Deconstructing Paxos*†

Romain Boichat+ Partha Dutta+ Svend Frølund* Rachid Guerraoui+

+ Swiss Federal Institute of Technology, CH-1015 Lausanne

* Hewlett-Packard Laboratories, 1501 Page Mill Rd, Palo Alto

Abstract

The Paxos part-time parliament protocol of Lamport provides a very practical way to implement a fault-tolerant deterministic service by replicating it over a distributed message passing system. The contribution of this paper is a faithful deconstruction of Paxos that preserves its efficiency in terms of forced logs, messages and communication steps. The key to our faithful deconstruction is the factorisation of the fundamental algorithmic principles of Paxos within two abstractions: weak leader election and round-based consensus, itself based on a round-based register abstraction. Using those abstractions, we show how to reconstruct, in a modular manner, known and new variants of Paxos. In particular, we show how to (1) alleviate the need for forced logs if some processes remain up for sufficiently long, (2) augment the resilience of the algorithm against unstable processes, (3) enable single process decision with shared commodity disks, and (4) reduce the number of communication steps during stable periods of the system.

Keywords: Distributed systems, fault-tolerance, replication, Paxos, modularisation, abstraction.

Contact author: Romain Boichat. E-mail: Romain.Boichat@epfl.ch, Tel (Fax): +41 21 693 6702 (7570).

---

*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 [11]. 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 explored the use of powerful tools to reason about the correctness of the parliament protocol [12, 16], 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 which enabled them easily adapt the protocol to specific functioning modes throughout the seasons. In particular, during winter, the parliament was heated and some legislators did never leave the chamber: their guaranteed presence helped alleviate the need for expensive writing of decrees on ledgers. This was easy to obtain precisely because the subprotocol used to "store and lock" decrees was precisely codified. In spring, and with the blooming days coming, some legislators could not stop leaving and entering the parliament and their indiscipline prevented progress in the protocol. However, because the election subprotocol used to choose the parliament president was factored out and precisely codified, the protocol could easily be adapted to cope with indisciplined legislators. During summer, very few legislators were in the parliament and it was hardly possible to pass any decree because of the lack of the necessary majority. Fortunately, it was easy to modify the subprotocol used to store and lock decrees and devise a powerful technique where a single legislator could pass decrees by directly accessing the ledgers of other legislators. Fall was a protest period and citizens wanted a faster procedure to pass decrees. Paxons noticed that, in most periods, messengers did not loose messages and legislators replied in time. They could devise a variant of the protocol that reduced the number of communication steps needed to pass decrees during those periods. This powerful optimisation was obtained through a simple refinement of the subprotocol used to propose new decrees.

†This work was partially supported by the Swiss National Fund grant No. 510 207.

1 Introduction

The Paxos Algorithm

The Paxos part-time parliament algorithm of Lamport [11] provides a very powerful 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*) [19]. Each correct replica computes every request and returns the result to the corresponding client which selects the first returned result. Paxos maintains 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. During stable periods, Paxos rapidly achieves progress.[1] As pointed out in [12, 16] however, Paxos is rather tricky and it is difficult to factor out the abstractions that comprise the algorithm. Deconstructing the algorithm and identifying those abstractions is an appealing objective towards specific reconstructions and practical implementations of it.

In [12, 16], Lampson, De Prisco and Lynch focused on the key issue in the Paxos algorithm used to agree on a total order for delivering client requests to the replicas. This agreement aspect, factored out within a consensus abstraction, is deconstructed into a storage and a register part. As pointed out in [12, 16], one can indeed obtain a pedagogically appealing state machine replication algorithm as a straightforward sequence of consensus instances, but faithfully preserving the efficiency of the original Paxos algorithm goes through opening the consensus box and combining some of its underlying algorithmic principles with non-trivial techniques such as log piggy-backing and leasing. The aim of our paper is to describe a faithful deconstruction top to bottom, of the entire Paxos replication algorithm. Our deconstruction is faithful in the sense that it relies on abstractions that do no need to be opened in order to preserve the efficiency of the original Paxos replication scheme.

The Faithful Deconstruction

A key to our faithful deconstruction is the identification of the new notion of *round-based consensus*, which is in a sense, finer-grained than consensus.[2] This new abstraction is precisely what allows us to preserve efficiency without sacrificing modularity. Our deconstruction of the *overall* Paxos state machine replication algorithm is modular, and yet it preserves the efficiency of the original algorithm in terms of forced logs, messages and communication steps. We use round-based consensus in conjunction with a leader election abstraction, both as first class citizens at the level of the replication algorithm. Round-based consensus allows us to expose the notion of round up to the replication scheme, as in the original Paxos replication algorithm (but in a more modular manner) and merge all forced logs of the round at the lowest level of abstraction. Round-based consensus also allows a process to propose more than once (e.g., after a crash and a recovery) without implying a forced log. Having the notion of leader as a first class abstraction at

[1] In fact, the liveness of the algorithm relies on partial synchrony assumptions whereas safety does not: Paxos is "indulgent" in the sense of [6]. In a *stable* period where the leader communicates in a timely manner with a majority of the processes (most frequent periods in practice), two communication steps (four if the client process is not leader) and one forced log at a majority of the processes are enough to perform a request and return a reply.

[2] The *round-based consensus* is actually strictly weaker than consensus: it can be implemented with a majority of correct processes and does not fall within the FLP impossibility, yet it has a meaningful liveness property. Roughly speaking, round-based consensus is the abstraction that we obtain after extracting the leader election from consensus.

the level of the replication algorithm (and not hidden by a consensus box) enables the client to send its request directly to the leader, which can process several requests in a row.

Effective Reconstructions

Not only do our abstractions of leader election and round-based consensus help faithfully deconstruct the original Paxos replication algorithm, they also enable us to straightforwardly reconstruct known and new variants of it by only modifying the implementation of one of our abstractions. For example, we show how to easily obtain a modularisation of the so-called Disk Paxos replication algorithm [5], where progress is ensured with a single correct process and a majority of correct disks, by simply modifying a component in round-based consensus (its round-based register).[3] We also show how to cleanly obtain the "Fast" Paxos variant by integrating the "lease-based" tricky optimisation, sketched in [11] and pointed out in [12]. This optimisation makes it possible in stable periods of the system (where "enough" processes communicate in a timely manner) for any leader to determine the order for a request in a single round-trip communication step.

We also construct two new variants of Paxos. The first one is more resilient than the original one in the sense that it copes with unstable processes, i.e., processes that keep on crashing and recovering forever. (The original Paxos replication algorithm might not achieve progress in the presence of such processes.) Our second variant alleviates the need for stable storage and relies instead on some processes being always up. This variant is more efficient than the original one (stable storage is usually considered a major source of overhead) and intuitively reflects the practical assumption that only part of the total system can be down at any point in time, or indirectly, that the system configuration has a "large" number of replicas.[4] We point out that further variants can be obtained by mixing the variants we present in the paper, e.g., a Fast Disk Paxos

algorithm or a Fast Paxos algorithm than handles unstable processes.

Thanks to our modular approach, we could implement Paxos and its variants as a framework. We give here practical implementation measures of the various replication algorithms in this framework.

Roadmap

The rest of the paper is organised as follows. Section 2 describes the model and the problem specification. Section 3 gives the specification of our abstractions. We show how to implement these specifications in a crash-stop model in Section 4, and how to transpose the implementation in a more general crash–recovery model in Section 5. Section 6 describes four interesting variants of the algorithm. Section 7 discusses related work. Appendix A gives some performance measurements of our framework implementation. Appendix B gives an implementation of the failure detector Ω in a crash-recovery model with partial asynchrony assumptions.

[3] This typically makes sense if we have shared hard disks (some parallel database systems use this approach for fail-over when they mount each others disks) or if we have some notion of network-attached storage.

[4] Note that such a configuration does not preclude the possibility of process crash-recovery. There is here a trade-off that reflects the real-world setting: fewer processes + forced logs vs more processes without forced logs.

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). Note that we do not make here any assumption on the relative speed of processes. 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 *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.[5] 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 and synchronise 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., a message cannot have the same id even after the crash and recovery of a process. Channels can lose or drop 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 at time t, then p_i sent m to p_j before time t.

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.

These properties characterise the links between processes and are independent of the process failure pattern occurring in the execution. The last property is sometimes called *weak loss*, e.g., in [14]. It reflects the usefulness of the communication channel. Without the weak loss property, any interesting distributed problem would be trivially impossible to solve. By introducing the notion of correct process into the *fair loss* property, we define the conditions under which a message is delivered to its recipient process. Indeed, the delivery of a message requires the recipient process to be running at the time the channel attempts to deliver it, and therefore depends on the failure pattern occurring in the execution. The *fair loss* property indicates that a message can be lost, either because the channel may not attempt to

[5] In practice, a process is required to stay up long enough for the computation to terminate. In asynchronous systems however, characterising the notion of "long enough" is impossible.

deliver the message or because the recipient process may be down when the channel attempts to deliver the message to it. In both cases, the channel is said to commit an *omission failure*.

We assume the presence of a discrete global clock whose range ticks $\mathcal{T}$ 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 total order are impossible [4]), but these assumptions will be encapsulated inside our *weak leader election* abstraction and used only to ensure progress (liveness). We give the implementation (with some details on the partial synchrony model) of the failure detector on which is based our weak leader election in Appendix B. Finally, we define a *stable* period when (i) the weak leader election returns the same process p_l at all processes, (ii) there is a majority of processes that remains up, and (iii) no process or link crashes or recovers. Otherwise, we say that the system is in an *unstable* period.

3 Abstractions: Specifications

Our deconstruction of Paxos is based on two main abstractions: a *weak leader election* and a *round-based consensus*, itself based on a *round-based register* (sub)abstraction. These "shared memory" abstractions export operations that are invoked by the processes implementing the replicated service. As in [10], we say that an operation invocation inv_2 *follows* (is *subsequent* to) an operation invocation inv_1, if inv_2 was called after inv_1 has returned. Otherwise, the invocations are *concurrent*.

Roughly speaking, Paxos ensures that all processes deliver messages in the same order. The round-based consensus encapsulates the subprotocol used to "agree" on the order; the round-based register encapsulates the subprotocol used (within round-based consensus) to "store" and "lock" the agreement value (i.e., the order); and the weak leader election encapsulates the subprotocol used to eventually choose a unique leader that succeeds in storing and locking a final decision value in the register. We give here the specifications of these abstractions, together with the specification of the problem we solve using these abstractions, i.e., total order delivery. (Implementations are given in the next sections.) The specifications rely on the notion of process correctness: we assume that processes fail only by crashing, and a process is correct if there is a time after which the process is always-up (i.e., not crashed).[6]

3.1 Round-Based Register

Like a standard register, a *round-based register* is a shared register that has two operations: $read(k)$ and $write(k, v)$. These operations are invoked by the processes in the system. Unlike a standard register, the operation invocations of a round-based register (1) take as a parameter an integer k (i.e., a round number), and (2) may commit or abort. Note that the notion of round is the same for round-based register and round-based consensus: it corresponds to the notion of ballots in the original Paxos. The commit/abort outcome reflects the success or the failure of the operation. More precisely, the $read(k)$ operation takes as input an integer k. It returns a pair $(status, v)$ where $status \in \{commit, abort\}$ and $v \in V$ represents the set of possible values for the register; $\perp \in V$ is the initial value of the register. If $read(k)$ returns $(commit, v)$ (resp. $(abort, v)$), we say that $read(k)$ *commits* (resp. *aborts*) with v. The $write(k, v)$ oper-

[6] Note that the validity period of this definition is the duration of a protocol execution, i.e., in practice, a process is correct if it eventually remains up long enough for the protocol to terminate.

ation takes as input an integer k and a value $v \in V$. It returns $status \in \{commit, abort\}$. If $write(k, v)$ returns $commit$ (resp. $abort$), we say that $write(k, v)$ *commits* (resp. *aborts*).[7] A round-based register satisfies the following properties:

- **Read-write-abort:** If $read(k)$ or $write(k, *)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' > k$.
- **Read-write-commit:** If $read(k)$ or $write(k, *)$ commits, then no subsequent $read(k')$ or $write(k', *)$ can commit with $k' < k$.[8]
- **Read-commit:** If $read(k)$ = $(commit, v)$ and $v \neq \perp$, then some operation $write(k', v)$ was invoked with $k' \leq k$.
- **Write-commit:** If $write(k, v)$ commits and no subsequent $write(k', v')$ is invoked with $k' \geq k$ and $v' \neq v$, then any $read(k'')$ that commits, commits with v if $k'' > k$.

These properties define the conditions under which the operations *can* abort or commit. Indirectly, these conditions relate the values read and written on the register. We first describe the condition under which an invocation *can* abort. Roughly speaking, an operation invocation aborts only if there is a *conflicting* invocation. Like in [11], the notion of "conflict" is defined here in terms of round numbers associated with the operations. Intuitively, a $read()$ that commits returns the value written by a "previous" $write()$, or the initial value $\perp$ if no $write()$ has been invoked. A $write()$ that commits forces a subsequent $read()$ to return the value written, unless this value has been overwritten.
The read-write-abort condition captures the intuition that a $read(k)$ (resp. a $write(k, v)$) conflicts with any other operation ($read(k')$ or $write(k', v)$) made with $k' > k$. The read-write commit condition expresses the fact that, to commit an operation, a process must use a round number that is higher than any round number of an already committed invocation. The read-commit condition captures the intuition that no value can be read unless it has been "previously" written. If there has not been any such write, then the initial value $\perp$ is returned. The write-commit condition captures the intuition that, if a value is (successfully) written, then, unless there is a subsequent write, every subsequent successfully read must return that value. Informally, the two conditions (read-commit, write-commit) ensure that the value read is the "last" value written.

To illustrate the behaviour of a round-based register, consider the example of Figure 1. Three processes p_1, p_2 and p_3 access the same round-based register. Process p_1 invokes $write(1, X)$ before any process invokes any operation on the register: operation $write(1, X)$ commits and the value of the register is X: p_1 gets $commit$ as a return value. Later, p_2 invokes $read(2)$ on the register: the operation commits and p_2 gets $(commit, X)$ as a return value. If p_3 later invokes $write(1, Y)$, then the operation aborts: the return value is $abort$ (because p_2 has invoked $read(2)$). The register value remains X. If p_3 later invokes $write(3, Y)$, the operation commits: the new register value is then Y.

[7]Note that even if a $write()$ aborts, its value might be subsequently read, i.e., the $write()$ operation is not *atomic*.

[8]Note that we deliberately do not restrict the case where different processes perform invocations with the same round number. Paxos indeed assumes round number uniqueness as we will see in Section 4.

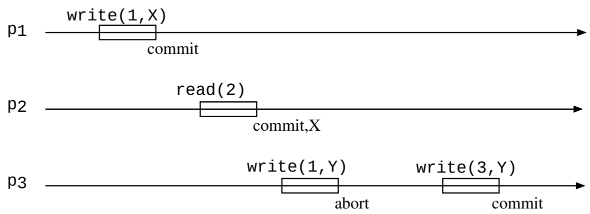


Figure 1. Round-based register example

3.2 Round-Based Consensus

We introduce below our round-based consensus abstraction. This abstraction captures the subprotocol used in Paxos to agree on a total order. Our consensus notion corresponds to a single instance of total order, i.e., one batch of messages. To differentiate between consensus instances, i.e., batch of messages, we index the consensus instances with an integer (L). We represent our consensus notion in the form of a shared object with one operation: $propose(k, v)$ [9]. This operation takes as input an integer k (i.e., a round number which is the same one used in the round-based register) and an initial value v in a domain V (i.e., a proposition for the consensus). It returns a $status$ in $\{commit, abort\}$ and a value in V. We say that a process p_i *proposes* a value $init_i$ for round k when p_i invokes function $propose(k, init_i)$. We say that p_i *decides v* in round k (or commits round k) when p_i returns from the function $propose(k, init_i)$ with $commit$ and v. If the invocation of $propose(k, v)$ returns $abort$ at p_i, we say that p_i aborts round k. Round-based consensus has the following properties:

- **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 aborts round k, then some process has proposed (some value) in a round $k' > k$; if a process p_i commits round k, then no process can subsequently commit round $k' < k$.

The agreement and validity properties of our round-based consensus abstraction are similar to those of the traditional consensus abstraction [9]. Our termination property is however strictly weaker. If processes keep concurrently proposing values with increasing round numbers, then no process might be able to decide any value. In a sense, our notion of consensus has a conditional termination property. In comparison to [12], the author presents a consensus that does not ensure any liveness property. As stated by Lampson, the reason for not giving any liveness property is to avoid the applicability of the impossibility result of [4]. Our round-based consensus specification is weaker than consensus and does not fall into the impossibility result of [4], but nevertheless includes a liveness property. In the rest of the paper, when no ambiguity is possible, we shall simply use the term consensus instead of round-based consensus.

In Figure 2, process p_2 commits consensus with value Y for round 2. Process p_1 then triggers consensus by invoking $propose(1, X)$ but aborts because process p_2 proposed with a higher round number and prevents p_1 from committing. Process p_1 then proposes with value X for round 4, and this time p_1 commits. Process p_3 aborts when it proposes with value Z for round 3.

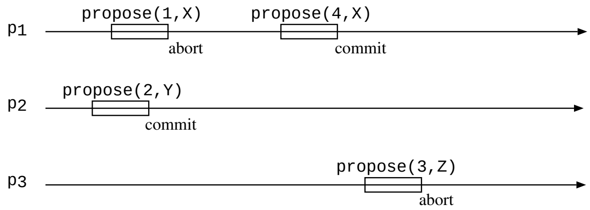


Figure 2. Round-based consensus example

3.3 Weak Leader Election

Intuitively, a *weak leader election* abstraction is a shared object that elects a leader among a set of processes. It encapsulates the subprotocol used in Paxos to choose a process that decides on the ordering of messages. The weak leader election object has one operation, named $leader()$, which returns a process identifier, denoting the current leader. When the operation returns p_j at time t and 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). We say that a process p_i is an *eventual perpetual leader* if (1) p_i is correct, and (2) eventually every invocation of $leader()$ returns p_i. Weak leader election satisfies the following property: *Some process is an eventual perpetual leader.*

It is important to notice that the property above does not prevent the case where, for an arbitrary period of time, various processes are simultaneously leaders.[9] However, there must be a time after which the processes agree on some unique correct leader. Figure 3 depicts a scenario where every process elects process p_1, and then p_1 crashes; eventually every process elects then process p_2.

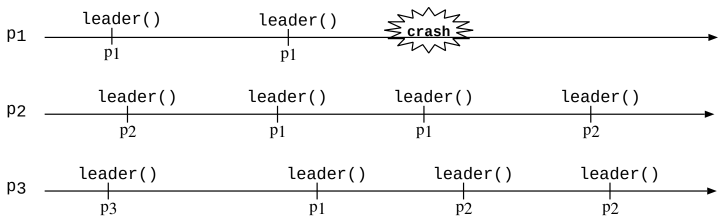


Figure 3. Weak leader election example

3.4 Total Order Delivery

The main problem solved by the actual Paxos protocol is to ensure total order delivery of messages (i.e., requests broadcast to replicas).[10] Total order broadcast is 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. Our total order broadcast protocol has the following properties:

- **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.

[9] In this sense our weak leader election specification is strictly weaker then the notion of leader election introduced in [18].

[10] In fact, Paxos also deals with causal order delivery of messages, but we do not consider that issue here.

- **Agreement:** If a process TO-Delivers a message m, then every correct process eventually TO-Delivers m.
- **Validity:** For any message m, (i) every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process, and (ii) 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.

It is important to notice that the total order property we consider here is slightly stronger from the one introduced in [8]. In [8], it is stated that if any 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 $m_1; m_2; m_3$ 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 [11].

4 Abstractions: Implementations

In the following, we give wait-free [9] implementations of our three abstractions and show how they can be used to implement a simple variant of the Paxos protocol in the particular case of a crash-stop model (following the architecture of Figure 4). We will show how to step to a crash-recovery model in the next section.

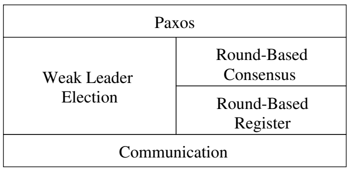


Figure 4. Architecture

We simply assume here that messages are not lost or duplicated 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 weak leader election abstraction, we assume the failure detector Ω introduced in [2].

4.1 Round-Based Register

The algorithm of Figure 5 implements the abstraction of a round-based register. The algorithm works intuitively as follows. Every process p_i has a copy of the register value, denoted by v_i, and initialised to $\perp$. A process reads or writes a value by accessing a majority of the copies with a round number. According to the actual round number, a process p_i might "accept" or not the access to its local copy v_i. Every process p_i has a variable $read_i$ that represents the highest round number of a $read()$ "accepted" by p_i, and a variable $write_i$ that represents the highest round number of a $write()$ "accepted" by p_i. The algorithm is made up of two procedures ($read()$ and $write()$) and two tasks that handle READ and WRITE messages. Each task is executed in one atomic step to avoid mutual exclusion problems for

```
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}
   v_i ← ⊥                                                   {p_i's estimate of the register value}
procedure read(k)
   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, v)                                       {read() is aborted}
   else
      select the [ackREAD,k, k′, v] with the highest k′
      return(commit, v)                                      {read() is committed}
procedure write(k, 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)                                          {write() is aborted}
   else
      return(commit)                                         {write() is committed}
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, v_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
      v_i ← v                                                {A new value is "adopted"}
      send [ackWRITE,k] to p_j
```

Figure 5. A wait-free round-based register in a crash-stop model

the common variables. We assume here that a task is implemented as a thread in Java™.

Lemma 1. *Read-write-abort: If $read(k)$ or $write(k, *)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' > k$.*

Proof. Assume that some process p_j invokes a $read(k)$ (resp. $write(k, *)$) that returns $abort$ (i.e., aborts). By the algorithm of Figure 5, this can only happen if some process p_i has a value $read_i > k$ or $write_i > k$, which means that some process has invoked $read(k')$ or $write(k')$ with $k' > k$. □

Lemma 2. *Read-write-commit: If $read(k)$ or $write(k, *)$ commits, then no subsequent $read(k')$ or $write(k', *)$ can commit with $k' < k$.*

Proof. Let process p_i be any process that commits $read(k)$ (resp. $write(k, *)$). This means that a majority of the processes have "accepted" $read(k)$ (resp. $write(k, *)$). For a process p_j to commit $read(k')$ or $write(k', *)$ with $k' < k$, a majority of the processes must "accept" $read(k')$ (resp. $write(k', *)$). Hence, at least one process must "accept" $read(k)$ (resp. $write(k, *)$) and then $read(k')$ (resp. $write(k', *)$) with $k' < k$ which is impossible by the algorithm of Figure 5: a contradiction. □

Lemma 3. *Read-commit: If $read(k)$ = $(commit, v)$ and $v \neq \perp$, then some operation $write(k', v)$ was invoked with $k' \leq k$.*

Proof. By the algorithm of Figure 5, if some process p_j commits $read(k)$ with $v \neq \perp$, then some process p_i must have sent to p_j a message [ackREAD,$k, write_j, v$] then some process p_m must have invoked $write(k', v)$ with $k' \leq k$, otherwise p_i would have sent [nackREAD,k] to p_j or [ackREAD,$k, 0, \perp$] $\square$

Lemma 4. *Write-commit: If $write(k, v)$ commits and no subsequent $write(k', v')$ is invoked with $k' \geq k$ and $v' \neq v$, then any $read(k'')$ that commits, commits with v if $k'' > k$.*

Proof. Assume that some process p_i commits $write(k, v)$, and assume that no subsequent $write(k', v')$ has been invoked with $k' \geq k$ and $v' \neq v$, and that for some $k'' > k$ some process p_j commits $read(k'')$ with v'. Assume by contradiction that $v \neq v'$. Since $read(k'')$ commits with v', by the read-commit property, some $write(k'', v')$ was invoked before or at the same round k''. However, this is impossible since we assumed that no $write(k', v')$ operation with $k' \geq k$ and $v' \neq v$ has been invoked, i.e., v_i remains unchanged to v: a contradiction. $\square$

Proposition 5. *The algorithm of Figure 5 implements a round-based register.*

Proof. Directly from lemmata 1, 2, 3 and 4. $\square$

Proposition 6. *With a majority of correct processes, the implementation of Figure 5 is wait-free.*

Proof. The only wait statements of the protocol are the guard lines that depicts the waiting for a majority of replies. These are non-blocking since we assume a majority of correct processes. Indeed, a majority of correct processes always send a message to the requesting process either of type [ackREAD, nackREAD], or of type [ackWRITE, nack-WRITE]. $\square$

4.2 Round-Based Consensus

The algorithm of Figure 6 implements a round-based consensus object that relies on a wait-free round-based register. As in [11, 16], we assume round number uniqueness, i.e., a process p_i proposes only for round k=i, k=i+n, k=i+2n, ... The basic idea of the algorithm is the following. For a process p_i to propose a value for a round k, p_i first reads the value of the register with k, and if the $read(k)$ operation commits, p_i invokes a $write(k, v)$ (or p_i's initial value instead of v if no value has been written). If the $write(k, v)$ operation commits, then the process decides the value written (i.e., returns this value). Otherwise, p_i aborts and returns $abort$ (line 7).

```
1: procedure consensus()                                   {Constructor, for each process p_i}
2:    v ← ⊥; reg ← new register()
3: procedure propose(k, init_i)
4:    if reg.read(k) = (commit, v) then
5:       if (v =⊥) then v ← init_i
6:       if (reg.write(k, v) = commit) then return(commit, v)
7:    return(abort, init_i)
```

Figure 6. A wait-free round-based consensus using a wait-free round-based register

Lemma 7. *Validity: If a process decides a value v, then v was proposed by some process.*

Proof. Let p_i be a process that decides some value v. By the algorithm of Figure 6, either (a) v is the value proposed by p_i, in which case validity is satisfied, or (b) v has been read by p_i in the register. Consider case (b), by the read-commit property of the register, some process p_j must have invoked some $write()$ operation. Let p_j be the the first process that invokes $write(k_0, *)$ with k_0 equal to the smallest k ever invoked for $write(k, v)$. By the algorithm of Figure 6, there are two cases to consider: either (a) v is the value proposed by p_j, in which case validity is ensured, or (b) v has been read by p_j in the register. For case (b), by the read-commit property of the register, for p_j to read v, some process p_m must have invoked $write(k', v)$ with $k' \leq k_0$: a contradiction. Therefore, v is the value proposed by p_j and validity is ensured. □

Lemma 8. *Agreement: No two processes decide differently.*

Proof. Assume by contradiction that two processes p_i and p_j decide two different values v and v'. Let p_i decides after committing $write(k, v)$ and p_j decides after committing $write(k', v')$. Assume without loss of generality that $k' > k$ (remember that we assume round number uniqueness) and $k' > k'' > k$ is the round number such that the next $write()$ operation is invoked. By the algorithm of Figure 6, p_j must have committed $read(k'')$ before invoking $write(k'', v'')$, and by the write-commit property p_j must have committed $read(k'')$ with $v'' = v$ before invoking $write(k'', v'')$. Therefore, even if $write(k'', v'')$ aborts, p_j tries to write v and not $v'' \neq v$. Therefore, the next time p_j commits $write(k', v')$, then $v' = v$, i.e., decides v: a contradiction. □

Lemma 9. *Termination: If a process p_i aborts in a round k, then some process has proposed (some value) a round $k' > k$; if a process p_i commits a round k, then no process can subsequently commit a round $k' < k$.*

Proof. For the first part, assume that a process p_i aborts a round k. By the algorithm of Figure 6, this means that p_i aborts $read(k)$ or $write(k, *)$. By the read-write-abort property, some process must have proposed in a round $k' > k$. Consider now the second part. Assume that a process p_i commits a round k. By the algorithm of Figure 6 and the read-write-commit property, no process can subsequently commit any $read(k')$ or $write(k', *)$ with $k' < k'$. Hence no process can subsequently commit a round $k' < k$. □

Proposition 10. *The algorithm of Figure 6 implements a wait-free round-based consensus.*

Proof. Termination, agreement and validity follows from lemmata 7, 8 and 9. The implementation of round-based consensus is wait-free since it is based on a wait-free round-based register and does not introduce any "wait" statement. □

4.3 Weak Leader Election

Figure 7 describes a simple implementation of a wait-free weak leader election. The protocol relies on the assumptions (i) that at least one process is correct and (ii) the existence of failure detector Ω [2]: Ω outputs (at each process) 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 exactly one correct process p_l is always trusted by every correct process.*[11] Our weak leader election

[11]It was shown in [2] that Ω is the weakest failure detector to solve consensus and total order broadcast in a crash-stop system model. Failure detector Ω can be implemented in a message passing system with partial synchrony assumptions [3].

relies on Ω in the following way. The output of failure detector Ω at process p_i is denoted by Ω_i. The function simply returns the value of Ω_i.

procedure leader() $\{$*For each process* $p_i\}$
  return(Ω_i)

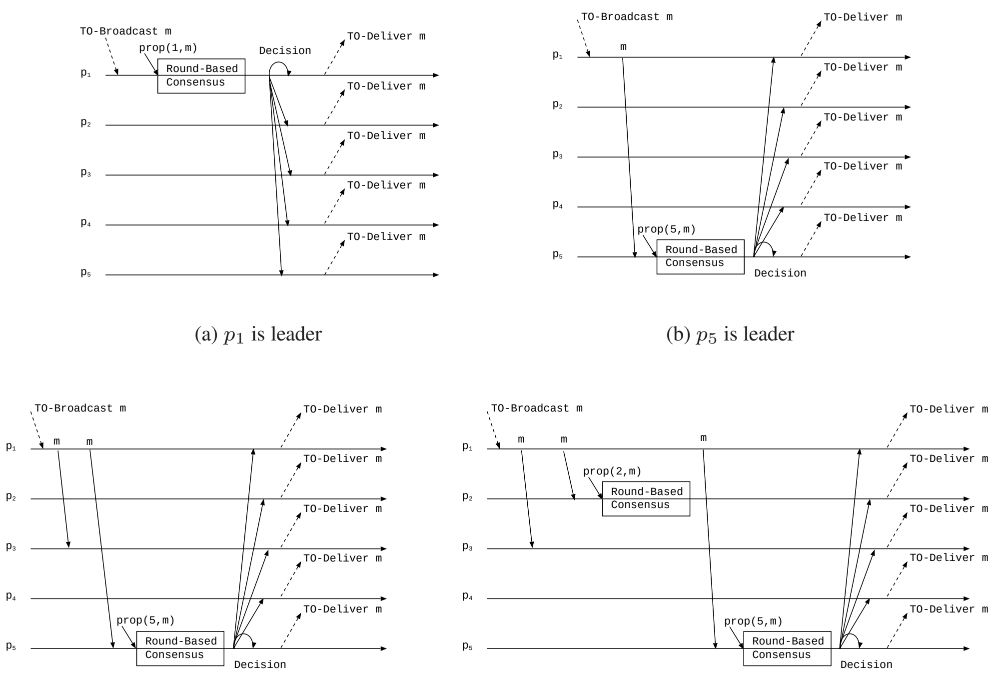
Figure 7. A wait-free weak leader election with Ω

Proposition 11. *With failure detector Ω and the assumption that at least one process is correct, the algorithm of Figure 7 implements a wait-free weak leader election.*

Proof. Follows from the property of Ω [2]. $\square$

4.4 A Simple Variant of Paxos

The algorithm of Figure 9 can be viewed as a simple and modular version of Paxos in a crash-stop model (whereas the original Paxos protocol considers a crash-recovery model - see next section). The algorithm uses a series of consecutive round-based consensus (or simply consensus) instances: each consensus instance being used to agree on a batch of messages. Every process differentiates consecutive consensus instances by maintaining a local counter (L): each value of the counter corresponds to a specific consensus instance and is indexed to the $propose()$ operation. Consensus instances are triggered according to the output of the weak leader election protocol: only leaders trigger consensus instances.



(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 8. Execution schemes

We give here an intuitive description of the algorithm. When a process p_i TO-Broadcasts a message m, p_i consults

the weak leader election protocol and sends m to leader p_j. When p_j receives m, p_j triggers a new consensus instance by proposing all messages that it received (and not yet TO-Delivered) and set the round number to the process id. Note that in order to decide on a batch of messages, more than one consensus round might be necessary; various invocation consensus for the same batch (L) are differentiated with round number k. Due to round number uniqueness, no process can propose twice for the same round k.[12] In fact, p_j starts a new task *propose* (L^{th}) that keeps on trying to commit consensus for this batch (L), as long as p_j remains leader. If consensus commits, p_j sends the decision to every process. Otherwise, task *propose* periodically invokes consensus with the same batch of messages but increases its round number by n, unless p_j stops being leader or some consensus instance for the same batch commits. When p_i elects another process p_k, p_i sends to p_k every message that p_i received, and not yet TO-Delivered. By the weak leader election property, eventually every correct process elects the eventual perpetual leader p_l, and sends its messages to p_l. By the round-based consensus specification, eventually p_l commits consensus and sends the decision to every process. Once p_i receives a decision for the L^{th} batch of messages, p_i stops task *propose* for this batch. Process p_i TO-Delivers this batch of messages only if it is the next one that was expected, i.e., if p_i has already TO-Delivered messages of batch L-1. If it is not the case, p_i waits for the next expected batch (*nextBatch*) to respect total order. Within a batch of messages, processes TO-Deliver messages using a deterministic ordering function.
Note that an array of round-based registers is used in the total order broadcast protocol: each round-based register corresponds to the "store and lock" of a given consensus instance. Finally, note that a process p_i instantiates a round-based register when (i) p_i instantiates a round-based consensus, or (ii) p_i receives for the first time a message for the L^{th} consensus, i.e., L^{th} register of the array.

Figure 8 depicts four typical execution schemes of the algorithm. We assume for all cases that (i) process p_1 TO-Broadcasts a message m, (ii) process p_5 is the eventual perpetual leader, and (iii) L =1. ($prop(*)$ stands in the figures for $propose(*)$.) In Figure 8(a), p_1 elects itself, triggers a new consensus instance by invoking $propose(1, m)$, commits, and sends the decision to all. In Figure 8(b), p_1 elects p_5 and sends m to p_5. Process p_5 then invokes $propose(5, m)$, commits, then sends the decision to all. In Figure 8(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 commits consensus 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 8(d), p_1 elects p_3 (which does not elect itself), then p_1 elects p_2, which elects itself and invokes $propose(2, m)$ but aborts. Finally, p_1 elects p_5, and, as for case (c), p_5 commits consensus and sends the decision to all.

Precise description. We give here more details about the algorithm of Figure 9. We first describe the main data structure, and then the main parts of the algorithm. Each process p_i maintains a variable *TO_delivered* that contains the messages that were 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

[12] Allowing two processes to propose for the same round could violate agreement. For example, process p_1 invokes $propose(1, v)$ and commits, and process p_2 invokes $propose(1, v')$. The termination property of consensus allows p_2 to commit: agreement would indeed be violated. However, if p_i invokes $propose(1, v)$, crashes and recovers, p_1 can then invoke $propose(1, v)$ or even $propose(1, v')$ without violating the properties of round-based consensus.

been decided but not yet TO-Delivered are put in the set *AwaitingToBeDelivered*. 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 protocol: (a) when a process receives some message, task *launch* starts[13] task *propose* if the process p_i is leader, or if p_i is not leader, sends the messages it did not yet TO-Delivered to the leader; (b) task *propose* keeps on starting round-based consensus while p_i is leader, until a decision is reached; (c) primitive *receive* handles received messages, and stops task *propose* once p_i receives a decision; 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.

- In task *launch*, process p_i triggers the upon case when the set *TO_undelivered* contains new messages or whether p_i elects another leader (line 15). Note that the upon case is executed only once per received message to avoid multiple consensus instances of the exact same batch 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-Delivered. Otherwise, before starting a new consensus instance, p_i first verifies at line 16 if (i) it already received the decision for this batch of messages, or (ii) it already TO-Delivered this batch of messages. Process p_i verifies then if it is a leader, and if so, p_i increments the batch number to initiate a consensus for a new batch of messages (L+1), i.e., p_i starts task *propose* with *TO_undelivered* as the batch of messages and the round number set to the id of p_i. If p_i is not leader, then p_i sends the messages it did not yet TO-Delivered to the leader.

- In task *propose*, a process p_i periodically invokes consensus (proposes) if p_i is leader. By the property of weak leader election, one of the correct processes (p_l) will be the eventual perpetual leader. Once p_l is elected by every correct process, p_l receives all batches of messages from every correct process, proposes and commits consensus (line 31) and then sends the decision to all (line 34). Note that in this primitive, p_i proposes the same batch of messages but with an increasing round number.

- In the primitive *receive*, when process p_i receives the decision of consensus (line 36), p_i first stops task *propose*$_L$: p_i does not stop other batches (task *propose*) - i.e., this could influence the result of some other consensus instances (line 37). Process p_i then verifies that the decision received is the next decision that was expected (*nextBatch*). Otherwise, there are two cases to consider: (i) p_i is ahead, or (ii) p_i is lagging. For case (i), if p_i is ahead (i.e., 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 40). For case (ii), 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 in order for p_j to update itself (p_i) when p_j receives this "on purpose lagging" message. Process p_i waits until it gets the next expected batch in order to satisfy the total order property.

- 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.

[13]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). Moreover, the variable *TO_delivered* means the union of all arrays *TO_delivered*$[L]$.

We assume here a system model where messages keep being broadcast indefinitely. This assumptions is precisely what enables us to ensure the uniformity of agreement without additional forced logs and communication steps.

Lemma 12. *If the eventual perpetual leader proposes a batch of messages, it eventually decides.*

Proof. Assume by contradiction that process p_i is the eventual perpetual leader that proposes a batch of messages and never decides. By the algorithm of Figure 9, p_i keeps incrementing round number k (line 33). Let k_0 be the smallest round number reached by p_i such that no process else than p_i ever invokes any operation. By the algorithm of Figure 9, such round number exists because, unless it is leader, no other process invokes any operation on the consensus. By the termination property of consensus and since the implementation of consensus is wait-free, p_i commits $propose(k_0, *)$, which means that p_i decides a value: a contradiction. □

Lemma 13. *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.*

Proof. Suppose by contradiction that a process p_i TO-Broadcasts a message m but never TO-Delivers m. Remember that every time p_i elects a new process, p_i sends m to this new leader. By the weak leader property, eventually p_i elects the eventual perpetual leader process p_l and p_i sends m to p_l. By lemma 12, p_l proposes, decides and sends the decision to all processes. There are now two cases to consider: (a) p_l does not crash, or (b) p_l crashes. For case (a), by the properties of the channels, p_i receives the decision from p_l and TO-Delivers m: a contradiction. For case (b), if p_l crashes, p_l was not an eventual perpetual leader: a contradiction. □

Lemma 14. *Agreement: If a process TO-Delivers a message m, then every correct process eventually TO-Delivers m.*

Proof. Suppose by contradiction that a process p_i TO-Delivers m and let p_j be any correct process that does not TO-Deliver m. Process p_i must have received the decision from some process p_k (p_k could be p_i). There are two cases to consider: (a) p_k is a correct process, or (b) p_k is a faulty process. For case (a), since p_k TO-Delivered m, by the reliable properties of the channels, every correct process receives the decision and TO-Delivers m: a contradiction. For case (b), since we assume that new messages keep coming, the eventual perpetual leader p_l TO-Delivers m and therefore sends at some time the decision to every correct process: a contradiction. As explained earlier, due to round number uniqueness, no two processes can propose for the same round, therefore every correct process decides the same value for consensus. □

Lemma 15. *Validity: For any message m, (i) every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process, and (ii) every process p_i TO-Delivers m at most once.*

Proof. For the first part (i), suppose by contradiction that some process p_i TO-Delivers a message m that was not TO-Broadcast by any process. For a message m to be TO-Delivered, by the algorithm of Figure 9, m must be decided through round-based consensus. By the validity property of consensus, m has to be proposed (line 24). In order to be proposed, m has to be in the set *TO_undelivered* (line 20); then to be in the set *TO_undelivered*, m has to be in the set *Received* (line 46). Finally, for m to be in set *Received*, m has to be TO-Broadcast or sent (lines 6 & 26). Ultimately, for m to be sent, m must be TO-Broadcast: a contradiction. For the second part (ii), p_i cannot TO-Deliver more than once a message m. This is impossible since line 8 removes all the messages that have been already TO-Delivered. Of course, we assume that p_i distinguishes all messages that appear twice in the variable $msgSet$. □

Lemma 16. *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.*
Proof. Suppose by contradiction that p_i TO-Delivers a message m before a message m' and p_j TO-Delivers m' before m. There are two cases to consider: (a) m and m' are in the same message set, and (b) m and m' are in different message sets. For case (a), since every process delivers messages following the same deterministic order, m is delivered before m' on both processes: a contradiction. For case (b), suppose that m is part of $msgSet^L$ and $m' \in msgSet^{L'}$ where $L < L'$. For m to be TO-Delivered, $msgSet^L$ has to be received as a DECIDE or UPDATE message (line 36). If p_i TO-Delivers m before m', then p_j cannot TO-Deliver m' before m since the predicate of guard line 38 forbids p_j to TO-Deliver batches of messages out of order: a contradiction. Nevertheless, p_j could receive the L'^{th} batch of messages before the L^{th} batch of messages, but the batch would be put in the set *AwaitingtoBeDelivered*. □

Proposition 17. *The algorithm of Figure 9 satisfies the termination, agreement, validity and total order properties.*
Proof. Directly from the lemmata 13, 14, 15 and 16. □

5 A Faithful Deconstruction of Paxos

This section describes a *faithful* and *modular* deconstruction of Paxos [11]. It is *modular* in the sense that it builds upon our abstractions: the specifications of these are not changed, only their implementations are slightly modified. It is *faithful* in the sense that it captures the practical spirit of the original Paxos protocol: it preserves the efficiency of Paxos and tolerates temporary crashes of links and processes. Just like with the original Paxos protocol, we preclude the possibility of *unstable* processes: either processes are correct (eventually always-up), or they eventually crash and never recover. We will come back to this assumption in the next section.

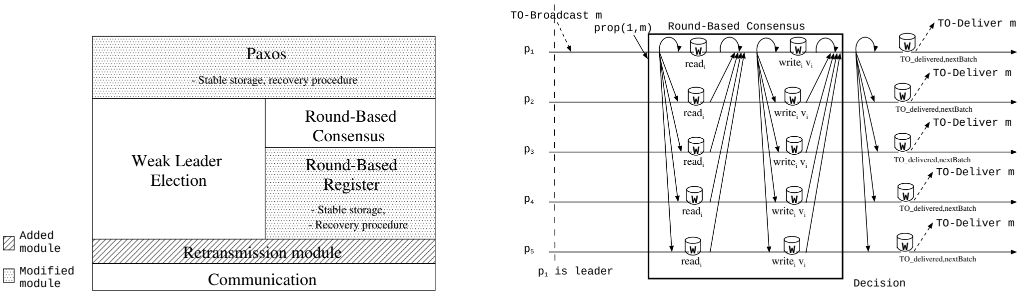
To step from a crash-stop model to a crash-recovery model, we mainly adapt the round-based register and slightly modify the global protocol to deal with recovery (in shade in Figure 10(a), therefore we only present these abstractions in this section). Every process performs some forced logs so that it can consistently retrieve its state when it recovers. To cope with temporary link failures, we build upon a *retransmission* module, associated with two primitives *s-send* and *s-receive*: if a process p_i *s-sends* a message to a correct process p_j and p_i does not crash, the message is eventually *s-received*.

5.1 Retransmission Module

We describe here a retransmission module that encapsulates retransmissions 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 eventually s-receives m.* Figure 11 gives the algorithm of the retransmission module. All messages that need to be retransmitted are put in the

1: For each process p_i:
2: **procedure** initialisation:
3: *Received*[] $\leftarrow \emptyset$; *TO_delivered*[] $\leftarrow \emptyset$; **start task**$\{$*launch*$\}$
4: *TO_undelivered* $\leftarrow \emptyset$; *AwaitingToBeDelivered*[] $\leftarrow \emptyset$; $K \leftarrow 1$; *nextBatch* $\leftarrow 1$
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 deliver all messages in *TO_delivered*[*nextBatch*] in some deterministic order $\{$*TO-Deliver*$\}$
10: *nextBatch* $\leftarrow$ *nextBatch* $+1$
11: **while** *AwaitingToBeDelivered*[*nextBatch*] $\neq \emptyset$ **do**
12: *TO_delivered*[*nextBatch*] $\leftarrow$ *AwaitingToBeDelivered*[*nextBatch*]- *TO_delivered*; atomically deliver *TO_delivered*[*nextBatch*]
13: *nextBatch* $\leftarrow$ *nextBatch*+1
14: **task** launch $\{$*Upon case executed only once per received message*$\}$
15: **upon** *Received - TO_delivered* $\neq \emptyset$ **or** leader has changed **do** $\{$*If upon triggered by a leader change, jump to line 26*$\}$
16: **while** *AwaitingToBeDelivered*[K+1] $\neq \emptyset$ **or** *TO_delivered*[K+1] $\neq \emptyset$ **do**
17: $K \leftarrow K$+1
18: **if** $K =$ *nextBatch* **and** *AwaitingToBeDelivered*[K] $\neq \emptyset$ **and** *TO_delivered*[K] $= \emptyset$ **then**
19: deliver(*AwaitingToBeDelivered*[K])
20: *TO_undelivered* $\leftarrow$ *Received* $-$ *TO_delivered*
21: **if** leader()= p_i **then**
22: **while** *propose*$_K$ **is active do**
23: $K \leftarrow K$+1
24: **start task** propose$_K(K, i, TO_undelivered)$; $K \leftarrow K$+1
25: **else**
26: send($TO\_undelivered$) to leader()
27: **task** propose($L, l, msgSet$) $\{$*Keep on proposing until consensus commits*$\}$
28: committed $\leftarrow$ *false*; consensus$_L \leftarrow$ new consensus()
29: **while not** committed **do**
30: **if** leader()= p_i **then**
31: **if** consensus$_L$.propose($l, msgSet$) = ($commit, returnedMsgSet$) **then**
32: committed $\leftarrow$ *true*
33: $l \leftarrow l$+n
34: send(DECISION,L, $returnedMsgSet$) to all processes
35: **upon receive** m from p_j **do**
36: **if** $m =$ (DECISION,*nextBatch*,$msgSet^{K_{p_j}}$) **or** $m =$ (UPDATE,K_{p_j},*TO_delivered*[K_{p_j}]) **then**
37: **if task** *propose*$_{K_{p_j}}$ **is active then stop task** *propose*$_{K_{p_j}}$
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*: send(UPDATE,L,*TO_delivered*[L]) to p_j $\{$*If* $p_j \neq p_i\}$
41: **else**
42: *AwaitingToBeDelivered*[K_{p_j}] = $msgSet^{K_{p_j}}$; 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* $\cup\ msgSet_{TO\_undelivered}$ $\{$*Consensus messages are added to the consensus box*$\}$

Figure 9. A modular crash-stop variant of Paxos



(a) Modules

(b) Added forced logs

Figure 10. The impact of a crash-recovery model

variable *xmitmsg*. Messages in *xmitmsg* are erased but the Paxos layer stops retransmitting messages except for the DECISION or UPDATE messages once a decision has been reached. The no creation and fair loss properties are trivially satisfied.

```
for each process p_i:
procedure initialisation:
   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                               {Retransmit all messages received and sent}
   while true do
      for all m ∈ xmitmsg do
         s-send(m)
```

Figure 11. Retransmission module

Proposition 18. 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 eventually s-receives m.

Proof. Suppose that p_i s-sends a message m to a process p_j and then does not crash. Assume by contradiction that p_j is correct, yet p_j does not s-receive m. There are two cases to consider: (a) p_j does not crash, or (b) p_j crashes and eventually recovers and remains always-up. For case (a), by the fair loss properties of the links, p_j receives and then s-receives m: a contradiction. For case (b), since process p_i keeps on sending m to p_j, there is a time after which p_i sends m to p_j and none of them crash afterwards. As for case (a), by the fair loss property of the links, p_j eventually receives m, then s-receives m: a contradiction. □

5.2 Round-Based Register

We give in Figure 12 the implementation of a round-based register in a crash-recovery model. The main differences with our crash-stop implementation given in the previous section are the following. As shown in Figure 10(b), a process logs the variables $read_i$, $write_i$ and v_i, in order to be able to recover consistently its precedent state after a crash. A recovery procedure re-initialises the process and retrieves all variables. The send (resp. receive) primitive is also replaced by the s-send (resp. s-receive) primitive.

Proposition 19. *With a majority of correct processes, the algorithm of Figure 12 implements a wait-free round-based register.*

Lemma 20. *Read-write-abort: If $read(k)$ or $write(k, *)$ aborts, then some operation $read(k')$ or $write(k', *)$ was invoked with $k' > k$.*

Lemma 21. *Read-write-commit: If $read(k)$ or $write(k, *)$ commits, then no subsequent $read(k')$ or $write(k', *)$ can*

```
procedure register()                                        {Constructor, for each process p_i}
   read_i ← 0
   write_i ← 0
   v_i ← ⊥
procedure read(k)
   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, v)
   else
      select the [ackREAD,k, k', v] with the highest k'
      return(commit, v)
procedure write(k, 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(commit)
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, v_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
      v_i ← v; store{write_i, v_i}                           {Modified from Figure 5}
      s-send [ackWRITE,k] to p_j
upon recovery do                                             {Added procedure to Figure 5}
   initialisation
   retrieve{write_i, read_i, v_i}
```

Figure 12. A wait-free round-based register in a crash-recovery model

commit with $k' < k$.

Lemma 22. *Read-commit: If* $read(k)$ *= (*$commit, v$*) and* $v \neq \perp$*, then some operation* $write(k', v)$ *was invoked with* $k' \leq k$.

Lemma 23. *Write-commit: If* $write(k, v)$ *commits and no subsequent* $write(k', v')$ *is invoked with* $k' \geq k$ *and* $v' \neq v$*, then any* $read(k'')$ *that commits, commits with* v *if* $k'' > k$.

The proofs for lemmata 20 through 23 are similar to those of lemmata 1 through 4 since: (a) if p_i invokes a $read()$ or a $write()$ operation and then does not crash, by the property of the retransmission module, p_i keeps on sending messages (e.g., READ messages for the $read()$ operation) until it gets a majority of replies (e.g., ackREAD or nackREAD); (b) since all variables are logged before sending any positive acknowledgement messages, a process does not behave differently if it crashes and recovers. If a process crashes and recovers, it recovers its precedent state and therefore acts as if it did not crash.

5.3 Weak Leader Election

The implementation of the weak leader election does not change in a crash-recovery model. However, the failure detector Ω has only been defined in a crash-stop model [2]. Interestingly, its definition (*there is a time after which exactly one correct process* p_l *is always trusted by every correct process*) does not change in a crash-recovery model (the notion of correctness changes though). We give in Appendix B an implementation of the failure detector Ω in a crash-recovery model with partial synchrony assumptions.

5.4 Modular Paxos

Figure 10(b) shows that compared to a crash-stop version, the total order broadcast protocol adds (i) a recovery procedure, and (ii) one forced log to store the set *TO_delivered* and the variable *nextBatch*. We now say that a process TO-Delivers a message m when the process logs m. In a stable period, a process can TO-Deliver a message after three forced logs and two round trip communication steps (if the leader is the process that broadcasts the message). Section 6.4 introduces a powerful optimisation that requires only one forced log at a majority of processes and one round-trip communication step (if the requesting process is leader).

Proposition 24. *With a wait-free round-based consensus, and a wait-free weak leader election, the algorithm of Figure 13 ensures the termination, agreement, validity and total order properties in a crash-recovery model without unstable processes.*

Lemma 25. *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.*

Lemma 26. *Agreement: If a process TO-Delivers a message m, then every correct process eventually TO-Delivers m.*

Lemma 27. *Validity: For any message m, (i) every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process, and (ii) every process p_i TO-Delivers m at most once.*

Lemma 28. *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.*

The proofs for lemmata 25 through 28 are identical to those of from lemmata 13 to 16 since: (a) if p_i TO-Broadcasts m and then does not crash; by the property of the retransmission module, p_i keeps on sending m to the leader, therefore the predicate at line 17 of Figure 13 becomes *true* at the eventual perpetual leader; (b) by the weak leader election property, one of the correct processes will be an eventual perpetual leader p_l that decides; by its definition, p_l is eventually always-up, and then eventually keeps on sending the decision to all processes, therefore all correct processes s-receive the decision (even those that crash and recover); (c) the implementation is build on a wait-free round-based register and on a wait-free round-based consensus that are tolerant to crash-recovery (without unstable processes); (d) when a process crashes and recovers, it retrieves its precedent state by retrieving *TO_delivered* and *nextBatch*; (e) when recovering, *Received* is set to *TO_delivered* otherwise the predicate of line 17 would never be *false* and would keep on proposing messages; and (f) since processes keep on broadcasting messages, the leader process eventually updates a process that has crashed and recovered with all lagging messages.

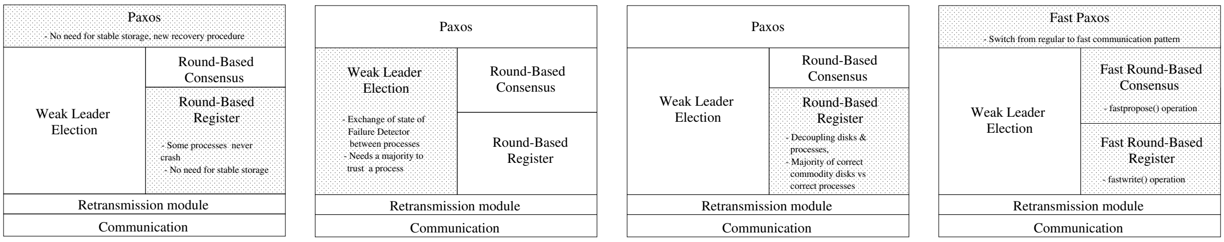
6 The Four Seasons

This section presents four interesting variants of the Paxos protocol. Subsection 6.1 describes a variant of the protocol that alleviates the need for stable storage under the assumption that some processes never crash. This is

```
1:  For each process p_i:
2:  procedure initialisation:
3:     Received[] ← ∅; TO_delivered[] ← ∅; start task{launch}
4:     TO_undelivered[] ← ∅; AwaitingToBeDelivered[]← ∅; K ← 1; nextBatch ← 1
5:  procedure TO-Broadcast(m)
6:     Received ← Received ∪ m
7:  procedure deliver(msgSet)
8:     TO_delivered[nextBatch] ← msgSet - TO_delivered;
9:     atomically deliver all messages in TO_delivered[nextBatch] in some deterministic order
10:     store{TO_delivered,nextBatch}                                   {TO-Deliver, added to Figure 9}
11:     nextBatch ← nextBatch +1       {Stop retransmission module ∀ messages of nextBatch-1 except DECIDE or UPDATE}
12:     while AwaitingToBeDelivered[nextBatch] ≠ ∅ do
13:        TO_delivered[nextBatch] ← AwaitingToBeDelivered[nextBatch]- TO_delivered; atomically deliver TO_delivered[nextBatch]
14:        store{TO_delivered,nextBatch}    {Stop retransmission module ∀ messages of nextBatch except DECIDE or UPDATE}
15:        nextBatch ← nextBatch+1
16:  task launch                                          {Upon case executed only once per received nessage}
17:     upon Received - TO_delivered ≠ ∅ or leader has changed do    {If upon triggered by a leader change, jump to line 28}
18:        while AwaitingToBeDelivered[K+1] ≠ ∅ or TO_delivered[K+1] ≠ ∅ do
19:           K ← K+1
20:           if K = nextBatch and AwaitingToBeDelivered[K] ≠ ∅ and TO_delivered[K] = ∅ then
21:              deliver(AwaitingToBeDelivered[K])
22:        TO_undelivered ← Received − TO_delivered
23:        if leader()= p_i then
24:           while propose_K is active do
25:              K ← K+1
26:           start task propose_K(K, i, TO_undelivered); K ← K+1
27:        else
28:           s-send(TO_undelivered) to leader()
29:  task propose(L, l, msgSet)                                 {Keep on proposing until consensus commits}
30:     committed ← false; consensus_L ← new consensus()
31:     while not committed do
32:        if leader()= p_i then
33:           if consensus_L.propose(l, msgSet) = (commit, returnedMsgSet) then
34:              committed ← true
35:           l ← l+n
36:     s-send(DECISION,L, returnedMsgSet) to all processes
37:  upon s-receive m from p_j do
38:     if m = (DECISION,nextBatch,msgSet^{K_pj}) or m = (UPDATE,K_pj,TO_delivered[K_pj]) then
39:        if task propose_{K_pj} is active then stop task propose_{K_pj}
40:        if K_pj ≠ nextBatch then                                            {p_j is ahead or behind}
41:           if K_pj < nextBatch then                                                   {p_j is behind}
42:              for all L such that K_pj < L < nextBatch: s-send(UPDATE,L,TO_delivered[L]) to p_j     {If p_j ≠ p_i}
43:           else
44:              AwaitingToBeDelivered[K_pj] = msgSet^{K_pj}; s-send(UPDATE,nextBatch-1,TO_delivered[nextBatch-1]) to p_j   {If p_j ≠ p_i}
45:        else
46:           deliver(msgSet^{K_pj})
47:     else
48:        Received ← Received ∪ msgSet_TO_undelivered              {Consensus messages are treated in the consensus box}
49:  upon recovery do                                                           {Added procedure to Figure 9}
50:     initialisation
51:     retrieve{TO_delivered, nextBatch}; K ← nextBatch; nextBatch ← nextBatch+1; Received ← TO_delivered
```

Figure 13. A modularisation of Paxos

obtained mainly by modifying the implementation of our round-based register. Subsection 6.2 describes a variant of the protocol that copes with unstable processes through a modification of our weak leader election implementation. Subsection 6.3 describes a variant of the protocol that guarantees progress even if only one process is correct. This is obtained through an implementation of our round-based register that assumes a decoupling between disks and processes, along the lines of [5]. Subsection 6.4 describes an optimised variant (Fast Paxos) of the protocol that is very efficient in stable periods. These variants are orthogonal, except 6.1 and 6.3 (because of their contradictory assumptions).



(a) Winter (b) Spring (c) Summer (d) Fall

Figure 14. Modified (in shade) modules from a crash-recovery variant

6.1 Winter: Avoiding Stable Storage

Basically, we assume here that some of the processes never crash and, instead of stable storage, we store the crucial information of the register inside "enough" processes (in main memory). The protocol assumes that the number of processes that never crash (n_a) is strictly greater than the number of faulty processes: n_f. As depicted by Figure 14(a), the weak leader election and the round-based consensus remain unchanged. We mainly change the round-based register implementation and we add to the Paxos protocol a recovery procedure that relies on initialisation messages instead of stable storage. Basically, a recovered process p_i asks all other processes to return the set of messages that they have TO-Delivered and p_i initialises its state using those messages.

Round-Based Register. The trick in the round-based register implementation is to ensure that the register's value is "locked" in at least one process that never crashes. Intuitively, any $read()$ or $write()$ uses a threshold that guarantees this property, as we explain below. (The idea is inspired by [1].) When a process recovers, it stops participating in the protocol, except that it periodically broadcasts a RECOVERED message. When a process p_i receives such message from a process p_j, p_i adds p_j to a set R_i of processes (known to have recovered). This scheme allows any process to count the number of *recovered* processes. While collecting ackREAD or ackWRITE messages, if p_i detects that a new process p_k has recovered ($R_i \neq PrevR_i$), p_i restarts the whole procedure of reading or writing. For p_i to commit a $read()$ (resp. $write()$) invocation), p_i must receive max(n_f+1, n-n_f-$|R_i|$) ackREAD (resp. ackWRITE) messages.

Proposition 29. *The algorithm of Figure 15 implements a wait-free round-based register in a crash-recovery model without stable storage assuming that $n_a > n_f$.*

$seqrd$ (resp.$seqwr$) distinguishes the phases when p_i has restarted to s-send READ (resp. WRITE) messages because p_i received a RECOVERED message
procedure register() {*Constructor, for each process* p_i}
  $read_i \leftarrow 0$
  $write_i \leftarrow 0$
  $v_i \leftarrow \perp$
  $R_i \leftarrow \emptyset$; $PrevR_i \leftarrow \emptyset$ {***Added to Figure 5***}
  $seqrd_{p_i} \leftarrow 0$; $seqwr_{p_i} \leftarrow 0$ {*Variable use to distinguish retrial,* ***added to Figure 5***}
procedure read(k)
  repeat {***Added to Figure 5***}
    s-send [READ,k, $seqrd_{p_i}$] to all processes
    $PrevR_i \leftarrow R_i$
    wait until s-received [ackREAD,k, $seqrd_{p_i}$,*,*] **or** [nackREAD,k, $seqrd_{p_i}$] from max(n_f+1, n-n_f-$|R_i|$)processes
  until $R_i = PrevR_i$ {***Added to Figure 5***}
  if s-received at least one [nackREAD,k, $seqrd_{p_i}$] **then**
    return($abort$, v)
  else
    select the [ackREAD,k, $seqrd_{p_i}$, k', v] with the highest k'
    return($commit$, v)
procedure write(k, v)
  repeat {***Added to Figure 5***}
    s-send [WRITE,k, $seqwr_{p_i}$, v] to all processes
    $PrevR_i \leftarrow R_i$
    wait until s-received [ackWRITE,k, $seqwr_{p_i}$] **or** [nackWRITE,k, $seqwr_{p_i}$] from max(n_f+1, n-n_f-$|R_i|$)processes
  until $R_i = PrevR_i$ {***Added to Figure 5***}
  if s-received at least one [nackWRITE,k, $seqwr_{p_i}$] **then**
    return($abort$)
  else
    return($commit$)
task wait until s-receive [READ,k, $seqrd_{p_j}$] from p_j
  if $write_i > k$ **or** $read_i > k$ **then**
    s-send [nackREAD,k, $seqrd_{p_j}$] to p_j
  else
    $read_i \leftarrow k$
    s-send [ackREAD,k, $seqrd_{p_j}$, $write_i$, v_i] to p_j
task wait until s-receive [WRITE,k, $seqwr_{p_j}$, v] from p_j
  if $write_i > k$ **or** $read_i > k$ **then**
    s-send [nackWRITE,k, $seqwr_{p_j}$] to p_j
  else
    $write_i \leftarrow k$
    $v_i \leftarrow v$
    s-send [ackWRITE,k, $seqwr_{p_j}$] to p_j
upon s-receive RECOVERED **from** p_j **do** {***Added procedures to Figure 5***}
  $R_i \leftarrow R_i \cup p_j$
upon recovery do
  initialisation; $read_i \leftarrow \infty$; $write_i \leftarrow \infty$
  s-send RECOVERED to all processes

Figure 15. A wait-free round-based register in a crash-recovery model without stable storage

Lemma 30. *Read-write-abort: If $read(k)$ or $write(k,*)$ aborts, then some operation $read(k')$ or $write(k',*)$ was invoked with $k' > k$.*

Lemma 31. *Read-write-commit: If $read(k)$ or $write(k,*)$ commits, then no subsequent $read(k')$ or $write(k',*)$ can commit with $k' < k$.*

Lemma 32. *Read-commit: If $read(k)$ = $(commit, v)$ and $v \neq \perp$, then some operation $write(k', v)$ was invoked with $k' \leq k$.*

Lemma 33. *Write-commit: If $write(k, v)$ commits and no subsequent $write(k', v')$ is invoked with $k' \geq k$ and $v' \neq v$, then any $read(k'')$ that commits, commits with v if $k'' > k$.*

The proofs for lemmata 29 through 33 are identical to those of lemmata 20 through 23. They are based on the following aspects: (a) we assume that $n_a > n_f$; (b) when a process crashes and recovers, it keeps on sending RECOVERED messages which ensures that a recovered process is never considered correct; and (c) since a process waits for the maximum between n_f+1 and n-n_f-$|R_i|$, the register's value is always locked into an *always-up* process.

The Paxos Variant. Figure 16 presents a Paxos variant for a crash-recovery model without stable storage.

Proposition 34. *With a wait-free round-based consensus, and a wait-free weak leader election, the algorithm of Figure 16 ensures the termination, agreement, validity and total order properties in a crash-recovery model (without any stable storage) assuming that $n_a > n_f$.*

Lemma 35. *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.*

Lemma 36. *Agreement: If a process TO-Delivers a message m, then every correct process eventually TO-Delivers m.*

Lemma 37. *Validity: For any message m, (i) every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process, and (ii) every process p_i TO-Delivers m at most once.*

Lemma 38. *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.*

The proofs for lemmata 34 through 38 are identical to those of lemmata 25 through 28 since the recovery procedure requests every participant to s-send back their state when they s-receive a RECOVERED message. A process that crashes and recovers receives the "latest state" from at least one *always-up* process.

6.2 Spring: Coping with Unstable Processes

We discuss here a Paxos variant that copes with unstable processes, i.e., processes that keep crashing and recovering forever. We adapt our modular protocol by simply changing the implementation of our weak leader election protocol as depicted in Figure 14(b). All our other modules remain unchanged.

Intuitively, the issue with unstable processes is the following. Consider an unstable process p_i (i.e., p_i keeps on crashing and recovering), and suppose that its Ω_i module permanently outputs p_i, whereas the correct processes permanently consider some other correct process p_j as leader. This is possible since Ω "only" guarantees that some correct process is always trusted by every *correct* process. For instance, an unstable process is free to permanently

1: For each process p_i:
2: **procedure** initialisation:
3: *Received*[] $\leftarrow \emptyset$; *TO_delivered*[] $\leftarrow \emptyset$; **start task**$\{$*launch*$\}$
4: *TO_undelivered*[] $\leftarrow \emptyset$; *AwaitingToBeDelivered*[]$\leftarrow \emptyset$; $K \leftarrow 1$; $k \leftarrow 0$; *nextBatch* $\leftarrow 1$
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 deliver all messages in *TO_delivered*[*nextBatch*] in some deterministic order $\{$*TO-Deliver*$\}$
10: *nextBatch* $\leftarrow$ *nextBatch* $+1$ $\{$*Stop retransmission module* $\forall$ *messages of nextBatch-1 except* DECIDE *or* UPDATE$\}$
11: **while** *AwaitingToBeDelivered*[*nextBatch*] $\neq \emptyset$ **do**
12: *TO_delivered*[*nextBatch*] $\leftarrow$ *AwaitingToBeDelivered*[*nextBatch*]- *TO_delivered*; atomically deliver *TO_delivered*[*nextBatch*]
13: *nextBatch* $\leftarrow$ *nextBatch*+1 $\{$*Stop retransmission module* $\forall$ *messages of nextBatch-1 except* DECIDE *or* UPDATE$\}$
14: **task** launch $\{$*Upon case executed only once per received message*$\}$
15: **upon** *Received - TO_delivered* $\neq \perp$ **or** leader has changed **do** $\{$*If upon triggered by a leader change, jump to line 26*$\}$
16: **while** *AwaitingToBeDelivered*[K+1] $\neq \emptyset$ **or** *TO_delivered*[K+1] $\neq \emptyset$ **do**
17: $K \leftarrow K$+1
18: **if** $K =$ *nextBatch* **and** *AwaitingToBeDelivered*[K] $\neq \emptyset$ **and** *TO_delivered*[K] $= \emptyset$ **then**
19: deliver(*AwaitingToBeDelivered*[K])
20: *TO_undelivered* $\leftarrow$ *Received* $-$ *TO_delivered*
21: **if** leader()= p_i **then**
22: **while** *propose*$_K$ **is active do**
23: $K \leftarrow K$+1
24: **start task** propose$_K(K, i, TO_undelivered)$; $K \leftarrow K$+1
25: **else**
26: s-send($TO\_undelivered$) to leader()
27: **task** propose($L, l, msgSet$) $\{$*Keep on proposing until consensus commits*$\}$
28: committed $\leftarrow$ *false*; consensus$_L$ $\leftarrow$ new consensus()
29: **while not** committed **do**
30: **if** leader()= p_i **then**
31: **if** consensus$_L$.propose(l, $msgSet$) = ($commit$, $returnedMsgSet$) **then**
32: committed $\leftarrow$ *true*
33: $l \leftarrow l$+n
34: s-send(DECISION,L, $returnedMsgSet$) to all processes
35: **upon s-receive** m from p_j **do**
36: **if** $m =$ (DECISION,*nextBatch*,$msgSet^{K_{p_j}}$) **or** $m =$ (UPDATE,K_{p_j},*TO_delivered*[K_{p_j}]) **then**
37: **if task** *propose*$_K$ **is active then stop task** *propose*$_K$
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* $\cup$ $msgSet_{TO\_undelivered}$ $\{$*Consensus messages are treated in the consensus box*$\}$
47: **upon** recovery **do** $\{$***Added procedure to Figure 9***$\}$
48: initialisation; s-send(UPDATE,0,$\emptyset$) to all processes

Figure 16. A variant of Paxos in a crash-recovery model without stable storage

elect itself. The presence of two concurrent leaders can prevent the commitment of any consensus decision and hence prevent progress. We basically need to prevent unstable processes from being leaders after some time. We modify our new leader election protocol as follows: (a) every process p_k exchanges the output value of its Ω_k with all other processes, and (b) the function $leader()$ returns p_l only when a majority of processes thinks that p_l is leader. The latter step is required to avoid the following case. Imagine an unstable process p_u that invokes $leader()$ which returns p_u, then crashes, recovers and keeps on doing the same scheme forever. Process p_u always trusts itself which violates the Ω property. By waiting for a majority of processes, we ensure that the values (Ω_i) of at least one correct process belongs to the set $\Omega[]$. Therefore, p_u cannot trusts itself forever (or any unstable processes) since its epoch number is eventually greater than any correct process. This idea, inspired by [7], assumes a majority of correct processes. Note that this assumption is now needed both in the implementation of the register and in the implementation of the leader election protocol.

We give the implementation of this new weak leader election in Figure 17 and it is easy to verify that the implementation is wait-free under the assumption that a majority of processes are correct. Now, the weak leader election exchanges the output of Ω between every process. However, this exchange phase can be piggy-backed on the I-AM-ALIVE messages in the implementation of Ω (see Appendix B). Thus, the exchange phase does not add any communication steps.

1: initialisation: $\Omega[] \leftarrow \perp$; **start task** EXCHANGE
2: **procedure leader**() $\quad$ {***Modified from Figure 7**, for each process p_i*}
3: $\quad$ **wait until** $p_l \in \lceil \frac{n+1}{2} \rceil\ \Omega[k]$
4: $\quad$ **return**(p_l)
5: **task** exchange $\quad$ {***Added task to Figure 7***}
6: $\quad$ **periodically** send Ω_{p_i} to all processes
7: **upon** receive Ω_{p_j} from p_j **do**
8: $\quad$ $\Omega[j] \leftarrow \Omega_{p_j}$

Figure 17. A wait-free weak leader election with Ω and unstable processes

Proposition 39. The algorithm of Figure 17 ensures that some process is an eventual perpetual leader.

Proof. Suppose, by contradiction, there are more than one eventual perpetual leader or there is no eventual perpetual leader. Consider the first case, suppose that there are forever two eventual perpetual leaders. This contradicts the definition of an eventual perpetual leader. Now, consider the second case where there is no eventual perpetual leader. By the property of Ω failure detector, eventually all correct processes trust only one correct process p_l. By line 3 of Figure 17, it is impossible for any process to elect forever a process other than p_l. The *leader*() function is non-blocking since there is a majority of correct processes. So eventually the invocation of *leader*() at every process returns in a bounded time (or the process crashes) and always returns p_l, so there is one eventual perpetual leader p_l: a contradiction. □

6.3 Summer: Decoupling Disks and Processes

The Paxos protocol ensures progress only if there is a time after which a majority of the processes are correct. The need for this majority is due to the fact that a process cannot decide on a given order for any two messages, unless this information is "stored and locked" at a majority of the processes. If disks and processes can be decoupled, which is considered a very reasonable assumption in some practical systems [5], a process might be able to decide on some order as long as it can "store and lock" that information within a majority of the disks. We simply modify the implementation of our round-based register (Figure 14(c)) to obtain a variant of Paxos that exploits that underlying configuration.

In this Paxos variant, we assume that disks can be directly (and remotely) accessed by processes, and failures of disks and processes are separated. Every process has an assigned block on each disk, and maintains a record $dblock[p_i]$ that contains three elements: $read_i$, $write_i$ and v_i; $disk[d_j][p_k]$ denotes the block on disk d_j in which process p_k writes $dblock[p_k]$. We denote by $read_d()$ (resp. $write_d()$) the operation of reading (resp. writing) on a disk. As in [5], we assume that every disk ensures that (i) an operation $write_d(k, *)$ cannot overwrite a value of an earlier round $k' < k$, and (ii) a process must wait for acknowledgements when performing a $write_d()$ operation, and (iii) $write_d()$ and $read_d()$ are atomic operations.

The round-based register protocol works as follows. For the $read()$ operation, a process p_i tries to write$_d$ on each disk p_j its $dblock[p_i]$ ($\forall p_j$ $disk[p_j][p_i]$). After writing, p_i reads$_d$ for any p_j and any p_k: $disk[p_j][p_k]$. If p_i reads$_d$ a block with a round that is lower than the round of the highest $write_i$, the $read()$ operation aborts. Otherwise, the $read()$ commits and returns the value associated with the highest $write_i$. A similar scheme is used for the $write()$ operation. Note that the round-based register implementation is simpler than the previous round-based register due to the usage of disks.

1: **procedure** register() $\{$*Constructor, for each process* $p_i\}$
2: The operation $write_d()$ stores the whole block into disk. For presentation clarity, we have put as a parameter the value that is actually modified.
3: **procedure** read(k)
4: $write_d(k)$ $\{read_i = k\}$
5: $read_d()$ $\{$*Wait for a majority of disk block*$\}$
6: **if** (received a block with $read_j > k$ or $write_j > k$) **then return**($abort, init_i$)
7: choose v_{max} from the block with highest $write_j$; **return**($commit, v_{max}$) $\{v_{max} = \perp$ *if* $write_j = 0\}$
8: **procedure** write(k, v)
9: $write_d(k, v)$ $\{write_i = k, v_i = v\}$
10: $read_d()$ $\{$*Wait for a majority of disk block*$\}$
11: **if** (received a block with $read_j > k$ **or** $write_j > k$) **then return**($abort, v$) **else** return ($commit, v$)
12: **upon recovery do**
13: $read_d(); read_i \leftarrow MAX(read_{received}); write_i \leftarrow MAX(write_{received})$ $\{$*Read all blocks*$\}$
14: $v_i \leftarrow dblock[].v_{write_i}$ $\{$*Take v from the block with the highest* $v_i\}$

Figure 18. A wait-free round-based register built on commodity disks

Lemma 40. *Read-write-abort: If* $read(k)$ *or* $write(k, *)$ *aborts, then some operation* $read(k')$ *or* $write(k', *)$ *was invoked with* $k' > k$.

Proof. Assume that some process p_j invokes a $read(k)$ (resp. $write(k, *)$) that returns $abort$ (i.e., aborts). By the algorithm of Figure 18, this can only happen if some process p_i has a value $read_i > k$ or $write_i > k$ (line 6 and 11), which means that some process has invoked $read(k')$ or $write(k')$ with $k' > k$. □

Lemma 41. *Read-write-commit: If* $read(k)$ *or* $write(k, *)$ *commits, then no subsequent* $read(k')$ *or* $write(k', *)$ *can commit with* $k' < k$.

Proof. Remember that we assume that a $write_d(k', *)$ cannot overwrite$_d$ a $write_d(k, *)$ with $k' < k$. In the algorithm of Figure 18, p_i invokes $write_d()$ in both procedures, therefore p_i cannot commit $read(k')$ or $write(k', *)$ with $k' < k$. □

Lemma 42. *Read-commit: If $read(k)$ = ($commit, v$) and $v \neq \perp$, then some operation $write(k', v)$ was invoked with $k' \leq k$.*

Proof. By the algorithm of Figure 18, if some process p_j commits $read(k)$ with $v \neq \perp$, then some process p_i must have write$_d$ to some disk since v_i is only modified in the $write()$ operation. Otherwise v_{max} would be equal $\perp$. □

Lemma 43. *Write-commit: If $write(k, v)$ commits and no subsequent $write(k', v')$ is invoked with $k' \geq k$ and $v' \neq v$, then any $read(k'')$ that commits, commits with v if $k'' > k$.*

Proof. Assume that some process p_i commits $write(k, v)$, and assume that no subsequent $write(k', v')$ has been invoked with $k' \geq k$ and $v' \neq v$, and that for some $k'' > k$ some process p_j commits $read(k'')$ with v'. Assume by contradiction that $v \neq v'$. Since $read(k'')$ commits with v', by the read-commit property, some $write(k'', v')$ was invoked before or at the same round k''. However, this is impossible since we assumed that no $write(k', v')$ operation with $k' \geq k$ and $v' \neq v$ has been invoked, i.e., v_i remains unchanged to v: a contradiction. □

Proposition 44. *The algorithm of Figure 18 implements a wait-free round-based register.*

Proof. Directly from lemmata 40, 41, 42 and 43 and the fact that we assume a majority of correct disks. □

6.4 Fall: Fast Paxos

In Paxos, when a process p_i TO-Broadcasts a message m, p_i sends m to the leader process p_l. When p_l receives m, p_l triggers a new round-based consensus instance by proposing a batch of messages. A round-based consensus is made up of two phases, a $read$ phase and a $write$ phase. The $read$ phase figures out if some value was already written, while the $write$ phase either writes a new value (if the register contained $\perp$) or rewrites the last written value. In the specific case of $k = 1$ (i.e., the first round), p_1 can safely invoke the $write(1, *)$ operation without reading: indeed, if any other process has read or written any value, the $write(1, *)$ invocation of p_1 aborts. In this case, consensus (if it commits) can be reached significantly faster than in a "regular" scenario.

Interestingly, this optimisation can actually be applied whenever the system stabilises (even if processes do not know when that occurs). Indeed, the key idea behind that optimisation is that p_1 knows that writing directly at round 1 is safe because in case of any other write, p_1's write would be automatically aborted. In fact, once a leader gets elected and commits a value, the leader can send a new message to all processes indicating that, for the subsequent consensus instances, only this process can try to directly write onto the register. This new message can be piggy-backed onto the messages of the $write()$ primitive, thus avoiding any additional communication steps. Moreover, the last decision is piggy-backed onto the next consensus invocation, thus saving one more communication step.

Hence, the optimised protocol goes through two modes. Whenever a leader p_i commits consensus (in the initial regular mode), it switches to the *fast* mode and tries to directly impose its value for next consensus. If the system is stable, p_i succeeds and hence needs only one forced log and one communication round trip. We introduce here a specific

$fastpropose()$ operation that invokes $write()$ directly and ensures that only one process can invoke $fastpropose()$ per consensus, i.e., per batch of messages (independently of the round number). A $fastpropose()$ invokes $write()$ with a round number range between 1 and n, while for $propose()$, i.e., regular $write()$, the round number range starts at n+1. This way, a process can differentiate a $write()$ from a $propose()$ or a $fastpropose()$. If the $fastpropose()$ does not succeed, p_i goes back to the *regular* mode. We implement this mode switching by refining our round-based consensus and round-based register abstractions. We give here the intuition.

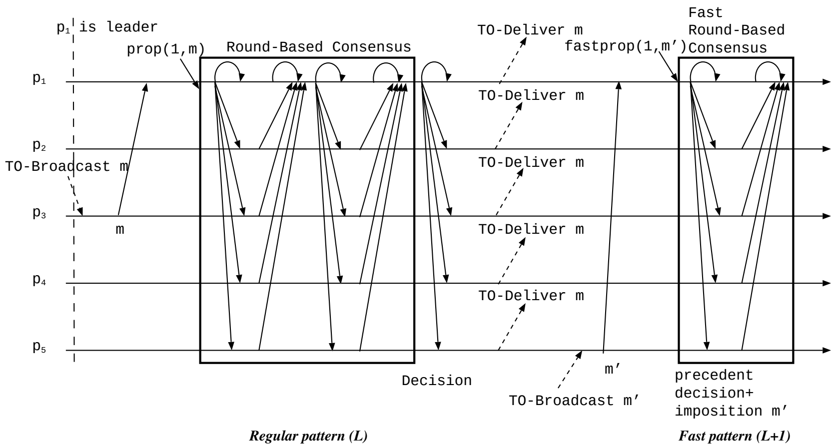


Figure 19. Communication steps for a regular followed by a fast communication pattern

Basically, we change the initialisations of our round-based consensus and round-based register abstractions. We use, in their constructors, a boolean variable *fast* that is set to *true* (resp. *false*) to distinguish the two cases. We add one specific operation $fastpropose()$ to the interface of round-based consensus. Our modular Paxos protocol is also slightly modified to invoke the $fastpropose()$ operation. Figure 19 depicts the different communication steps schemes; for clarity, we omit forced logs. Process p_1 executes a regular communication pattern for message m and then a fast communication pattern for the next consensus (message m'). First, p_3 elects p_1 and sends m to p_1. When p_1 commits consensus for batch L and with the permission to allow the next batch to be performed in a fast mode, p_1 switches to the fast mode for batch L+1. When p_5 TO-Broadcasts m', p_5 elects p_1 and sends m' to p_1. Process p_1 then imposes the decision for batch L+1 and piggy-backs the last decision (L) on the same consensus invocation (L+1). The L+1 batch of messages is decided but will be TO-Delivered only with the next batch of messages (L+2).

Fast Round-Based Register. The fast round-based register has similar $read()$ and $write()$ operations than a regular round-based register. A variable *permission* is added to the returned values of the $write()$ primitive: *permission* is set to *true* if the variable v from the current and the next consensus are empty, otherwise it is set to *false*. The variable *permission* indicates to the upper layer that the process can directly invoke Fast Paxos for the next consensus. If a process p_i receives a nackWRITE message, it returns ($abort$,*false*). If p_i gathers only ackWRITE message, then it returns ($commit$,*true*) only if p_i received only ackWRITE messages with *permission* set to *true*, otherwise p_i returns ($commit$,*false*). Note that v_i is modified and stored after *permission* is set, indeed only one process can perform a Fast Paxos per consensus. Fast round-based register has a different constructor since it extracts (if there is any) the decision that is piggy-backed from the invocation and simulates the reception of a DECIDE message. Note also that line 32 of Figure 20 prevents the violation of the agreement property.

1: **procedure** register() {*Constructor, for each process* p_i}
2: $read_i \leftarrow 0$
3: $write_i \leftarrow 0$
4: $v_i \leftarrow \perp$
5: if any, extract *msgSet* and K_{p_j} and simulate the receive of a message (DECIDE,K_{p_j},*msgSet*) {***Added from Figure 12***}
6: *permission* $\leftarrow$ *false* {***Added from Figure 12***}
7: **procedure read**(k)
8: s-send [READ,k] to all processes
9: **wait until** received [ackREAD,k,*,*] **or** [nackREAD,k] from $\lceil \frac{n+1}{2} \rceil$ processes
10: **if** received at least one [nackREAD,k] **then**
11: **return**($abort$, v)
12: **else**
13: select the [ackREAD,k, k', v] with the highest k'
14: **return**($commit$, v)
15: **procedure write**(k, v) {***Modified from Figure 12***}
16: s-send[WRITE,k, v] to all processes
17: **wait until** received [ackWRITE,k,*] **or** [nackWRITE,k] from $\lceil \frac{n+1}{2} \rceil$ processes
18: **if** received at least one [nackWRITE,k] **then**
19: **return**($abort$,*false*)
20: **else**
21: **if** received at least one [ackWRITE,k,*false*] **then return**($commit$,*false*) **else return** ($commit$,*true*)
22: **task wait until** receive [READ,k] from p_j
23: **if** $write_i > k$ **or** $read_i > k$ **then**
24: s-send [nackREAD,k] to p_j
25: **else**
26: $read_i \leftarrow k$; **store**$\{read_i\}$
27: s-send [ackREAD,k, $write_i$, v_i] to p_j
28: **task wait until** received [WRITE,k, v] from p_j {***Modified from Figure 12***}
29: **if** $write_i > k$ **or** $read_i > k$ **then**
30: s-send [nackWRITE,k] to p_j
31: **else**
32: **if** $k \leq n$ **then** $write_i \leftarrow n$+1 **else** $write_i \leftarrow k$
33: *permission* $\leftarrow$ (($v_i = \perp$) **and** ($v_{i+1} = \perp$))
34: $v_i \leftarrow v$; **store**$\{write_i, v_i\}$
35: s-send [ackWRITE,k,*permission*] to p_j
36: **upon recovery do**
37: initialisation
38: **retrieve**$\{write_i, read_i, v_i\}$

Figure 20. Wait-free fast round-based register

Fast Round-Based Consensus. Fast round-based consensus has a parameterised constructor: *fast* indicates if the mode is fast or not, and the new constructor instantiates a new register using the *fast* parameter. Fast round-based consensus exports the primitive $propose()$ of a regular round-based consensus (augmented with the return value *nextFast*). The variable *nextFast* is a boolean that indicates if the next batch of messages can be executed in a fast manner. Its value is set to the return value of the fast round-based register (*permission*). Moreover, *nextFast* is set in such way that for a particular batch L, it returns *true* only once independantly of the number of invocation of $propose()$ or $fastpropose()$. A process p_i can perform Fast Paxos for batch L+1 only if p_i commits consensus (either by $propose()$ or $fastpropose()$) for batch L with *nextFast* set to *true*. The fast round-based consensus also exports a new primitive $fastpropose()$ that takes as input an integer and an initial value v (i.e., a proposition for the fast consensus). It returns a $status$ in $\{commit, abort\}$, a value v' and a boolean value *nextFast*. The $fastpropose()$ primitive is a $propose()$ primitive that satisfies the validity and agreement properties of the regular $propose()$ primitive plus the following *Fast Termination* property if $fastpropose()$ is invoked only with round number $n \geq k \geq 1$:

- **Fast Termination:** If a process p_i aborts $fastpropose(*, *)$, then some process has performed a different

$fastpropose(-,-)$ invocation; if $fastpropose(*,*)$ commits then no different $fastpropose(-,-)$ can commit.

In fact, the $fastpropose()$ primitive is straightforward to implement since it only invokes the $write()$ primitive with round number between 1 and n of the fast round-based register.

```
procedure consensus(fast)                                   {Constructor, for each process p_i, modified from Figure 6}
   v ← ⊥; reg ← new register(); writeRes ← abort; nextFast ← false      {Initialisation,modified from Figure 6}
procedure propose(k, init_i)
   if reg.read(k) = (commit, v) then
      if (v =⊥) then v ← init_i
      (writeRes,nextFast) ← reg.write(k, v)
      if writeRes=commit then return(commit, v,nextFast) else return(abort, init_i,nextFast)
   return(abort, init_i,false)
procedure fastpropose(k, init_i)                                               {Added from Figure 6}
   (writeRes,nextFast) ← reg.write(k, init_i)
   if writeRes=commit then return(commit, init_i,nextFast) else return(abort, init_i,nextFast)
```

Figure 21. Wait-free fast round-based consensus

Lemma 45. *Fast Termination: If a process p_i aborts $fastpropose(*,*)$, then some process has performed a different $fastpropose(-,-)$ invocation; if $fastpropose(*,*)$ commits then no different $fastpropose(-,-)$ can commit.*

Proof. We assume here that processes invoke $fastpropose()$ only with round number $n \geq k \geq 1$. There are two cases to consider: (i) two different processes invoke $fastpropose()$ for the same consensus, or (ii) a process invokes $fastpropose()$ twice for the same consensus. Consider case (i), let us assume by contradiction that two different processes p_i and p_j invoke $fastpropose()$. Assume moreover that p_i returns from $fastpropose()$, by line 32 of Figure 20, when p_j tries to invoke $fastpropose()$, by the algorithm of Figure 20, p_j cannot succeed since $write_i$ is already set to n+1: a contradiction. Now consider case (ii). Assume that p_i invokes $fastpropose()$ twice for the same consensus number, since $write_i$ is stored, p_i cannot commit twice $fastpropose()$ with *nextFast* set to *true*: a contradiction.□

Proposition 46. *Figure 21 implements a wait-free fast round-based consensus in a crash-recovery model.*

Proof (sketch). The proof is based on lemma 45 and the fact that the proofs of the validity and agreement properties are similar to the proofs of lemmata 7 and 8. □

Fast Paxos. Intuitively, once a process p_i returns from $propose()$ or $fastpropose()$ with *nextFast* set to *true* for batch L, it implies that a process has the permission to execute a fast consensus, i.e., invoke $fastpropose()$ for batch L+1. We slightly modify the Paxos algorithm by adding an array $fast$[] that is set to *false* initially. When a process p_i decides for batch L (in the regular mode), p_i sends the decision to every process and sets the variable *fast*[L+1] to *true* if $fastpropose()$ or $propose()$ returns with *nextFast* set to *true* (changes from a regular to a fast mode for the next consensus). The next time p_i invokes a new consensus (*fast*[L] is *true*), p_i (i) piggy-backs the last decision (if there is any) to the new instantiation of consensus, and (ii) invokes $fastpropose()$. This invocation has a different impact on the round-based register as explained earlier. When p_i commits $fastpropose()$, p_i (a) does not need to send the decision to every process since the decision is piggy-backed onto the next consensus invocation, and (b) sets *fast* of the next consensus to *true* so that p_i can perform again a Fast Paxos. When p_i aborts $fastpropose()$, p_i sets *fast* back to *false* since p_i cannot force the decision for this consensus, i.e., the communication pattern becomes regular again.

Note that it is necessary in the fast mode that the last decision (if there is any) to be piggy-backed onto the invocation of the constructor of our round-based register. Otherwise, the process that creates the round-based register will not be able to TO-Deliver the last decision. Since there can be concurrent executions of consensus, when a process commits a regular consensus for batch L, the next fast consensus will not always be batch L+1. Consider the following example, if a process p_i starts three consensus for batch number L=1,2, and 3; when p_i commits batch number L=1, p_i sets *fast* to *true* for batch number 2 and not 4 (only the subsequent batch number of L is set to *true* and not the last batch number started). Note also that the last decision piggy-backed is *TO_deliver*[L-1] but it can be empty. In this case, the last decision piggy-backed is the latest decision that p_i has, e.g, *AwaitingToBeDelivered*[latestDecisionReceived] or *TO_delivered*[lastestTODelivered]. Note that we assume here that lines 24 and 25 are executed atomically.

Lemma 47. *There can be only one invocation of $fastpropose()$ per consensus.*

Proof. By the algorithm of Figure 22, processes invoke $fastpropose()$ only with round number $n \geq k \geq 1$. There are two cases to consider: (i) two different processes invoke $fastpropose()$ for the same consensus, or (ii) a process invokes $fastpropose()$ twice for the same consensus. Consider case (i), let us assume by contradiction that two different processes p_i and p_j invoke $fastpropose()$ for consensus number L+1. For both processes, to invoke $fastpropose()$ for consensus L+1, *fast*[L+1] must be set to *true*, which requires a process to perform a successful $propose()$ (or $fastpropose()$) which returns *nextFast* as *true* for consensus L. Assume that p_i returns from $propose()$ (or $fastpropose()$) with *nextFast* to *true*: a majority of processes have returned with *permission* set to *true* (hence $v_L \neq \perp$ at a majority of processes) and no process has returned with *permission* set to *false*. When p_j invokes $propose()$ or $fastpropose()$, by the algorithm of Figure 20, p_j has to return with *nextFast* to *false* since two majorities will always intersect: a contradiction. Now consider case (ii). Assume that p_i invokes $fastpropose()$ twice for the same consensus number L+1, by the algorithm of Figure 22, p_i must have crashed and recovered between the two invocations of $fastpropose()$. When p_i recovers, *fast*[L+1] is reset to *false* (initialisation). To invoke $fastpropose()$ after having recovered, p_i has to perform a successful $propose()$ (or $fastpropose()$) with *nextFast* set to *true* for consensus L. This is impossible because a majority of processes have already their $v_L \neq \perp$: a contradiction. □

Proposition 48. *With a wait-free round-based consensus, and a wait-free weak leader election, the algorithm of Figure 22 ensures the termination, agreement, validity and total order properties in a crash-recovery model.*

Lemma 49. *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.*

Lemma 50. *Agreement: If a process TO-Delivers a message m, then every correct process eventually TO-Delivers m.*

Lemma 51. *Validity: For any message m, (i) every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process, and (ii) every process p_i TO-Delivers m at most once.*

Lemma 52. *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.*

By lemma 47, the proofs for lemmata 49 through 52 are identical to those of lemmata 25 through 28 since (a) the

1: For each process p_i:
2: **procedure** initialisation:
3: *Received*[] ← ∅; *TO_delivered*[] ← ∅; *fast*[] ← {*false*,..} {***Modified from Figure 13***}
4: *TO_undelivered* ← ∅; *AwaitingToBeDelivered*[]← ∅; $K \leftarrow 1$; *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 deliver all messages in *TO_delivered*[*nextBatch*] in some deterministic order
10: **store**{*TO_delivered,nextBatch*}
11: *nextBatch* ← *nextBatch* +1 {*Stop retransmission module* ∀ *messages of nextBatch-1 except* DECIDE *or* UPDATE}
12: **while** *AwaitingToBeDelivered*[*nextBatch*] ≠ ∅ **do**
13: *TO_delivered*[*nextBatch*] ← *AwaitingToBeDelivered*[*nextBatch*]- *TO_delivered*; atomically deliver *TO_delivered*[*nextBatch*]
14: **store**{*TO_delivered,nextBatch*}
15: *nextBatch* ← *nextBatch*+1 {*Stop retransmission module* ∀ *messages of nextBatch-1 except* DECIDE *or* UPDATE}
16: **task** launch {*Upon case executed only once per received message*}
17: **upon** *Received* - *TO_delivered* ≠ ∅ **or** leader has changed **do** {*If upon triggered by a leader change, jump to line 28*}
18: **while** *AwaitingToBeDelivered*[K+1] ≠ ∅ **or** *TO_delivered*[K+1] ≠ ∅ **do**
19: $K \leftarrow K$+1
20: **if** K = *nextBatch* **and** *AwaitingToBeDelivered*[K] ≠ ∅ **and** *TO_delivered*[K] = ∅ **then**
21: deliver(*AwaitingToBeDelivered*[K])
22: *TO_undelivered* ← *Received* − *TO_delivered*
23: **if** leader()= p_i **then**
24: **while** *propose*$_K$ **is active do**
25: $K \leftarrow K$+1
26: **start task** propose$_K(K, i, TO_undelivered)$; $K \leftarrow K$+1
27: **else**
28: s-send($TO\_undelivered$) to leader()
29: **task** propose($L, l, msgSet$) {***Modified from Figure 13***}
30: committed ← *false*
31: **if** *fast*[L] **then** {***Added from Figure 13***}
32: piggy-back *TO_delivered*[L-1] (if not empty) otherwise latest decision onto next instantiation and invocation of consensus
33: consensus$_L$ ← new consensus(*true*)
34: **if** consensus$_L$.fastpropose($l, msgSet$) = ($commit, returnedMsgSet$,*nextFast*) **then**
35: **if** L = *nextBatch* **then** deliver($returnedMsgSet$) **else** *AwaitingToBeDelivered*[L] = $returnedMsgSet$; *committed* ← *true*
36: *fast*[L] ← *false*; *fast*[L+1] ← *nextFast*
37: **if** consensus$_L = \perp$ **then** consensus$_L$ ← new consensus(*false*)
38: **while not** committed **do**
39: $l \leftarrow l + n$
40: **if** leader()= p_i **then**
41: **if** consensus$_L$.propose($l, msgSet$) = ($commit, returnedMsgSet$,*nextFast*) **then**
42: committed ← *true*; s-send(DECISION,L, $returnedMsgSet$) to all processes; *fast*[L+1] ← *nextFast*
43: **else**
44: *fast*[L+1] ← *false*
45: **upon s-receive** m from p_j **do**
46: **if** m = (DECISION,*nextBatch*,$msgSet^{K_{p_j}}$) **or** m = (UPDATE,K_{p_j},*TO_delivered*[K_{p_j}]) **then**
47: **if task** *propose*$_{K_{p_j}}$ **is active then stop task** *propose*$_{K_{p_j}}$
48: **if** K_{p_j} ≠ *nextBatch* **then** {p_j *is ahead or behind*}
49: **if** K_{p_j} < *nextBatch* **then** {p_j *is behind*}
50: **for all** L such that $K_{p_j} < L <$ *nextBatch*: s-send(UPDATE,L,*TO_delivered*[L]) to p_j {*If* $p_j \neq p_i$}
51: **else**
52: *AwaitingToBeDelivered*[K_{p_j}] = *msgSet*$^{K_{p_j}}$; s-send(UPDATE,*nextBatch-1*,*TO_delivered*[*nextBatch-1*]) to p_j {*If* $p_j \neq p_i$}
53: **else**
54: deliver(*msgSet*$^{K_{p_j}}$)
55: **else**
56: *Received* ← *Received* ∪ $msgSet_{TO\_undelivered}$ {*Consensus messages are treated in the consensus box*}
57: **upon recovery do**
58: initialisation
59: **retrieve**{*TO_delivered, nextBatch*}; $K \leftarrow$ *nextBatch*; *nextBatch* ← *nextBatch*+1; *Received* ← *TO_delivered*

Figure 22. Fast Paxos in a crash-recovery model

properties of the $fastpropose()$ primitive are more restrictive than the $propose()$ primitive; and (b) the properties of the regular $propose()$ remain the same.

7 Related Work

The contribution of this paper is a *faithful deconstruction* of the Paxos replication algorithm. Our deconstruction is faithful in the sense that it preserves the efficiency of the original Paxos algorithm. This promotes the implementation of the algorithm in a modular manner, and the reconstruction of variants of it that are customised for specific environments.

In [12, 16], the authors focused on the consensus part of Paxos with the aim of either explaining the algorithm and emphasising its importance [12] or proving its correctness [16]. In [12, 16], the authors discussed how a state machine replication algorithm can be constructed as a sequence of consensus instances. As they pointed out however, that might not be the most efficient way to obtain a replication scheme. Indeed, compared to the original Paxos protocol, additional messages and forced logs are required when relying on a consensus box. This is in particular because the very nature of traditional consensus requires every process to start consensus, i.e, adds messages compared to Paxos, and, in a crash-recovery model, every process needs to log its initial value. Considering a finer-grained and round-based consensus abstraction, separated from a leader election abstraction, is the key to our faithful deconstruction of the Paxos replication algorithm. Our round-based consensus allows a process to propose more than once without implying a forced log, and allows us to merge all logs at the lowest abstraction level while exporting the round number up to the total order broadcast layer.

Our round-based consensus abstraction is somehow similar to the "weak" consensus abstraction identified by Lampson in [12]. There are two fundamental differences. "Weak" consensus does not ensure any liveness property. As stated by Lampson, the reason for not giving any liveness property is to avoid the applicability of the impossibility result of [4]. Our round-based consensus specification is weaker than consensus and does not fall into the impossibility result of [4], but nevertheless includes a liveness property. The termination property of our round-based consensus coupled with our leader election property is precisely what allows us to ensure progress at the level of total order broadcast.

In [5], a variant of Paxos, called Disk Paxos, decouples processes and stable storage. A crash-recovery model is assumed and progress requires only one process to be up and a majority of functioning disks. Thanks again to our modular approach, we implement Disk Paxos by only modifying the implementation of our round-based register. The algorithm of Section 6.3 is faithful to Disk Paxos in that both have the same number of forced logs, messages and communication steps.[14] Note that our leader election implementation that copes with unstable processes can be used with Disk Paxos to improve its resilience.

Independently of Paxos, [15] presented a replication protocol that also ensures fast progress in stable periods of the system: our Fast Paxos variant can be viewed as a modular version of that protocol. In [13], a new failure detector, $\diamond C$, is introduced. This failure detector, which is shown to be equivalent to Ω, adds to the failure detection capabil-

[14]Variables bal, $mbal$ and inp in [5] correspond to $write_i$, $read_i$ and v_i in our case, while a ballot number in [5] corresponds to a round number in our case. As described in [5], Disk Paxos has two phases: (i) choose a value v, and (ii) try to commit v. In fact, phase 1 corresponds to the $read()$ operation of our round-based register, while phase 2 corresponds to its $write()$ operation. In both phases, processes perform one forced log (write_d) and read_d all blocks. Our $read()$ and $write()$ operations also perform the same steps (write_d and read_d all disks).

ity of $\diamond S$ [3] an *eventual leader election* flavour. Informally, this flavour allows every correct process to eventually choose the same correct process as leader and eventually ensure fast progress. We have shown that Ω can be directly used for that purpose, and we have done so in a more general crash-recovery model. Finally, [17] have given a total order broadcast in a crash-recovery model based on a consensus box [3]. As we pointed out, by using consensus as a black box, all processes need to propose an initial value which, in a crash-recovery model, means that they all need a specific forced log for that (this issue was also pointed in [17]). Precisely because of our round-based consensus abstraction, we are able to alleviate the need for this forced log.

Acknowledgements. We are very grateful to Marcos Aguilera and Sam Toueg for their helpful comments on the specification of our register abstraction. We would also like to thank the anonymous reviewers for their very helpful comments.

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] 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.

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

[4] 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.

[5] E. Gafni and L. Lamport. Disk paxos. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC-14)*, Lecture Notes in Computer Science, Toledo, Spain, October 2000. Springer-Verlag.

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

[7] R. Guerraoui and A. Schiper. Gamma-Accurate failure detectors. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG-10)*, number 1151 in Lecture Notes in Computer Science, Bologna, Italy, October 1996. Springer-Verlag.

[8] 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.

[9] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[10] 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.

[11] L. Lamport. The part-time parliament. Technical Report 49, Systems Research Center, Digital Equipement Corp, Palo Alto, September 1989. A revised version of the paper also appeared in ACM Transaction on Computer Systems vol.16 number 2.

[12] 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.

[13] M. Larrea, A. Fernandez, and S. Arévalo. Eventually consistent failure detectors (short abstract). In *Proceedings of the 14th International Symposium on Distributed Computing (DISC-14)*, Lecture Notes in Computer Science, Toledo, Spain, October 2000. Springer-Verlag.

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

[15] 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.

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

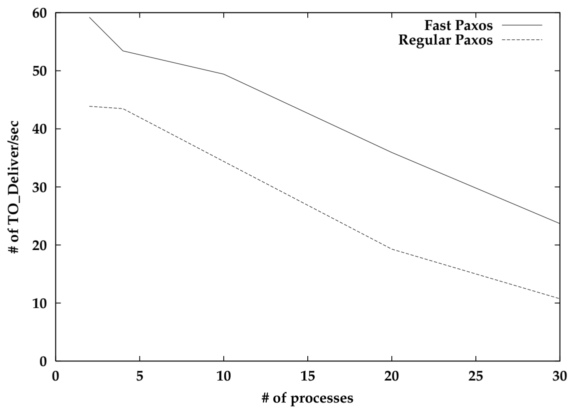
[17] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous systems where processes can crash and recover. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems (ICDCS-20)*, pages 288–295, Taipei, Taiwan, April 2000.

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

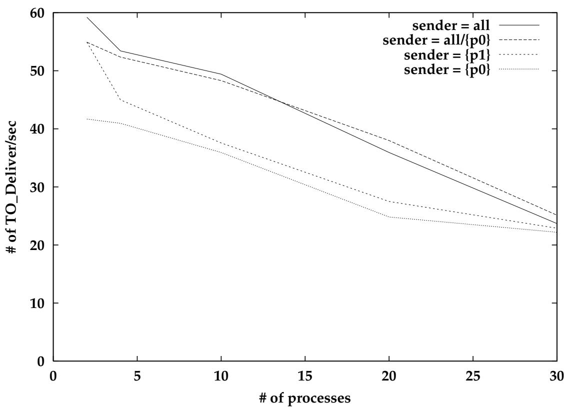
[19] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

A Optional Appendix. Performance measurements

We have implemented our abstractions on a network of Java machines as a library of distributed shared objects. We give here some performance measurements of our modular Paxos implementation in different configurations. These measurements were made on a LAN interconnected by Fast Ethernet (100Mb/s) on a normal working day. The LAN consisted of 60 UltraSUN 10 (256Mb RAM, 9 Gb Harddisk) machines. All stations were running Solaris 2.7, and our implementation was running on Solaris Java HotSpot™ Client VM (build 1.3.0_01, mixed mode). The effective message size was of 1Kb and the performance tests consider only cases where as many broadcasts as possible are executed. In all tests, we considered stable periods where process p_0 was the leader and one process was running per machine.



(a) Fast Paxos vs Regular Paxos



(b) Varying the number of broadcasters (Fast Paxos)

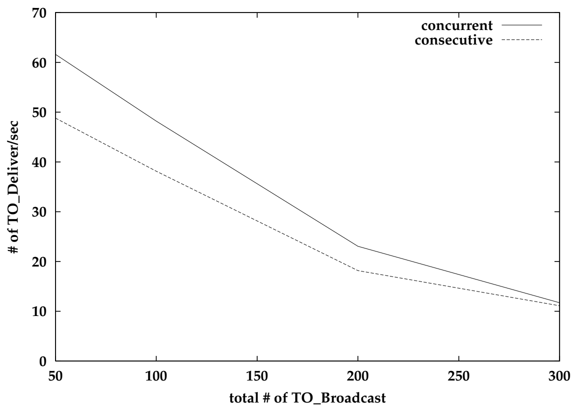
Figure 23. Broadcast performance

Figure 23(a) depicts the throughput difference between Regular Paxos and Fast Paxos. Not surprisingly, Fast Paxos has a higher throughput. The overall performance of both algorithms decreases since the leader has to send and receive messages from an increasing number of processes.

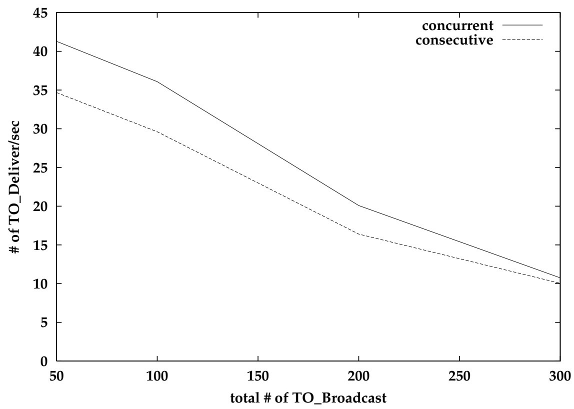
Figure 23(b) depicts the performance of Fast Paxos when the number of broadcasting processes increases. We considered four cases, (i) only the leader broadcasts, (ii) one process other than the leader broadcasts, (iii) all processes except the leader broadcast, and (iv) all processes broadcast. Distributing the load of the broadcasting processes to a larger number of processes improves the average throughput. As expected, the throughput is lower when the leader is the unique broadcasting process, since it is the most overloaded. Case (iii) has a better throughput than case (iv) after 12 processes since the leader does not broadcast and can allow more processing power than case (iv). This shows that broadcasting messages slows down a process, and this is also verified by the increased throughput when another process than the leader (case ii) is broadcasting.[15]

Figure 24 compares Fast Paxos in two different modes: (i) concurrent consensus instances are started, and (ii) only consecutive consensus instances are launched. Not to overwhelm the process with context switching, Paxos is implemented using a thread pool that is limited to ten, i.e., at most ten concurrent consensus run at each process. The

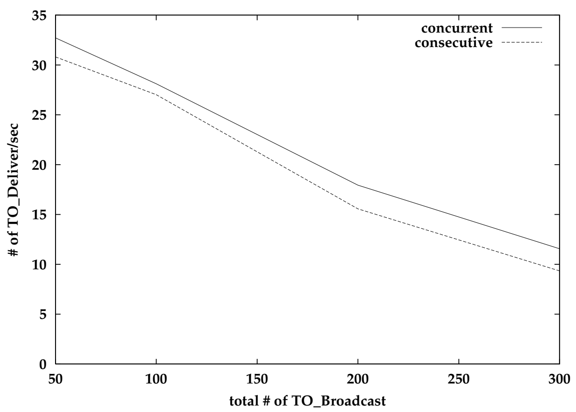
[15] When increasing the number of processes, the performances come close to each other because the capacity of Paxos is reached.



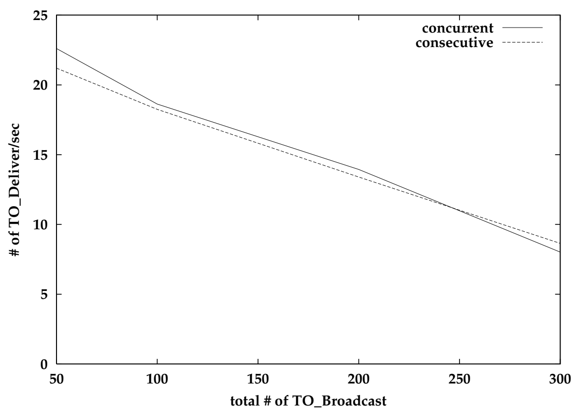
(a) 3 processes



(b) 6 processes



(c) 10 processes

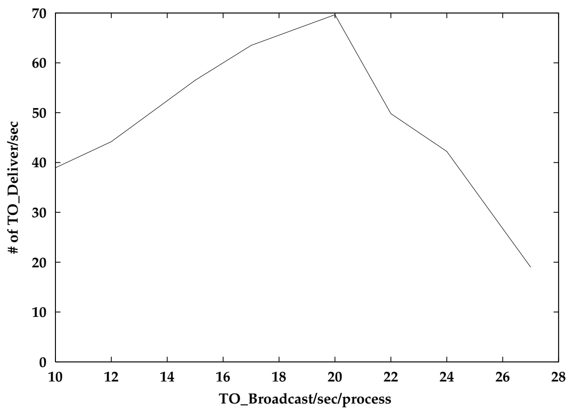


(d) 20 processes

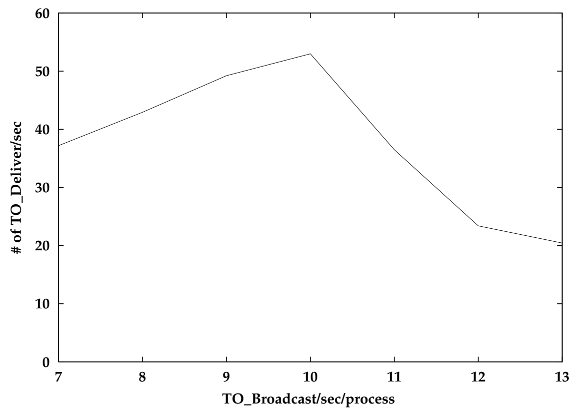
Figure 24. Concurrent vs consecutive (Fast Paxos)

throughput in both modes decreases as the number of protocol instances increases. At first, the concurrent version gives better performance, but this diminishes as the number of broadcast increases. In fact, the increasing computation needed (in the task *launch*) impedes the performance of the concurrent version, i.e., performance degrades. The results show that the more process a system has, the less difference there is in throughput between consecutive and concurrent executions, i.e, when there are more processes in the system, there are less consensus instances that are launched.

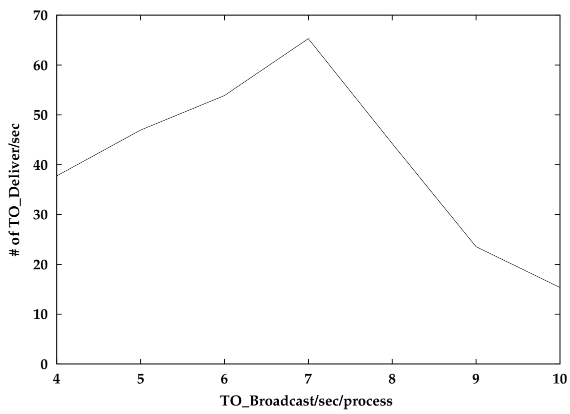
Figure 25 depicts the broadcast rate at which the best throughput can be achieved from 4 to 10 processes. For all cases, the throughput increases (approximately) linearly until a certain point, e.g., up to 10 broadcast/sec/process for a six processes system and then the throughput falls suddenly linearly. Above the breakpoint, the leader again becomes the bottleneck, its task *receive* is overwhelmed by the number of broadcasts it has to handle, thus delaying new protocol instances.



(a) 4 processes



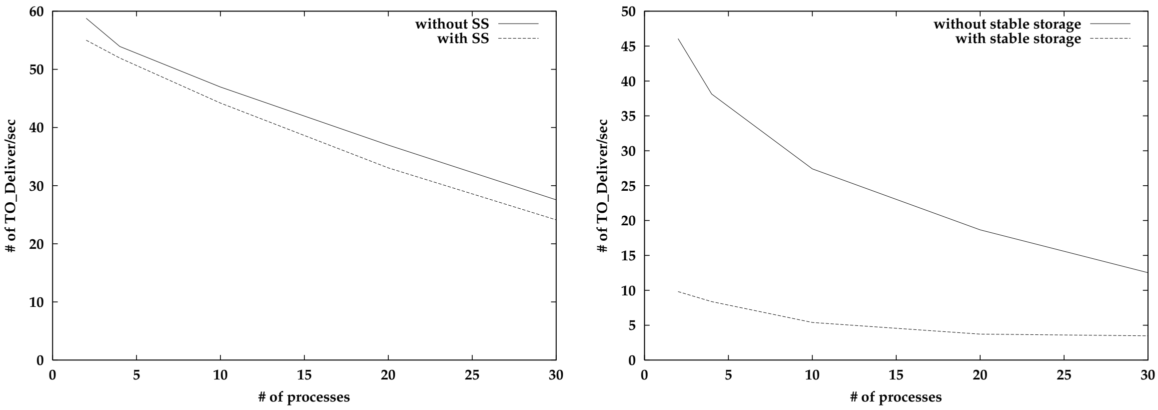
(b) 6 processes



(c) 10 processes

Figure 25. Best throughput (Fast Paxos)

Figure 26(a) depicts the impact of forced logs for the Fast Paxos algorithm. When forced logs are removed, the increased performance is minimal since the algorithm is fine-tuned and waits for a certain number of broadcast messages before launching a consensus. The TO-Delivery rate is by far better when a consensus is launched for a certain number of messages rather than starting a consensus for each single broadcast message. The number of consensus becomes too big and slows down the algorithm. Due to this optimisation, there are few instances of consensus per second and hence few stable storage access per second. Therefore, upon removal of stable storage, the performance improvement is not drastic as one might think. This result shows that the winter season protocol is not really useful for a practical system.[16] However, Figure 26(b) shows that forced logs have an impact on performance. If Fast Paxos launches a large number of consensus per second, i.e., a consensus is started consecutively for each single broadcast message. (There are no other consensus instance running in parallel, but there can be many consensus instances per second.) In this case, the impact of forced logs is quite significant, as shown in Figure 26(b).



(a) Concurrent execution

(b) Consecutive execution

Figure 26. Comparison between forced logs and no stable storage (Fast Paxos)

Finally, Figure 27 gives the recovery time required by a process depending on the number of messages retrieved from the stable storage. The number of retrieved messages is proportional to the number of reads from the disk, thus increasing the recovery time.

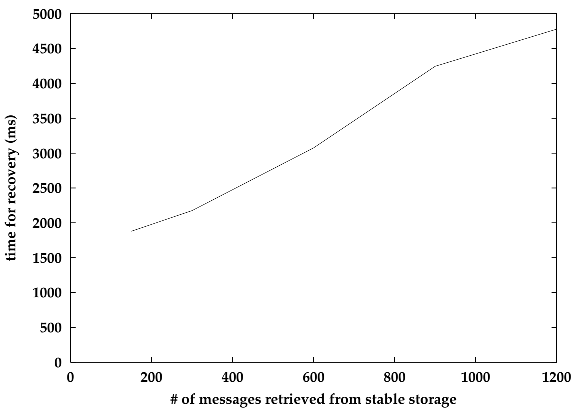


Figure 27. Recovery time

[16]Moreover, Note that for a long-lived application, this model is not really practical, since every process is likely to crash and recover at least once during the life of the application.

B Optional Appendix. Implementation of Ω in a Crash-Recovery Model with partial synchrony

Figure 28 gives the implementation of the failure detector Ω in a crash-recovery model with partial synchrony assumptions. We assume that message communication times are bound by an unknown period but hold after some global stabilisation time. Intuitively, the algorithm works as follows. A process p_i keeps track of the processes that it trusts in a set denoted *trustlist*. A process p_i keeps on sending I-AM-ALIVE messages to every process. Periodically, p_i removes of its *trustlist* the processes from which it did not receive, within a certain threshold, any I-AM-ALIVE message. When p_i receives an I-AM-ALIVE message from some process p_j and if p_j was not part of the *trustlist*, p_i then adds p_j to its *trustlist* and increments p_j's threshold. However, an unstable process can be trusted, therefore the algorithm counts the number of times that a process crashes and recovers. This scheme allows a process to detect when a process crashes and recovers, an unstable process has an unbouded epoch number at a correct process, while a correct process has an epoch number that stops increasing. When p_i crashes and recovers, p_i sends a RECOVERED message to every process (line 8). When p_j receives a RECOVERED message from p_i, p_j updates the epoch number of p_i at line 21 and p_j adds p_i to its *trustlist*. Variable Ω.trustlist contains the process, within the trustlist, that has the lowest epoch number (line 15), and if several of these exist, select the one with the lowest id.

Processes exchange their epoch number and take the maximum of all epoch numbers to prevent the following case. Assume that processes p_2, p_3, p_4 never crash and that process p_1 crashes and recovers. When p_1 recovers, assume that every process except p_1 receives the RECOVERED message from p_1. Therefore, p_1 has epoch$_{p_1} = 0, 0, 0, 0$, while the other processes have epoch$_{p_{2,3,4}} = 1, 0, 0, 0$. Each process has the same trustlist, indeed Ω_{p_1} outputs p_1 and $\Omega_{p_{2,3,4}}$ outputs p_2 which violates the property of Ω, exchanging their epoch number and taking the maximum such case is avoided. Therefore, when receiving the trustlist, p_i also takes the maximum between its epoch number and the one it received from p_j. Note that the MIN function gives the *first* index that realises the minimum.

Proposition 62. *The algorithm of Figure 28 satisfies the following property in a crash-recovery model with partial synchrony assumptions: There is a time after which exactly one correct process is always trusted by every correct process.*

Proof. There is a time after which every correct process stops crashing and remains always-up. Therefore, every correct process keeps on sending I-AM-ALIVE message to every process. Thanks to the partial synchrony assumptions, we know that after some global stabilisation time, a message does not take longer than a certain period of time to go from one process to another. Eventually, every process guesses this period of time by incrementing Δ_{p_i} at line 19. By the fair loss property of the links, every correct process then receives an infinite number of times I-AM-ALIVE messages. Therefore, every correct process eventually has the same set trustlist and epoch list, indeed they output all the same process. Eventually, this process is correct since the algorithm chooses the process with the lowest epoch number (remember that an unstable process has a non decreasing epoch number at a correct process). $\square$

for each process p_i:
upon initialisation or recovery **do**
  Ω.trustlist $\leftarrow \perp$; trustlist$_{p_i} \leftarrow \Pi$
  for all $p_j \in \Pi$ **do**
    $\Delta_{p_i}[p_j] \leftarrow$ default time-out interval
    $epoch_{p_i}[p_j] \leftarrow 0$
  start task$\{$*updateD*$\}$
  if *recovery* **then** send(RECOVERED) to all
task updateD
  repeat periodically
    send (I-AM-ALIVE,epoch$_{p_i}$) to all processes
    for all $p_j \in \Pi$ **do**
      if $p_j \in$ trustlist$_{p_i}$ **and** p_i did not receive I-AM-ALIVE from p_j during the last $\Delta_{p_i}[p_j]$ **then**
        trustlist$_{p_i} \leftarrow$ trustlist$_{p_i} \setminus \{p_j\}$
    Ω.trustlist $\leftarrow$ MIN($p_k \in$ trustlist$_{p_i} \mid p_k =$ MIN($epoch_{p_i}$))
upon receive m from p_j **do**
  if $m =$ (I-AM-ALIVE,epoch$_{p_j}$) **then**
    if $p_j \notin$ trustlist$_{p_i}$ **then**
      trustlist$_{p_i} \leftarrow$ trustlist$_{p_i} \cup \{p_j\}$; $\Delta_{p_i}[p_j] \leftarrow \Delta_{p_i}[p_j] + 1$
    for all $p_k \in \Pi$ **do**
      epoch$_{p_i}[p_k] \leftarrow$ MAX(epoch$_{p_j}[p_k]$, epoch$_{p_i}[p_k]$)
  else if $m =$ RECOVERED **then**
    $epoch_{p_i}[p_j] \leftarrow epoch_{p_i}[p_j] + 1$; trustlist$_{p_i} \leftarrow$ trustlist$_{p_i} \cup \{p_j\}$

Figure 28. Implementing Ω in a crash-recovery model with partial synchrony assumptions