ACM SIGACT News Distributed Computing Column 11

Sergio Rajsbaum*

Abstract

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

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

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

Reconstructing Paxos [1]

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

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

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

Abstract

The celebrated Paxos algorithm of Lamport implements a fault-tolerant deterministic service by replicating it over a distributed message-passing system. In a companion paper [2], we presented a deconstruction of the algorithm by factoring out its fundamental algorithmic principles within two abstractions: an eventual leader election and an eventual register abstractions. Using those abstractions, we show in this paper how to reconstruct, in a modular manner, powerful variants of Paxos. In particular, we show how to (1) alleviate the need for stable storage access 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.

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

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

The Island of Paxos used to host a great civilisation which had developed a sophisticated part-time parliament protocol. Paxons codified various aspects of their parliament protocol which enabled them to 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 the 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. Their indiscipline prevented progress in the protocol. However, as the election subprotocol used to choose the parliament president was also 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 season 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. Again, this optimisation was obtained through a simple refinement of the subprotocol used to propose new decrees.

1 Introduction

The Paxos algorithm [6, 7] implements a highly-available deterministic service by replication it over several processes in a message-passing system. The algorithm maintains replica consistency by ensuring total order delivery of requests to the replicas. In a companion paper [2], we described the "Paxos algorithm" using two abstractions: $\diamond$Register (eventual register) and $\diamond$Leader (eventual leader election). Roughly speaking, the $\diamond$Register is used to "store and lock" the decision value, whereas $\diamond$Leader eventually elects a single correct process to impose the decision value. In [2], we described how to implement the abstractions in a crash-stop model and construct a variant of Paxos as a composition of a $\diamond$Leader and multiple $\diamond$Register instances. We then describe how to transpose the algorithm to the crash-recovery model with stable storage (where Paxos was originally described) by modifying the $\diamond$Register implementation.

This paper shows how to use our $\diamond$Register and $\diamond$Leader abstractions to build four interesting variants of Paxos, each particularly adapted for a specific system setting. Starting from the original Paxos (in the crash-recovery model with stable storage), each variant can be constructed by modifying the implementation of only one of the abstractions, and some minor changes to the overall Paxos algorithm. The ease of these transpositions illustrates the advantage of describing Paxos using our abstractions.

We first describe in Section 3 a variant of the Paxos that alleviates the need for stable storage access under the assumption that some of the processes never crash (during the lifetime of the of an execution instance). This is obtained mainly by modifying the implementation of $\diamond$Register. Then we describe in Section 4 a variant of Paxos that copes with unstable processes through a $\diamond$Leader implementation which exploits the presence of a correct majority to filter out unstable leaders. We describe in Section 5 a variant of Paxos that guarantees progress even if only one process is correct [4]. This is obtained through an implementation of $\diamond$Register that assumes a decoupling between disks and processes, along the lines of [4]. Finally we describe in Section 6 an optimised variant (Fast Paxos) [6, 8] of Paxos that is very efficient in stable periods. We give a discussion of correctness for the variant in Section 3. Correctness arguments for Section 4 and Section 6 are rather straightforward, and that of Section 5 is detailed in [4].

In Figure 1, we give an overview of the similarities and differences of the four variants from our Paxos implementation in [2]. Interestingly, these variants are orthogonal and can be combined, except the first and the third (because of their contradictory assumptions).

2 Background

$\diamond$Register can be accessed by a propose() primitive which either aborts or returns a decision value such that: (1) (*validity*) if a process decides v then some process proposed v, (2) (*agreement*) no two processes decide differently, and (3) (*termination*) if a single correct process proposes an infinite number of times, it eventually decides.

$\diamond$Leader is accessed through a leader() primitive which ensures that (1) after a process invokes leader(), either the process crashes or the invocation returns and (2) there is a time after which every completed leader() invocation returns the same correct process.

Paxos is essentially a total order broadcast algorithm. Total order broadcast is defined by primitives TO-Broadcast and TO-Deliver primitives and satisfies the following properties:

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

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

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

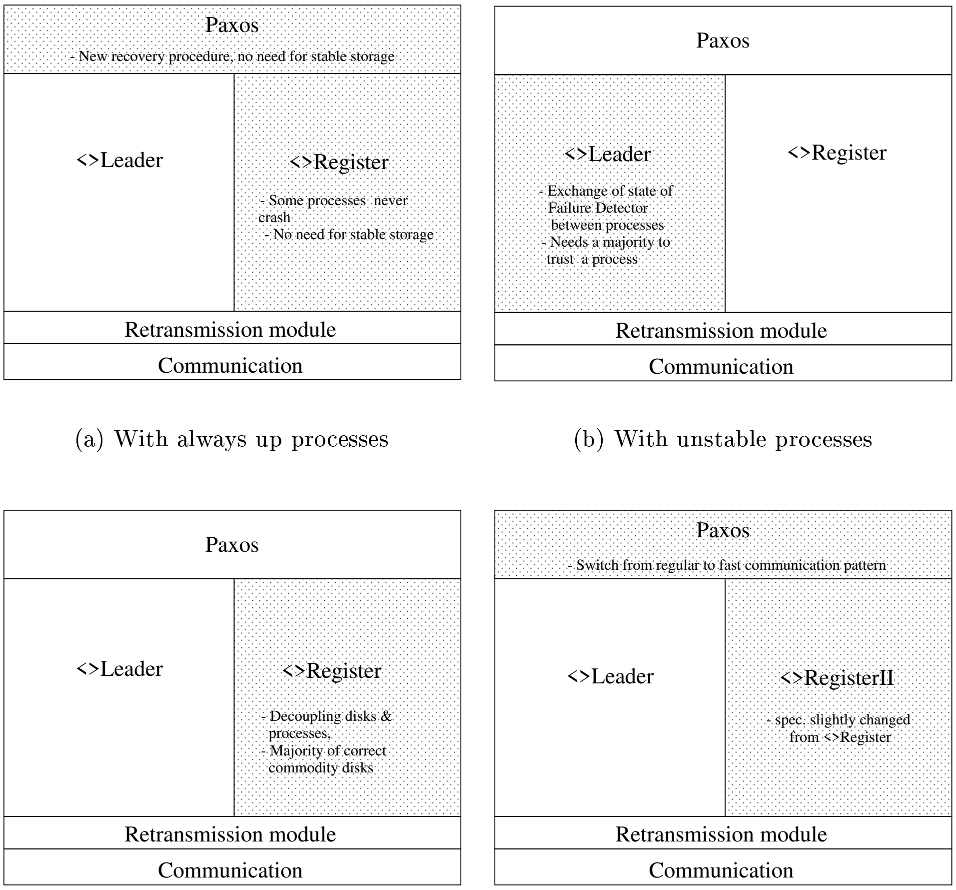
- Total order: Let p_i and p_j be any two processes that TO-Deliver some message m. If p_i TO-Delivers some message m' before m, then p_j also TO-Delivers m' before m.

To evaluate the performance of Paxos and its variants, we consider runs in *stable periods*, which may be the most frequent in practice if process crashes and communication time-outs are rare. In a stable period, there are no processes crashes or recoveries, a majority of processes are up, and $\diamond$Leader elects the same correct process at every process which is up. (However, processes may crash before or after a stable period.)

3 Avoiding Stable Storage

We assume here that some of the processes never crash, and instead of stable storage, we store the crucial information of the register inside these processes (in main memory). Note that these processes are not known in advance and might change from one run to the other. More precisely, we assume that the number of processes that never crash (n_a) is strictly greater than the maximum number of faulty processes (t).[2] As depicted by Figure 1(a), the $\diamond$Leader implementation remains

[2]Furthermore, we assume that t is known while n_a is not.



(a) With always up processes

(b) With unstable processes

(c) Disk Paxos

(d) Fast Paxos

Figure 1: Modified (in shade) modules from the original Paxos deconstruction

unchanged. We mainly change the $\Diamond$Register implementation (the modifications are inspired by [1]) and we add to the Paxos algorithm 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.

3.1 The $\Diamond$Register algorithm

Figure 2 describes our $\Diamond$Register implementation. Roughly speaking, the implementation ensures that the register's value is "locked" in at least one process that never crashes. Each phase (read and write) uses a threshold for the number of replies to guarantee this property. Each process maintains a variable *recovered* which is true iff the process has crashed at least once. Once a process recovers, it sends every ack/nack message with $recovered = true$. If a process p_i receives a message with $recovered = true$ from a process p_j, then p_i adds p_j to a set R_i of processes (known to have recovered). A read or a write phase consists of one or more *subphases*; each subphase consists of sending a READ (or WRITE) message to all and then receiving replies. In a subphase at p_i, if p_i detects no new recovered processes while collecting ackREAD (or ackWRITE) messages, then the subphase is the last one in that phase. Otherwise (i.e., while collecting ackREAD or ackWRITE messages, p_i detects that a new process p_k sends a message with $recovered = true$, and thus, $R_i \neq PrevR_i$), p_i starts another subphase to collect a new set of replies.

Inside a phase, a process p_i distinguishes between the acks of the current subphase from those of any earlier subphases by incrementing a counter $seqrd_i$ (or $seqwr_i$) and including the counter in the READ (resp. WRITE) message of the subphase. However, it is possible that p_i executes read subphase 1 of round 1 (i.e., $seqrd_i = 1$ and $k = 1$) and crashes before receiving the replies, and then recovers and again executes read subphase 1 of round 1, then p_i cannot distinguish the replies of the former subphase from that of the current one. This problem can be easily resolved by attaching a unique message identifier to READ and WRITE messages. For clarity of presentation, we assume that every recovered process can distinguish between the replies of any earlier subphase from the current one.

For any given phase, only the replies of the last subphase is considered; replies of all previous subphases are discarded. The rest of the algorithm is similar to the the $\Diamond$Register implementation in [2], which can be summarized as follows. On invoking propose(v), p_i increments the round number by n, starts the read phase and collect the replies (of the last subphase). If p_i receives any nackREAD message, the invocation aborts. Otherwise, p_i selects a $val \neq \bot$ with the highest $write$. If all vals with the highest $write$ are $\bot$, then p_i selects the invocation value v. Thereafter, p_i starts a write phase with the value selected in the read phase and collects the replies (of the last subphase). If p_i receives any nackWRITE message, the invocation aborts, otherwise, it returns the value selected in the read phase.

3.2 Correctness

The validity property of the $\Diamond$Register implementation (Figure 2) is straightforward. Termination follows from the observation that (1) there is a time after which there are no new processes which recover (as there are no unstable processes, processes crash and recover only finitely often), (2) the property of the retransmission module [2],[3] and (3) there are at least $t + 1$ processes which never

[3] A retransmission module is characterized by the s-send and s-receive primitives. It provides the following additional property over a fair lossy channel: if p_i s-sends a message to p_j, p_i does not crash after sending the message, and p_j is correct, then p_j s-receives the message an infinite number of times.

$seqrd$ (resp.$seqwr$) distinguishes the phases when p_i has restarted to s-send READ (resp. WRITE) messages
procedure register() {*Constructor, for each process* p_i}
  $read_i \leftarrow 0$
  $write_i \leftarrow 0$
  $val_i \leftarrow \bot$
  $v^* \leftarrow \bot$
  $k = i - n$
  $recovered \leftarrow false$ {*true iff* p_i *crash-recovers at least once*}
  $R_i \leftarrow \emptyset$; $PrevR_i \leftarrow \emptyset$ {R_i *is the set of processes from which* p_i *received recovered = true*}
  $seqrd_i \leftarrow 0$; $seqwr_i \leftarrow 0$ {*Variable used to distinguish retrial*}
procedure propose(v)
  $k \leftarrow k + n$
  repeat {*Repeat until no new process joins* R_i}
    $PrevR_i \leftarrow R_i$; $seqrd_i \leftarrow seqrd_i + 1$
    s-send [READ,$k, seqrd_i$] to all processes
    wait until s-received [ackREAD,$k, seqrd_i$,$*$, $*$, $*$] **or** [nackREAD,$k, seqrd_i, *$] from $n - t$ processes including at least $t + 1$ messages with $recovered = false$
    $R_i = R_i \cup \{p_j | p_i$ received $recovered = true$ from $p_j\}$
  until $R_i = PrevR_i$
  if s-received at least one [nackREAD,$k, seqrd_i, *$] **then**
    return($abort$)
  else
    select the [ackREAD,$k, seqrd_i, k', val'$,$*$] with the highest k'
    if $val' \neq \bot$ **then**
      $v^* \leftarrow val'$
    else
      $v^* \leftarrow v$
    repeat {*Repeat until no new process joins* R_i}
      $PrevR_i \leftarrow R_i$; $seqwr_i \leftarrow seqwr_i + 1$
      s-send [WRITE,$k, seqwr_i, v^*$] to all processes
      wait until s-received [ackWRITE,$k, seqwr_i, *$] **or** [nackWRITE,$k, seqwr_i, *$] from $n - t$ processes including at least $t + 1$ messages with $recovered = false$
      $R_i = R_i \cup \{p_j | p_i$ received $recovered = true$ from $p_j\}$
    until $R_i = PrevR_i$
    if s-received at least one [nackWRITE,$k, seqwr_i, *$] **then**
      return($abort$)
    else
      return(v^*)
task wait until s-receive [READ,$k, seqrd_j$] from p_j
  if $write_i \geq k$ **or** $read_i \geq k$ **then**
    s-send [nackREAD,$k, seqrd_j, recovered$] to p_j
  else
    $read_i \leftarrow k$
    s-send [ackREAD,$k, seqrd_j, write_i, val_i, recovered$] to p_j
task wait until s-receive [WRITE,$k, seqwr_j, v$] from p_j
  if $write_i > k$ **or** $read_i > k$ **then**
    s-send [nackWRITE,$k, seqwr_j, recovered$] to p_j
  else
    $write_i \leftarrow k$
    $val_i \leftarrow v$
    s-send [ackWRITE,$k, seqwr_j, recovered$] to p_j
upon recovery do
  initialisation; $R_i \leftarrow \{p_i\}$; $recovered = true$

Figure 2: A $\diamond$Register implementation in a crash-recovery model without stable storage

crash (because $n_a > t$). Now we discuss agreement.

We say that a phase (or subphase) is *complete* if the invoking process does not crash before completing the phase (resp. subphase). We say that a process p_i *participates* in a subphase if p_i replies to the READ or WRITE message of the subphase, and the reply of p_i is one of the replies collected by the subphase (in line 16 or line 30). Consider any two complete phases A and B. Let us denote the last subphase of A by A_z and the last subphase of B by B_z. We claim that *there is a process p_i such that p_i participates in A_z and B_z, and p_i has not crashed before sending replies to both A_z and B_z.* Since the result (success or abort) of a phase is determined by the replies received in the last subphase, roughly speaking our claim implies that, any two phases overlap at at least one process which has never crashed.

Recall that the correctness of the $\diamond$Register in crash-stop model relied on the fact that for any two phases, there is process p_i which participated in both phases (since two majorities always overlap). In crash-recovery model with stable storage, the common participant p_i might have crashed and recovered between the two phases; however every participant (including p_i) stores the crucial information in the stable storage. However, in the absence of stable storage in a crash-recovery model, we can only rely on participants which have not crashed before replying to both the phases. Our claim says that such a process exists for any two phases. Thus, once we show the above claim, the proof of agreement is similar to that in [2].

Let $P(A_z)$ and $P(B_z)$ be the set of processes which participated in subphase A_z and B_z, respectively. $P(A_z)$ contains $n-t$ processes of which $t+1$ replied to A_z before crashing. Similarly, $P(B_z)$ contains $n-t$ processes of which $t+1$ replied to B_z before crashing. Thus, there are two processes $p_c, p_d \in P(A_z) \cap P(B_z)$ such that, p_c did not crash before sending any reply to A_z and p_d did not crash before sending any reply to B_z. Let $c(A_z)$ and $c(B_z)$ be the time when p_c replied to subphases A_z and B_z, respectively. Similarly, define $d(A_z)$ and $d(B_z)$ for p_d.

Suppose by contradiction to our claim that every process that participates in both A_z and B_z has crashed at least once before it has replied to both the subphases. Then, p_c crashed before $\max\{c(A_z), c(B_z)\}$. Since p_c has not crashed before sending any reply to A_z, p_c has crashed between $c(A_z)$ and $c(B_z)$, and hence, $c(A_z) < c(B_z)$. Similarly, p_d has crashed between $d(B_z)$ and $d(A_z)$, and $d(B_z) < d(A_z)$. Let t and t' be the time when the first message (READ or WRITE) in A_z and B_z is sent, respectively.

We claim that $d(B_z) < t$. Suppose by contradiction that $d(B_z) \geq t$. We have already shown that p_d crashes between $d(B_z)$ and $d(A_z)$. From the definition of t we know that the second last subphase of phase A has finished before t. Since, p_d has crashed for the first time after $d(B_z) \geq t$, it follows that p_d has crashed for the first time after the second last subphase of phase A has completed. Thus, in the last subphase A_z, p_d is a recently recovered process. Therefore, from the algorithm of Figure 2, A_z cannot be the last subphase of phase A; a contradiction.

As t is the time at which the first message of A_z is sent and $d(A_z)$ is the time when p_d replies to A_z, we have $t < d(A_z)$. Thus, $d(B_z) < t < d(A_z)$. Similarly, we can show that $c(A_z) < t' < c(B_z)$.

Since t' is the time when the first message (READ or WRITE) in B_z is sent, and $d(B_z)$ is the time when p_d replied to B_z, we have $t' < d(B_z)$. Thus, $c(A_z) < t' < d(B_z) < t$, i.e., p_c replies to A_z before the first message of A_z was sent; a contradiction.

3.3 The total ordering algorithm

In the crash-recovery model without stable storage, it is impossible to satisfy the second validity property of the total order broadcast abstraction: *for any message m, every process p_i To-Delivers a message at most once.* Roughly speaking, on receiving a decided message set v after a crash-recovery, it is impossible for a process to determine whether it has already delivered v before crashing.

The Paxos variant without stable storage is identical to the original Paxos deconstruction [2], except for the following three changes: (1) the $\Diamond$Register implementation of Figure 2 is used, (2) all stable storage accesses are removed, and (3) the new recovery procedure is as follows:

- initialize; send(UPDATE, 0, $\emptyset$) to all

Recall that in the Paxos algorithm with stable storage, processes log the TO-Delivered message and the next expected batch (*nextBatch*) in the stable storage. In this variant (without the stable storage), on recovery, a process p_i sends an UPDATE message to all, stating that it has delivered the decided message set till batch 0 (i.e., p_i has not delivered any batch.) On reception of this UPDATE message, every process p_j replies with all the messages p_j has already delivered. Thus, p_i learns about the already delivered batches.

4 Coping with Unstable Processes

We discuss here a Paxos variant that copes with unstable processes, i.e., processes that keep crashing and recovering forever. (Recall that the original Paxos algorithm assumed that no process is unstable.) We adapt our modular deconstruction of Paxos by simply changing the implementation of our $\Diamond$Leader as depicted in Figure 1(b). All other modules remain unchanged.

Intuitively, the issue with unstable processes is the following. Consider an unstable process p_u, and suppose that its Ω_u module permanently outputs p_u, whereas the correct processes permanently consider some other correct process p_l as leader. This is possible since Ω only guarantees that some correct process is always trusted by every *correct* process; an unstable process is free to permanently elect itself. The presence of two concurrent leaders can prevent the commitment of any $\Diamond$Register decision and hence prevent progress.

To prevent unstable processes from being elected leader after some time, we modify the $\Diamond$Leader implementation as follows: (a) every process p_i exchanges the output value of its Ω_i with all other processes, and (b) p_i maintains an array $A_i[\ ]$ where $A_i[j]$ contains the most recent value of Ω received from p_j; the *leader*() at p_i returns p_l only if a majority of items in $A_i[\ ]$ is p_l (i.e., a majority of processes sent a message to p_i saying that p_l is the output of their local Ω module).

Consider a correct process p_i. Eventually Ω at p_i and all other correct processes permanently output some correct process p_l. Furthermore, eventually p_i receives $\Omega_{p_j} = p_l$ from every correct process p_j. Hence, eventually A_i contains p_l at a majority of indices, and then every subsequent *leader*() invocation at p_i returns p_l. Now consider an unstable process p_u. To compute *leader*(), p_u needs to consider a majority of items in $A_u[\ ]$, and hence, p_u necessarily considers the output of Ω from at least one correct process (because there are a majority of correct processes). Thus, eventually every majority of items in A_u contains p_l, which ensures that the output of *leader*() at p_u cannot be different from the correct process p_l. As a consequence, eventually no unstable process can elect a leader different from correct processes.

This idea, inspired by [5], assumes a majority of correct processes. Note that this assumption is now needed both in the implementation of $\Diamond$Register and in the implementation of $\Diamond$Leader. We give the new implementation of $\Diamond$Leader in Figure 3. Although the implementation introduces extra messages to exchange the output of Ω between processes, in practice, these exchange messages can be piggy-backed on the I-AM-ALIVE messages used in most failure detectors implementations [3, 1].

```
initialisation: A_i[] ← [⊥, ..., ⊥]; start task EXCHANGE
procedure leader()
   wait until there is a process p_l s.t. | {j | A_i[j] = p_l} | ≥ ⌈(n+1)/2⌉
   return(p_l)
task exchange
   periodically send Ω_{p_i} to all processes
upon receive Ω_{p_j} from p_j do
   A_i[j] ← Ω_{p_j}
```

Figure 3: A ◇Leader implementation tolerating unstable processes

5 Decoupling Disks and Processes

The original Paxos algorithm ensures progress only if a majority of the processes are correct. This majority is required because 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, 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.

Disk Paxos [4] is a variant of Paxos that uses commodity disks as witnesses to "store and lock" information instead of witness tasks at participating processes. A disk stores a set of values (divided into blocks) and remote processes can atomically read the values stored in the disk or can atomically modify its content.[4] Disk Paxos assumes that only a minority of disks may fail (by stopping to respond to any read or write request), and any number of processes may fail by crashing, although processes might recover subsequently. Even if only one process is up, Disk Paxos can progress if a majority of disks are up. The practical motivations behind using disks instead of processes as witnesses are twofold [4]: (1) disk are generally cheaper than processes and (2) disks may fail less often than processes since disks do not execute complicated application programs. It is straightforward to express Disk Paxos using our abstractions. Only the implementation of ◇Register needs modification from the original Paxos algorithm. Figure 4 describes the ◇Register implementation.

We assume that (a) a majority of the disks are correct, (b) the read and write at disks are atomic, (c) for every invocation on a correct disk d, the disk eventually sends an indication of the invocation being completed, and (d) for every process p_i and disk d, the write invocations of p_i on d reach d in the same order in which they were invoked by p_i.[5] Each disk d is divided into n blocks: $disk[d][p_1], ..., disk[d][p_n]$. Each block can be read and modified without modifying other blocks. Process p_i writes only at $disk[d][p_i]$ and can read any block. Upon invoking propose(v), p_i proceeds in two phases denoted by Phase 1 and Phase 2. Each phase can succeed or abort, and p_i executes Phase 2 only if Phase 1 succeeds. The round number is represented using two variables: (1) k_1 is used to distinguish Phase 1 of different invocations and takes the values $i, i+n, i+2n$, and so on; (2) k_2 is used to distinguish Phase 1 of an invocation from Phase 2, if Phase 1 succeeds then k_2 is set to k_1. Processes have an estimate v^* of the possible decision value, initialized to $\perp$.

In each phase, at a majority of disks d, p_i writes $[k_1, k_2, v^*]$ at $disk[d][p_i]$ and then reads all other blocks at d. A phase aborts iff it reads a block with k_1 greater than its own k_1. If p_i successfully

[4] Disk Paxos differs from "classic Paxos" because a disk cannot be programmed; otherwise, we could just program the disks similar to the witness tasks of processes in the original Paxos, and then use the corresponding algorithm.

[5] Otherwise, if an incomplete write invocation lingers in the system for an arbitrary amount of time, it might overwrite a value which has been successfully stored and locked [4].

completes Phase 1, then before starting Phase 2 p_i sets k_2 to k_1 and updates v^* as follows. Among the set of blocks read in Phase 1 with $v^* \neq \perp$, p_i updates its v^* to the v^* of the block with the highest k_2. If every block read has $v^* = \perp$, then v^* is updated to the v: the input value of the propose(). If Phase 2 succeeds, the propose() invocation returns v^*.

If p_i crashes and recovers, p_i updates its local variable before starting Phase 1. On recovery, p_i reads its own block at a majority of disk, and updates k_1, k_2, and v^* to the corresponding values in the block with the highest k_1.

```
procedure ◇Register()                                   {Constructor, for each process p_i}
  k_1 ← i − n; k_2 ← 0; v* ← ⊥          {denoted mbal, bal, and inp, respectively in [4]}
  blockSeen ← ∅
procedure propose(v)
  if first invocation after a recovery then
    recover()
  k_1 ← k_1 + n
  for every disk d do
    write [k_1, k_2, v*] at disk[d][p_i]
    ∀p_j ∈ Π, read disk[d][p_j]
  until done at a majority of disks
  if read any disk[d][p_j] s.t. disk[d][p_j].k_1 > k_1 then
    return(abort)
  else
    k_2 ← k_1
    blockSeen ← set of disk[d][p_j] read s.t. disk[d][p_j].v* ≠ ⊥
    if blockSeen ≠ ∅ then
      v* ← v* of the disk[d][p_j] ∈ blockSeen with highest disk[d][p_j].k_2
    else
      v* ← v
    for every disk d do
      write [k_1, k_2, v*] at disk[d][p_i]
      ∀p_j ∈ Π, read disk[d][p_j]
    until done at a majority of disks
    if read any disk[d][p_j] s.t. disk[d][p_j].k_1 > k_1 then
      return(abort)
    else
      return(v*)
procedure recover()
  for every disk d do
    read disk[d][p_i]
  until done at a majority of disks
  select block' = disk[d][p_i] with the highest value of disk[d][p_i].k_1
  k_1 ← block'.k_1
  k_2 ← block'.k_2
  v* ← block'.v*
```

Figure 4: A ◇Register implementation using commodity disks

6 Fast Paxos

In this section we present FastPaxos, a variant of Paxos that is optimized for the stable periods. In Paxos [2], when a process p_i TO-Broadcasts a message m, p_i sends m to the leader process p_l. On receiving m, p_l triggers a new *Converge* task which repeatedly invokes propose() on a ◇Register instance with increasingly higher round numbers until some value is decided. A propose() invocation is made up of two phases: read and write. A read phase figures out if some value has been already decided with a lower round number and tries to get a promise from processes to reject any read or write with a lower round number. (The propose() invocation aborts if the read phase detects the same or a higher round number at some process.) The write phase tries to decide on the value read in the previous phase.

It is easy to see that, for any given ◇Register instance, if p_1 invokes propose() with round number 1, p_i can safely skip the read phase because there can be no invocation with a lower round number. (Recall that propose() invocations of p_i has round numbers $i, i+n$, and so on.) We call such invocations "fastpropose". Thus, if p_1 is elected leader in failure-free stable runs (nice runs), then in those runs a decisions can be reached significantly faster than in the regular scenario.

Interestingly, this optimisation can 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 there can be no propose() invocation with a lower round number. Thus, if any leader p_l gets a guarantee that no process can invoke propose() with a round number lower than l, then p_l can safely invoke a fastpropose. Hence, if the system is stable, the leader can decide within two communication steps and a stable storage access.

We implement the above optimization as follows. Recall that the messages are delivered in ascending order of their batch number. (Inside a batch, messages are delivered in lexicographic order.) Each batch L is agreed using a distinct ◇Register instance, reg_L. For the optimization, we also use reg_L to select a "privileged process" which is allowed to invoke a fastpropose on the register instance of the next batch, reg_{L+1}. However, to ensure agreement, we require that there is at most one privileged process per batch, and to achieve the desired optimization, we require that the leader is selected as the privileged process in the stable periods.

6.1 ◇RegisterII

We introduce a variant of the ◇Register, denoted by ◇RegisterII. Processes access ◇RegisterII through propose() invocations. A process invokes propose() with two arguments, $v \in Values$ (where *Values* is the set of possible proposal values), and *fast* $\in \{$*true*,*false*$\}$. The invocation returns a pair $(v, nextFast)$ where v is a value in *Values* $\cup$ *abort*, and *nextFast* $\in \{$*true*,*false*$\}$. If a process invokes propose$(v, *)$ then we say that the process proposes v, and if an invocation by a process returns $(v, *)$ then we say that the process decides v. We say that an invocation by a process p_i on a register reg is *fresh* if the invocation is the first invocation by p_i on reg or the first invocation by p_i on reg after p_i recovers from a crash. (For any given p_i and reg, there may be multiple fresh invocations.) ◇RegisterII satisfies the validity and the termination properties of ◇Register, and the following two properties:

- **Conditional Agreement:** If at most one fresh invocation has *fast* = *true* then no two processes decide differently.
- **Exclusion:** (1) At most one propose() invocation returns *nextFast* = *true*, and (2) if a propose() invocation returns *nextFast* = *false*, then there was another propose() invocation.

An implementation of ◇RegisterII is presented in Figure 5, which is a simple modification of the ◇Register implementation in crash-recovery model [2]. Recall that in the ◇Register implementation in [2], every fresh invocation has a round number between 1 and n: a fresh invocation by process p_i had round number i. In our ◇RegisterII implementation, a fresh invocation by process p_i has round number i only if it is invoked with *fast* = *true*; otherwise a fresh invocation from p_i has round number $i + n$. Non-fresh invocations always have round number greater than n. If any invocation is fresh and has *fast* = *true*, then the invocation skips the read phase.

To see the conditional agreement property assume that at most one fresh invocation on the register has *fast* = *true*. If no invocation skips the read phase then no two processes can decide differently (follows from the correctness of ◇Register implementation in [2]). If an invocation skips the read phase then from the implementation, it follows that it is fresh and has *fast* = *true*. From our assumption, there is at most one such invocation. From the implementation, it follows that this invocation has round number between 1 and n, and all other invocations have round number greater than n. Thus, it can safely skip the read phase because there can be no invocation with a lower round number.

To implement the exclusion property of ◇RegisterII we slightly modify the write phase. On receiving a WRITE message from some invocation, a witness task replies an ack or a nack, and also sends an indication whether this invocation is the first one to change the register value *val* at that witness task.[6] If an invocation receives an indication from a majority of witness tasks that the invocation was the first one to change their *val*, then it returns *nextFast* = *true*. Since two majorities always overlap, no other invocation can return *nextFast* = *true*. Furthermore, it immediately follows that if some invocation returns *nextFast* = *false* then there must be another invocation which has changed *val* at some witness task.

6.2 FastPaxos

To implement FastPaxos, we replace ◇Register by ◇RegisterII, and perform some minor changes to the regular Paxos algorithm. Primarily, we use the exclusion property of the batch L register to ensure the condition required for agreement on the batch $L + 1$ register.

Every process keeps an array $fast$ which is intialized to *false* at every index, during intialization as well as after recovery from a crash. Primarily, we modify the Converge task. For batch L at p_i, the propose() invocation has input parameters *msgSet* (the message set which p_i proposes for batch L) and *fast[L]* (the L^{th} index of the *fast* array which is initialized to *false*). If a propose() invocation in the Converge task succeeds in deciding on the *msgSet* for batch L, *fast[L+1]* is set to *nextFast* returned by the invocation.

Our Paxos deconstruction in [2] requires that the underlying ◇Register abstraction provides agreement property: no two processes decide differently. However, in the present case, the underlying ◇RegisterII abstraction only provides the conditional agreement property. Thus, for the correctness of the algorithm, we need that the invocation on the ◇RegisterII instances are made in such a way that the "condition" of the conditional agreement property is satisfied. We now show how the above-mentioned modification to the Converge task in FastPaxos ensures the condition required by the conditional agreement property of ◇RegisterII.

Suppose by contradiction that the FastPaxos algorithm invokes two fresh invocations with *fast* = *true* on reg_{L+1} (register for batch $L + 1$). From the exclusion property of reg_L, we know that at most one propose() invocation of batch L can return *nextFast* = *true*. Thus, for batch $L + 1$ at most one process can have *fast[L+1]* = *true*. Hence, there are no two fresh invocation by different processes with *fast* = *true*. Now consider two fresh invocations by the same process p_i on reg_{L+1}.

[6] If the witness task replies nack, then obviously some other invocation has already changed the *val* of the witness task. Since a witness task stores *val* in stable storage before replying, once *val* is changed, the task can always detect that *val* has been modified, even if the host process crashes and recovers.

```
procedure ◇RegisterII()                                        {Constructor, for each process p_i}
   read_i ← 0
   write_i ← 0
   val_i ← ⊥
   v* ← ⊥
   k ← i − n
   permission ← false
procedure propose(v, fast)
   if (fast and k ≤ 0) or (k ≥ n + 1) then                     {if (fresh and fast) or (not fresh)}
      k ← k + n
   else
      k ← k + 2n              {fresh invocation which is not fast has round number greater than n}
   if k ≥ n + 1 then                          {if not(fresh and fast) then execute the read phase}
      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, false)
      else
         select the [ackREAD,k, k′, val′] with the highest k′
         if val′ ≠ ⊥ then
            v* ← val′
         else
            v* ← v
   else
      v* ← v
   s-send[WRITE,k, v*] to all processes
   wait until s-received [ackWRITE,k,*] or [nackWRITE,k] from ⌈(n+1)/2⌉ processes
   if s-received at least one [nackWRITE,k] then
      return(abort, false)
   else
      if s-received at least one [ackWRITE,k,false] then
         return(v*,false)
      else
         return(v*,true)
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}
      s-send [ackREAD,k, write_i, val_i] to p_j
task wait until s-received [WRITE,k, v] from p_j
   if write_i > k or read_i > k then
      s-send [nackWRITE,k] to p_j
   else
      if k ≤ n then write_i ← n + 1 else write_i ← k
      permission ← (val_i = ⊥)
      val_i ← v; store{write_i, val_i}
      s-send [ackWRITE,k,permission] to p_j
upon recovery do
   initialisation
   retrieve{write_i, read_i, val_i}
```

Figure 5: ◇RegisterII implementation in a crash-recovery model with stable storage

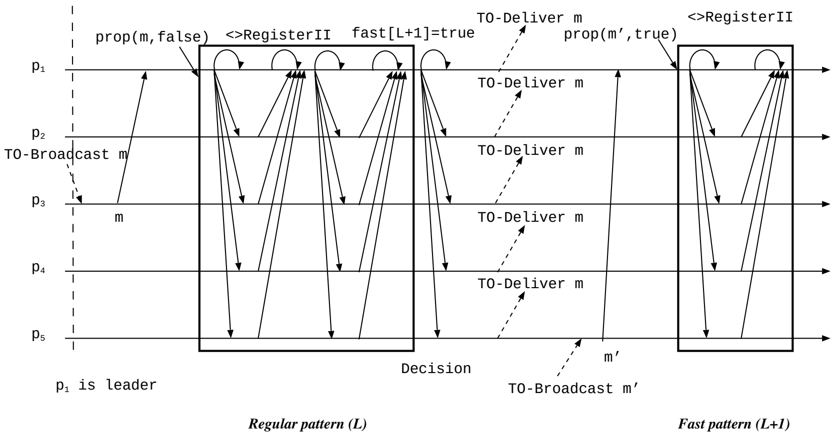


Figure 6: Communication steps for a regular followed by a fast communication pattern

Suppose that both invocations have *fast* = *true*. Since p_i invokes two fresh invocations on the same register reg_{L+1}, p_i has crashed and recovered between the invocation. From the algorithm we know that an invocation on reg_{L+1} has *fast* = *true* if the current value of *fast[L+1]* is *true*. However, at initialization (and after every recovery) *fast[L+1]* is set to *false*. Thus, at p_i, twice *fast[L+1]* was set to *true*. Variable *fast[L+1]* at p_i is set to *true* if and only if an invocation on reg_L by p_i returns *nextFast* = *true*. Hence, two distinct invocations by p_i on reg_L returned *true*, a violation of exclusion property of reg_L. Thus, there are no two fresh invocations on the same register with *fast* = *true*.

Recall that in our implementation of $\diamond$RegisterII, every fresh invocation with *fast* = *true* completes in two communication steps and a stable storage access. Thus, to obtain the optimization in stable periods, fresh invocations by the leader must have *fast* = *true*. Recall that in a stable period, only one correct process elects itself as a leader. Thus, for any batch L in stable period, only one process p_l invokes propose() on reg_L. From the exclusion property of reg_L, it follows that this invocation returns *nextFast* = *true*, and hence, p_l sets *fast[L+1]* = *true*. Thus, the invocation by p_l on reg_{L+1} has *fast* = *true*.

If a process starts the Converge task for different batches in parallel, the leader p_l may start the first invocation on reg_{L+1} before it completes the invocation on reg_L. In this case, *fast[L+1]* may be set to *true* only after propose() is invoked on reg_{L+1}, and thus, the invocation on reg_{L+1} may have *fast* = *false*. To avoid this scenario, a process starts Converge task for batch $L+1$ only when Converge task for batch L has been completed.

7 Acknowledgment

We are very grateful to Petr Kouznetsov for improving the correctness argument of the $\diamond$Register implementation in the crash-recovery model without stable storage.

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.

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

Figure 7: The Fast Paxos algorithm

[2] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *Distributed Computing Column of ACM SIGACT News*, 34(1):47–67, March 2003.

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

[4] E. Gafni and L. Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, March 2003.

[5] 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.

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

[7] L. Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58, December 2001.

[8] 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.