, task *launch* starts[7] task *Converge* if the process p_i is leader, or if p_i is not leader, sends the messages it did not yet TO-Deliver to the leader; (b) task *Converge* keeps on invoking *propose*() on a $\Diamond$Register while p_i is leader and until some message set is decided; (c) primitive *receive* handles received messages, and stops task *Converge*$(L, *)$ once p_i receives a decision for batch L; and (d) primitive *deliver* TO-Delivers messages. Each part is described below in more details. Initially, when a process p_i TO-Broadcasts a message m, p_i puts m into the set *Received* which has the effect of changing the predicate of guard line 15.

[7] When we say that a new task is started, we mean a new instance of the task with its own variables (since there can be more than one batch of messages being treated at the same time).

- In task *launch*, process p_i triggers the upon case when the set *TO_undelivered* contains new messages or when p_i elects another leader (line 15). Note that the upon case is executed only once per received message to avoid multiple batches with exactly the same set of messages. If the upon case is triggered by a leader change, p_i jumps directly to line 26 and sends to the leader all the messages it did not yet TO-Deliver. Otherwise, before trying to decide the set of messages for the next expected batch, p_i first verifies at line 16, (1) if it has already received the decision for this batch, or (2) if it has already TO-Delivered this batch. Process p_i then verifies whether it is a leader, and if so, p_i starts task *Converge*. The task *Converge* takes in as parameters the message set *TO_undelivered* and the batch number p_i wants to deliver next (*nextBatch*). If p_i is not leader, then p_i sends the message set *TO_undelivered* to the leader.

- In task *Converge*$(L, msgSet)$, a process p_i periodically invokes $propose()$ on the register$_L$ until some message set is decided or p_i stops electing itself the leader. By the property of $\Diamond$Leader, eventually one of the correct processes (p_l) will be the perpetual leader at all processes. Once p_l is elected by every process, for every subsequent batch, (1) p_l directly receives every message TO-Broadcast by processes,(2) is the only process to invoke $propose()$ on the register corresponding to that batch, and therefore, decides a message set. (If p_l does not decide, then p_l proposes an infinite number of times the value to the register without deciding, and other processes propose only a finite number of times to the register; a violation of termination.) Process p_l then sends the decision message for the batch to all.

- In the primitive *receive*, when process p_i receives a decision message from p_j for batch L (line 35), p_i first stops task *Converge*$(L, *)$. Process p_i then verifies that the decision received is the next decision that was expected (*nextBatch*). Otherwise, there are two cases to consider: (1) p_i is ahead, or (2) p_i is lagging behind. For the first case (if p_i is ahead, i.e., p_i receives a decision from a lower batch), p_i sends to p_j an UPDATE message for each batch that p_j is missing (line 39). For case 2 (if p_i receives a future batch), p_i buffers the messages of the batch in the set *AwaitingToBeDelivered* and p_i also sends to p_j an UPDATE message with *nextBatch*-1 (line 41) in order to update itself (p_i): when p_j receives this "on purpose lagging" message, p_j sends to p_i the UPDATE message for all missing batches.

- In the primitive *deliver*, process p_i TO-Delivers only the messages that were not already TO-Delivered (line 9 or 12) following the same deterministic order. We assume that p_i removes all messages that appear twice in the same batch of messages.[8]

6 A Faithful Deconstruction of Paxos

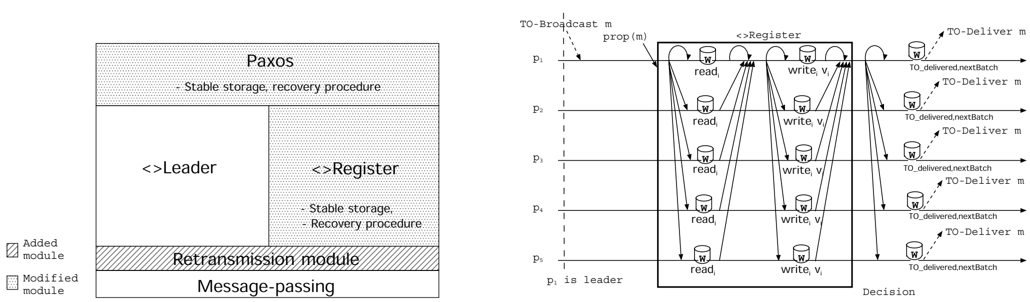
We show here how to step from our simple variant of Paxos in the crash-stop model to a *faithful* variant of the algorithm in the crash-recovery model. The variant we obtain preserves the efficiency of Paxos and tolerates temporary crashes of links and processes. As in the original Paxos algorithm, we assume that there are no *unstable* processes: either processes are eventually always-up (correct) or eventually always-down (faulty). (We show how to circumvent this restriction in [3].)

[8] In [15, 16], TO-Delivering a message and executing a round (in the *Converge()* task) are referred to as passing a decree and conducting a ballot, respectively.

```
1: For each process p_i:
2: procedure initialization:
3:    ld ← new ◇Leader; Received ← ∅; ∀l, TO_delivered[l] ← ∅; ∀l, AwaitingToBeDelivered[l] ← ∅
4:    TO_undelivered ← ∅; K ← 1; nextBatch ← 1 start task{launch}
5: procedure TO-Broadcast(m)
6:    Received ← Received ∪ m
7: procedure deliver(msgSet)
8:    TO_delivered[nextBatch] ← msgSet - TO_delivered
9:    atomically TO-Deliver each message in TO_delivered[nextBatch] in some deterministic order     {TO-Deliver}
10:   nextBatch ← nextBatch +1
11:   while AwaitingToBeDelivered[nextBatch] ≠ ∅ do
12:      TO_delivered[nextBatch] ← AwaitingToBeDelivered[nextBatch] - TO_delivered;
         atomically deliver TO_delivered[nextBatch] in some deterministic order     {TO-Deliver}
13:      nextBatch ← nextBatch+1
14: task launch
15:   upon Received - TO_delivered ≠ ∅ or leader has changed do {Upon case executed only once per received message.
      If upon triggered by a leader change, jump to line 26.}
16:      while AwaitingToBeDelivered[K+1] ≠ ∅ or TO_delivered[K+1] ≠ ∅ do
17:         K ← K+1
18:         if K = nextBatch and AwaitingToBeDelivered[K] ≠ ∅ and TO_delivered[K] = ∅ then
19:            deliver(AwaitingToBeDelivered[K])
20:      TO_undelivered ← Received - TO_delivered
21:      if leader() = p_i then
22:         while Converge(K, *) is active do
23:            K ← K+1
24:         start task Converge(K, TO_undelivered)
25:      else
26:         send(TO_undelivered) to ld.leader()
27: task Converge(L, msgSet)     {Keep on trying until some value is decided}
28:   returnedMsgSet ← abort
29:   register_L ← new ◇Register()
30:   while returnedMsgSet = abort do
31:      if ld.leader() = p_i then
32:         returnedMsgSet ← register_L.propose(msgSet)
33:   send(DECISION, L, returnedMsgSet) to all processes
34: upon receive m from p_j do
35:   if m = (DECISION, K_{p_j}, msgSet^{K_{p_j}}) or m = (UPDATE, K_{p_j}, TO_delivered[K_{p_j}]) then
36:      if task Converge(K_{p_j}, *) is active then stop task Converge(K_{p_j}, *)
37:      if K_{p_j} ≠ nextBatch then     {p_j is ahead or behind}
38:         if K_{p_j} < nextBatch then     {p_j is behind}
39:            for all L such that K_{p_j} < L < nextBatch: send(UPDATE, L, TO_delivered[L]) to p_j     {If p_j ≠ p_i}
40:         else
41:            AwaitingToBeDelivered[K_{p_j}] = msgSet^{K_{p_j}}; send(UPDATE, nextBatch-1, TO_delivered[nextBatch-1]) to p_j
                                                                                                     {If p_j ≠ p_i}
42:      else
43:         deliver(msgSet^{K_{p_j}})
44:   else
45:      Received ← Received ∪ set of messages contained in m
```

Figure 7: A crash-stop variant of Paxos

To step from a crash-stop model to a crash-recovery model, we mainly adapt the implementation of ◇Register and slightly modify the global algorithm to deal with recovery (in shade in Figure 8(a)): we only present the modified parts in this section. Every process stores some values in the stable storage so that it can consistently retrieve its state when it recovers. To cope with temporary link failures, we use a *retransmission* module with two primitives *s-send* and *s-receive*. We describe this module below.



(a) Modules (b) Added stable storage logs

Figure 8: From crash-stop to crash-recovery

6.1 Retransmission Module

This module (Figure 9) encapsulates retransmission issues to deal with temporary crashes of communication links. The primitives of the retransmission module (s-send and s-receive) preserve the no creation and fair loss properties of the underlying channels, and ensures the following property: *Let p_i be any process that s-sends a message m to a process p_j, and then p_i does not crash. If p_j is correct, then p_j s-receives m from p_i an infinite number of times.* Figure 9 gives the algorithm of the retransmission module. All messages that need to be retransmitted are put in the variable *xmitmsg*. As in the original Paxos algorithm, once a process TO-delivers the message set of a batch, all corresponding messages in *xmitmsg* of the used retransmission module could be erased, except DECISION and UPDATE messages.

6.2 ◇Register

We give in Figure 10 the implementation of a ◇Register in a crash-recovery model. The main differences with our crash-stop implementation given in the previous section (Figure 5) are the following: (a) as shown in Figure 8(b), a process stores the variables $read_i$, $write_i$ and val_i, in order to recover consistently its precedent state after a crash, (b) a recovery procedure at p_i re-initializes the process and retrieves all variables, and (c) the send (resp. receive) primitive is also replaced by the s-send (resp. s-receive) primitive.

However, there are two problems which arise in this model due to the generation of duplicate messages by the retransmission module, each of which can be addressed by considering the identifier of messages. (Recall that every message has a unique identifier.)

```
for each process p_i:
procedure initialization:
   xmitmsg ← ∅; start task{retransmit}
procedure s-send(m)                                              {To s-send m to p_j}
   if m ∉ xmitmsg then                {Ensure that m is not added to xmitmsg more than once}
      xmitmsg ← xmitmsg ∪ m
   if p_j ≠ p_i then
      send m to p_j
   else
      simulate s-receive m from p_i
upon receive(m) from p_j do
   s-receive(m)
task retransmit                                  {Periodically retransmit all messages}
   while true do
      for all m ∈ xmitmsg do
         s-send(m)
```

Figure 9: Retransmission module

- Suppose that some correct process p_l keeps on invoking *propose*() and no other process invokes *propose*(). In the read phase of each invocation, retransmission module at p_l s-sends the corresponding READ message. On receiving the first READ message, a process p_i replies ackREAD, whereas on receiving the second message (the duplicate message generated by the retransmission module), the process replies nackREAD (line 26). Since the channels are not FIFO, the nackREAD may be received by p_l before the ackREAD. This scenario may be repeated infinitely often, thus violating termination. To avoid this problem, before sending an ackREAD, a process stores (logs) the message identifier of the READ message along with its *read* variable (line 29). Subsequently, it replies ackREAD to every READ message with the same identifier.

- Since a process can crash and recover, a process p_l may propose a value v with a round number k, crash before completing the invocation, recover, and then propose v' ($\neq v$) with the same round number k. It is possible that other processes reply ackREAD to the first READ message, and send nackREAD to the second message. Since the round number is the same in both invocations, p_l may complete the read phase of second invocation on receiving the ackREAD messages for the previous invocation, and then p_l may send a WRITE message. Thus there may be WRITE messages with the same round number k but containing distinct values, v and v', which may lead again to a violation of agreement. This problem can also be avoided by tagging each ack or nack message with the message identifier of the READ or WRITE message which has generated it. An invocation only considers the replies of its own READ or WRITE messages.

6.3 ◇Leader

The ◇Leader abstraction corresponds to failure detector Ω in the crash-recovery model. Although Ω has only been defined in a crash-stop model [5], its definition (*i.e.; there is a time after which all correct processes trust the same correct process*) does not change in a crash-recovery model. Notice that the notion of correctness of a process changes from that in the crash-stop model. We give in [2] an implementation of Ω in a partial synchrony model.

```
procedure ◇Register()                                   {Constructor, for each process p_i}
   read_i ← 0
   write_i ← 0
   val_i ← ⊥
   v* ← ⊥
   k ← i − n
procedure propose(v)
   k ← k + n
   s-send [READ,k] to all processes
   wait until s-received [ackREAD,k,*,*] or [nackREAD,k] from ⌈(n+1)/2⌉ processes
   if s-received at least one [nackREAD,k] then
      return(abort)
   else
      select the [ackREAD,k, k′, val′] with the highest k′
      if val′ ≠ ⊥ then
         v* ← val′
      else
         v* ← v
      s-send [WRITE,k, v*] to all processes
      wait until s-received [ackWRITE,k] or [nackWRITE,k] from ⌈(n+1)/2⌉ processes
      if s-received at least one [nackWRITE,k] then
         return(abort)
      else
         return(v*)
task wait until s-receive [READ,k] from p_j
   if write_i ≥ k or read_i ≥ k then
      s-send [nackREAD,k] to p_j
   else
      read_i ← k; store{read_i}                          {Modified from Figure 5}
      s-send [ackREAD,k, write_i, val_i] to p_j
task wait until s-receive [WRITE,k, v] from p_j
   if write_i > k or read_i > k then
      s-send [nackWRITE,k] to p_j
   else
      write_i ← k
      val_i ← v; store{write_i, val_i}                   {Modified from Figure 5}
      s-send [ackWRITE,k] to p_j
upon recovery do                                         {Added procedure to Figure 5}
   initialization
   retrieve{write_i, read_i, val_i}
```

Figure 10: A ◇Register implementation in a crash-recovery model

6.4 Paxos

The deconstructed Paxos algorithm is described in Figure 11. Figure 8(b) depicts the fact that, compared to the crash-stop variant of Paxos, the crash recovery algorithm simply adds (1) a recovery procedure, and (2) one stable storage log to store the set *TO_delivered* and the variable *nextBatch* while TO-delivering a batch. Also, the send and receive primitives are replaced by s-send and s-receive primitives, respectively. We say here that a process TO-Delivers a message m exactly when it completes storing a *TO_delivered* value which contains m (Figure 11, line 9). The recovery procedure at p_i (1) re-initializes the process, (2) retrieves all variables, and (3) sends an UPDATE message to all processes containing *nextBatch* of p_i. (If p_i is lagging behind, then on receiving this UPDATE message, other processes will send the decided batches which p_i has missed.)

Consider a *stable* period where (a) the processes elect the same leader, (b) a majority of the processes are up, (c) no process crashes or recovers and (d) links do not lose messages. In such a period, a process can TO-Deliver a message after three stable storage logs and five communication steps if the leader is the TO-Broadcasting process: the read and the write phase at the leader each requires two communication steps and a stable storage log, one communication step is required to send the decided message set to all processes, and one stable storage log is done while TO-delivering the message. We discuss optimized variants of Paxos in a companion paper [3].

7 Concluding Remarks

The contribution of this paper is a faithful deconstruction of the Paxos replication algorithm into lower level abstractions. Our deconstruction is faithful in the sense that it preserves the efficiency and the resilience of the original Paxos algorithm. The style of the deconstruction promotes both the correctness reasoning and the practical implementation of the algorithm, in a modular manner. It also makes it easy to reconstruct variants of the algorithms that are customised for specific environments. In a companion paper [3], we discuss such reconstructions and show how, by composing our abstractions or re-implementing them differently, we obtain even more resilient or more efficient variants of Paxos that are adapted to specific settings of the distributed environment. Similar approaches were used elsewhere to obtain variants of Paxos that deal with non-deterministic replicas [8] or to deal with an infinite number of clients with shared memory [7].

As we discussed in the introduction, Paxos was viewed in [16, 17, 20] as a sequence of consensus instances. Compared to the original Paxos algorithm, additional messages and stable storage logs are required when relying on a consensus box. This is in particular because the very nature of consensus requires every process to start consensus (which adds messages compared to Paxos), and in a crash-recovery model, every process needs to store its proposal value in the stable storage. Furthermore, consensus typically relies on an underlying notion of leader and, when this notion is hidden within the consensus box, there is no way to know who the current leader is and to use that information to reduce the number of messages in stable periods of the system. Considering a finer-grained register abstraction, namely $\diamond$Register, separate from a leader election procedure, $\diamond$Leader, is the key to our faithful deconstruction of Paxos.

$\diamond$Register is close to the *k-converge* primitive (with $k = 1$) [23]. It is also very similar to the "weak" consensus abstraction identified in [17] with one fundamental difference however. "Weak" consensus does not provide any liveness property. As stated in [17], the reason for not having any liveness property is to avoid the applicability of the impossibility result of [10]. Our $\diamond$Register specification is weaker than consensus and does not fall into the impossibility result of [10], but

1: For each process p_i:
2: **procedure** initialization:
3: ld $\leftarrow$ new $\diamond$Leader; $Received \leftarrow \emptyset$; $\forall l$, $TO\_delivered[l] \leftarrow \emptyset$; $\forall l$, $AwaitingToBeDelivered[l] \leftarrow \emptyset$
4: $TO\_undelivered \leftarrow \emptyset$; $K \leftarrow 1$; $nextBatch \leftarrow 1$ **start task**$\{launch\}$
5: **procedure** TO-Broadcast(m)
6: $Received \leftarrow Received \cup m$
7: **procedure** deliver($msgSet$)
8: $TO\_delivered[nextBatch] \leftarrow msgSet$ - $TO\_delivered$;
9: atomically TO-Deliver each message in $TO\_delivered[nextBatch]$ in some deterministic order;
store$\{TO\_delivered, nextBatch\}$; stop retransmission module $\forall$ messages of nextBatch except DECIDE or UPDATE $\{$***modified from Figure 7***$\}$
10: $nextBatch \leftarrow nextBatch + 1$
11: **while** $AwaitingToBeDelivered[nextBatch] \neq \emptyset$ **do**
12: $TO\_delivered[nextBatch] \leftarrow AwaitingToBeDelivered[nextBatch]$- $TO\_delivered$
13: atomically deliver $TO_delivered[nextBatch]$in some deterministic order;
store$\{TO\_delivered, nextBatch\}$; stop retransmission module $\forall$ messages of nextBatch except DECIDE or UPDATE $\{$***modified from Figure 7***$\}$
14: $nextBatch \leftarrow nextBatch+1$
15: **task** launch
16: **upon** $Received$ - $TO\_delivered \neq \emptyset$ **or** leader has changed **do** $\{$*Upon case executed only once per received message. If upon triggered by a leader change, jump to line 27*$\}$
17: **while** $AwaitingToBeDelivered[K+1] \neq \emptyset$ **or** $TO\_delivered[K+1] \neq \emptyset$ **do**
18: $K \leftarrow K+1$
19: **if** $K = nextBatch$ **and** $AwaitingToBeDelivered[K] \neq \emptyset$ **and** $TO\_delivered[K] = \emptyset$ **then**
20: deliver($AwaitingToBeDelivered[K]$)
21: $TO\_undelivered \leftarrow Received - TO\_delivered$
22: **if** leader() = p_i **then**
23: **while** *Converge*$(K, *)$ **is active do**
24: $K \leftarrow K+1$
25: **start task** Converge($K, TO\_undelivered$)
26: **else**
27: s-send($TO\_undelivered$) to ld.leader()
28: **task** Converge($L, msgSet$) $\{$*Keep on trying until some value is decided*$\}$
29: $returnedMsgSet \leftarrow abort$
30: register$_L$ $\leftarrow$ new $\diamond$Register()
31: **while** $returnedMsgSet = abort$ **do**
32: **if** ld.leader() $= p_i$ **then**
33: $returnedMsgSet \leftarrow$ register$_L$.propose($msgSet$)
34: s-send(DECISION,$L, returnedMsgSet$) to all processes
35: **upon s-receive** m from p_j **do**
36: **if** $m =$ (DECISION,$K_{p_j}, msgSet^{K_{p_j}}$) **or** $m =$ (UPDATE,$K_{p_j}, TO\_delivered[K_{p_j}]$) **then**
37: **if task** Converge($K_{p_j}, *$) **is active then stop task** Converge($K_{p_j}, *$)
38: **if** $K_{p_j} \neq nextBatch$ **then** $\{$*p_j is ahead or behind*$\}$
39: **if** $K_{p_j} < nextBatch$ **then** $\{$*p_j is behind*$\}$
40: **for all** L such that $K_{p_j} < L < nextBatch$: s-send(UPDATE,$L, TO\_delivered[L]$) to p_j $\{$*If $p_j \neq p_i$*$\}$
41: **else**
42: $AwaitingToBeDelivered[K_{p_j}] = msgSet^{K_{p_j}}$; s-send(UPDATE,$nextBatch$-$1, TO\_delivered[nextBatch$-$1]$) to p_j $\{$*If $p_j \neq p_i$*$\}$
43: **else**
44: deliver($msgSet^{K_{p_j}}$)
45: **else**
46: $Received \leftarrow Received \cup$ set of messages contained in m
47: **upon** recovery **do** $\{$***Added procedure to Figure 7***$\}$
48: initialization
49: **retrieve**$\{TO\_delivered, nextBatch\}$; $K \leftarrow nextBatch$; $nextBatch \leftarrow nextBatch+1$; $Received \leftarrow TO\_delivered$
50: s-send(UPDATE,$nextBatch$-$1, TO\_delivered[nextBatch$-$1]$) to all

Figure 11: The Paxos deconstruction

nevertheless features a meaningful liveness property. The liveness property of our $\Diamond$Register coupled with $\Diamond$Leader is precisely what allows us to ensure progress at the level of the replication algorithm.

The round-based register introduced in [2] (and refined in [7], where it was given a precise specification) has the same power as our $\Diamond$Register: they can both be implemented in an asynchronous system when a majority of processes are correct. However, unlike round-based register, $\Diamond$Register does not involve rounds in its specification, and thus has simpler properties. The round-based computation is hidden inside our implementation of the $\Diamond$Register which is indeed close to the implementation of the round-based register [2]. However, the simpler properties of $\Diamond$Register can also be satisfied by an inefficient implementation. In particular, the termination property of $\Diamond$Register does not preclude a correct process p_i from aborting the *propose*() invocation an arbitrarily large number of times, even if p_i is the only process invoking *propose*() on the register. This is prevented in the specifications of [7].

Identifying the notion of $\Diamond$Leader in a precise way enables us to put Paxos and the agreement algorithms of [6, 1] on the same ground; in short, they rely on equivalent failure detectors. Identifying the actual register that would lead to a faithful deconstruction of those agreement algorithms would be an interesting exercise. Intuitively, some notion of locking is used in those algorithms as well, but it is not clear how that could be easily separated from the failure detection procedure and captured within some form of register. In general, it would be interesting to come up with a family of register abstractions that capture the various ways of locking decision values in indulgent agreement algorithms [9, 19, 4]. These algorithms all intuitively share the same flavour but it is not clear how to compare them in a rigorous way.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, May 2000.

[2] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. DSC Technical Report 2001-06, Department of Communication Systems, Swiss Federal Institute of Technology, Lausanne, January 2001.

[3] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Reconstructing Paxos. *Distributed Computing Column of ACM SIGACT News, to appear*, 2003.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI-3)*, pages 173–186, New Orleans, Louisiana, February 1999.

[5] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[6] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

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

[8] P. Dutta, S. Frolund, R. Guerraoui, and B. Pochon. An efficient universal construction for message-passing systems. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC-16)*, Toulouse, France, October 2002.

[9] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[11] R. Guerraoui. Indulgent algorithms. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC-19)*, pages 289–298, Portland, OR, July 2000.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[13] 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.

[14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[15] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems, 16(2):133-169, May 1998.

[16] L. Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):34–58, December 2001.

[17] B. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG-10)*, pages 1–15, Bologna, Italy, 1996.

[18] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[19] B. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing (PODC-7)*, pages 8–17, Toronto, Ontario, Canada, August 1988.

[20] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the Paxos algorithm. *Theoretical Computer Science*, 243(1-2):35–91, July 2000.

[21] L. Sabel and K. Marzullo. Election vs. consensus in asynchronous systems. TR 95-1488, Computer Science Department, Cornell University, Ithaca, New York, February 1995.

[22] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

[23] J. Yang, G. Neiger, and E. Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, Puerto Vallarta, Mexico, July 1998.