The Paxos Register

Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi
The University of Texas at Austin
Department of Computer Sciences
1 University Station #C0500
Austin, TX 78712
{harry, aclement, anand, lorenzo}@cs.utexas.edu

Contact Author: Harry Li (harry@cs.utexas.edu)
Contact Address: Dept. of Computer Sciences
The University of Texas at Austin
1 University Station #C0500
Austin, TX 78712

The Paxos Register

Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi
The University of Texas at Austin
Department of Computer Sciences
{harry, aclement, anand, lorenzo}@cs.utexas.edu

Abstract

We introduce the Paxos register to simplify and unify the presentation of Paxos-style consensus protocols. We use our register to show how Lamport's Classic Paxos and Castro and Liskov's Byzantine Paxos are the same consensus protocol, but for different failure models. We also use our register to compare and contrast Byzantine Paxos with Martin and Alvisi's Fast Byzantine Consensus. The Paxos register is a write-once register that exposes two important abstractions for reaching consensus: (i) read and write operations that capture how processes in Paxos protocols propose and decide values and (ii) tokens that capture how these protocols guarantee agreement despite partial failures. We encapsulate the differences of several Paxos-style protocols in the implementation details of these abstractions.

Keywords: Consensus, Paxos, Fast Byzantine Consensus, Byzantine Paxos, PBFT, Agreement

1 Introduction

We introduce the *Paxos register* to simplify and unify the presentation of Paxos-style consensus protocols. We frame Lamport's Classic Paxos [14] and Castro and Liskov's Byzantine Paxos [5] as implementations of this register and show that they are the same protocol, but for different failure models. We then use the Paxos register to highlight the similarities and differences between Byzantine Paxos and Martin and Alvisi's Fast Byzantine (FaB) Paxos algorithm [18]. Finally, we show how insights stemming from the Paxos register have led to the first deterministic asynchronous consensus algorithm that tolerates computationally unbound Byzantine adversaries by using secrets instead of digital signatures.

Since deterministically solving consensus in an asynchronous system with failures is impossible [10], Classic Paxos gives us best alternative for crash failures. It guarantees the safety properties of consensus and relies on synchrony only for liveness [9]. Byzantine Paxos and FaB Paxos provide the same guarantees as Classic Paxos for asynchronous systems but for Byzantine faults.

At a high-level, these consensus protocols are intuitively similar. They guarantee safety and liveness as explained above. In addition, all three protocols use leaders to coordinate actions among quorums [7, 17, 19] of processes. And yet, while these protocols share part of their names, the extent of the similarities between Lamport's protocol and Castro and Liskov's is unclear.

It is difficult to characterize these similarities for three main reasons. First, Paxos-style algorithms are non-trivial protocols that use asynchronous and unreliable communication to obtain quorums. The subtleties of the corner cases in such a setting can quickly become overwhelming[1]. Second, Byzantine Paxos and FaB Paxos are more complex than Classic Paxos because the former ones assume a weaker failure model. This additional complexity obfuscates the underlying similarities between these three protocols.

The Paxos register helps overcome these difficulties in two ways. The first way is that using a register hides the details of asynchronous communication and quorum operations. The second is that the register introduces *tokens*, which guard writes and define both which values are safe to write and when.

Processes issue read and write operations to this shared register. With a correct unique leader, it is easy to see how to guarantee agreement; only the leader writes to the Paxos register and the leader writes only one value to the register. Non-leader processes wait until they read a non-$\perp$ value. Guaranteeing agreement becomes harder if the leader fails. Processes need to elect a new leader who then should be careful to only write values consistent with previous writes.

We define the Paxos register's consistency semantics such that for a new leader, reads only return values consistent with the previous leader's writes. A newly elected leader therefore only needs to prove that it issued the appropriate read before it writes a value.

A *token* is a proof that a leader issued a particular read. To write a value, a newly elected leader must first present an appropriate token to the register. By guarding each write with a token, we obtain a write-once register. We claim that describing Paxos-style consensus protocols as operations on a write-once register simplifies the presentation of these protocols, and thus, makes them more accessible.

Differences among Paxos-style protocols manifest themselves in the implementation of the Paxos register, not in the specification. For example, a crash-tolerant Paxos register uses plain tokens, whereas a Byzantine-tolerant Paxos register uses secure tokens to prevent foul play.

The Paxos register is the first register-based treatment of both a crash-tolerant consensus protocol and a Byzantine-tolerant consensus protocol. Although registers simplify the exposition of deterministic asynchronous consensus protocols [11], the only existing unified presentation of these protocols does not use a register [16].

[1] It is a testament to Classic Paxos's steep learning curve that, to be qualified for a research position, candidates may be required to have at least once tried to understand it by reading the original paper [22].

Prior efforts that use a register to explain consensus are limited to either benign or Byzantine failures [4, 6, 8]. The Paxos register handles both kinds of faults and provides semantics similar to the traditional notion of regular semantics [13].

We give an overview of our approach in Section 3. Section 4 specifies the Paxos register and explains how guarding each write with a token yields a write-once register. In Sections 5 through 7, we show how Classic Paxos, Byzantine Paxos, and FaB Paxos implement the Paxos register. Finally, in Section 8, we demonstrate the Paxos register's power and flexibility by sketching IT ByzPaxos [2], a novel variant of Byzantine Paxos that is information-theoretically secure.

2 Related Work

Our work is the latest in a series of papers that revisit the Paxos protocol.

De Prisco et al. introduce the Clock General Timed Automaton (Clock GTA) [20] and use it to model, verify, and analyze Classic Paxos. Using the Clock GTA, they were the first to study the performance of Classic Paxos both during failure-free executions and in the presence of failures.

Lamport's second take at Classic Paxos [15] directly and concisely explains the protocol. Our goal is to maintain that clarity and simplicity while providing a characterization that also encompasses Castro and Liskov's Byzantine Paxos.

Lampson describes Abstract Paxos [16], a version of Lamport's original protocol, and derives Classic Paxos, Byzantine Paxos, and Disk Paxos [11] from it. These derivations focus on how a process chooses an appropriate value before trying to get enough processes to accept that value, which Lampson identifies as the key problem in implementing Paxos-like protocols. We leverage the existing body of work on quorum systems to capture the complexity of this choice in a register's read operation.

Boichat et al. separate Paxos's safety and liveness requirements into the *eventual register* and *leader election* modules [4]. Guerraoui and Raynal later refined this safety-liveness separation into the Alpha and Omega abstractions, respectively [12]. Both works use this separation to gain insight into the common internal structure of several crash-tolerant Paxos protocols. Our paper is complementary to these efforts: the Paxos register abstraction tries to elucidate one of the subtlest aspects of Paxos-style protocols—how to provide agreement when the leader fails—which the eventual register and Alpha operations abstract away into a single *propose* or *Alpha* step, respectively. We believe that exposing the complexity of how to guarantee agreement is a crucial step towards understanding Paxos variants, especially those variants that tolerate Byzantine faults.

Dutta et al. focus on establishing complexity bounds for asynchronous Byzantine consensus [8]. Their

treatment contains a construct, the *WriteProof*, that is akin to our token. The Paxos register differentiates our work from theirs because our specification enables us to unify the presentation of Classic Paxos and Byzantine Paxos.

Chockler and Malkhi's *ranked register* [6] drew inspiration from Boichat et al.'s earlier work [3]. Our Paxos register is similar to their ranked register but differs in two important ways. First, the Paxos register handles crash and Byzantine failures, while the ranked register handles only crash ones. Second, the Paxos register is similar to regular consistency semantics, whereas the ranked register's specification resembles neither safe, regular, nor atomic semantics.

Shao et al. [21] also explore multi-writer regular consistency semantics. Our semantics are closest to their weakest specification, MWR1. However, ours still qualitatively differs because we base our consistency semantics on a different partial-order than what real-time defines.

3 Overview

We now describe a high-level protocol that processes execute to reach consensus. Our description uses a shared register abstraction. Processes can play any of three roles: *proposer*, *acceptor*, or *learner* [15]. Proposers propose values by writing to the shared register. Acceptors are responsible for implementing the register abstraction, and learners decide values that they observe have been written to the register. We define the consensus problem as four properties:

Validity: If a correct learner decides value v, then some proposer wrote v.
Integrity: A correct learner decides at most one value.
Agreement: No two correct learners decide different values.
Termination: All correct learners eventually decide.

If all proposers are correct, solving consensus is easy: proposers are careful to write only one value to the register, and learners decide any value that they have seen a proposer write.

Designing consensus protocols (and understanding them) is difficult because proposers can fail, possibly leading to situations in which two correct learners decide different values. Paxos protocols avoid these situations in a common way: by implementing a Paxos register. A Paxos register guards each write with a token to provide the abstraction of a write-once register.

Tokens are proofs of which values are safe to write. A token for value v proves that it is safe for a proposer to write v. Proposers acquire these tokens by reading from the register, meaning that proposers propose values by reading and then writing.

If a proposer reads a value $v \neq \perp$, then that proposer may only write v. Otherwise, that proposer may write any value. With this restriction, the Paxos register guarantees that if a learner can decide a value v written by some proposer then subsequent proposers read v from the register, leading those proposers to again write v.

If proposers can fail in Byzantine ways, then faulty proposers could forge tokens, permitting such proposers to write values different from the ones they read. With Byzantine failures, tokens need to be secure to preserve safety.

For liveness, we assume a protocol exists that eventually selects a single correct proposer long enough for that proposer to write a value. For crash failures, we can implement a simple leader election protocol. For Byzantine failures, we can allot exponentially increasing windows of time to each proposer in a round-robin fashion [5]. Both of these approaches require timing assumptions to guarantee termination.

4 Paxos Register

4.1 Paxos Register Semantics

A Paxos register stores value and timestamp pairs. For convenience, we use the syntactic convention that v is a value and ts is a timestamp. The register provides *read* and *write* operations to access the value and timestamp. The register is initialized to $\perp$, a value that cannot be written. Furthermore, each read or write has begin and end times measured by a world clock. We point out that timestamps are usually implemented as monotonically increasing values that have little relation to the world clock.

Register Operations

The Paxos register's read operation takes no parameters and returns either an error or a token. Each token encapsulates a value and timestamp pair and serves as proof that a read returned that particular pair. The write operation takes two parameters—a value and a token—and returns immediately whether the value actually gets written or not. Paxos registers further depart from traditional registers [13, 17] in the following ways.

O1 If a read returns a token, the read's timestamp is the returned token's timestamp. If a read returns with an error, the read's timestamp is undefined.

O2 A write's timestamp is the timestamp of the write's token.

O3 A write is *legal* if the token's value is $\perp$ or v, where v is the value trying to be written.

O4 A write is *visible* if it is legal and can be read.

O5 The Paxos register supports a third operation, *acknowledged*, that tracks the progress of write operations. The acknowledged operation takes no parameters and returns a set of value-timestamp pairs, each pair corresponding to a visible write.

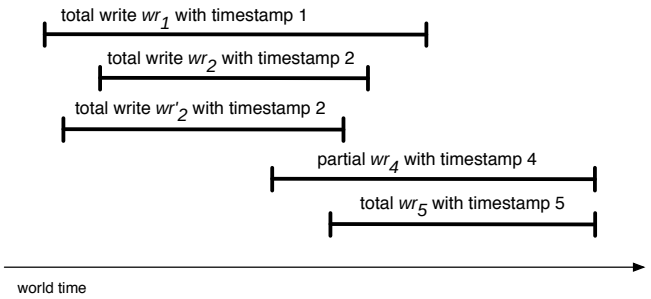

Figure 1: A sequence of disallowed Paxos register operations for two reasons—independent of the values they read or write. According to O8, wr_1 should end when wr'_2 ends and according to O7, wr_2 and wr'_2 cannot both be total.

O6 A write is *total* if the write's value-timestamp pair is in the returned set of any acknowledged operation. Otherwise, the write is *partial*.

O7 If a write is total, no other write for the same timestamp can be visible.

O8 Every write begins as partial and ends either when it becomes total or when an overlapping write with higher timestamp ends, whichever occurs first. An overlapping operation with a higher timestamp may prevent a write from becoming total.

Different protocols implement the above conditions in different ways. Figure 1 shows a sequence of operations that are disallowed by our specification.

Consistency Semantics

The semantics of a Paxos register are similar to regular consistency semantics [13]. In a register with regular semantics, a read that is not concurrent with a write returns the last written value. A read that is concurrent with a write can return the last written value or any value that is concurrently being written.

We alter this traditional definition in two ways. First, the read is only allowed to return the value of visible writes. Second, we redefine the notion of concurrency with respect to register operations.

Traditionally, two operations are concurrent if they overlap in real time. The register uses timestamps and the distinction between partial and total writes to define a different partial order:

C1 Writes are ordered by increasing timestamp.

C2 A total write precedes a read if the read returns a higher timestamp.

C3 A read precedes a write (whether partial or total) if the read's timestamp is lower than the write's.

Henceforth, we use 'overlapping' with respect to the real-time partial order and 'concurrent, previous, most recent, etc.' with respect to the above partial order definition. Figure 2 gives an example of how *our* partial order affects reads under regular semantics.

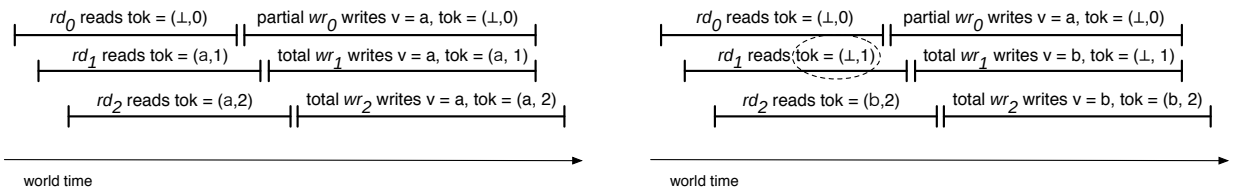

Figure 2: Two illustrations of the Paxos register's consistency semantics. Note that rd_1 is concurrent with wr_0, allowing rd_1 to read value 'a' or $\perp$ in the left and right pictures, respectively. However, rd_2 is after wr_1 in both examples because of C2. In both the left and right diagrams, wr_2 writes the already totally written value of wr_1.

4.2 Write-Once Register

By restricting the values that can be written via the tokens, a Paxos register implements a write-once register. A write-once register stores a value, initially $\perp$, that changes at most once. We define the value stored by a Paxos register to be the last totally written value. Note that reads issued before the first total write ends may return values that, strictly speaking, are never actually written to the Paxos register. These values come from concurrent partial writes.

We prove that all total writes to a Paxos register write the same value by proving the following stronger property.

Theorem 1. *If write(v, tok) is the first write that is total, then all writes with timestamp $ts' > ts$ also write v, where ts is tok's timestamp.*

Proof. Induct on the number n of writes after write(v, tok).

Base Case: There are 0 writes after write(v, tok). Trivial.

Inductive Hypothesis: The first $n \geq 0$ writes after write(v, tok) all write the same value.

1. Consider the $n + 1^{th}$ write(v', tok') after write(v, tok).
2. Let ts' be the timestamp of tok'.
3. ts' is greater than every timestamp of the first n writes after write(v, tok).
4. In general, a write for value v' and timstamp ts' can only be issued with a token whose timestamp is ts' and whose value is either $\perp$ or v'.
5. By C2, however, tok' cannot have value $\perp$ because write(v,tok) is total.
6. Now consider the read that results in a token with value v' and timestamp ts'. Because of C1-3, v' can either be the last totally written value before ts' or a value being concurrently written.
7. In either case, $v' = v$ because of the Induction Hypothesis.

□

Proposers and learners use the Paxos register to solve consensus by executing the protocol in Figure 3. Remember that acceptors implement the actual register and that `leader` eventually selects a single correct

Proposer i**'s protocol:**

let inp be the input value

repeat
  when i == leader()
    $token$:= read()
    if read did not return an error
      if $token.value == \perp$
        write(inp, tok)
      else
        write($token.value, token$)

Learner i**'s protocol:**

when acknowledged() $\neq \emptyset$
  decide v for any $(v, ts) \in$ acknowledged()

Figure 3: High-level protocol for proposers and learners.

proposer long enough for that proposer to write a value.

4.3 Discussion

The Paxos register abstracts away the details of asynchrony and quorum operations. In the next sections, we give implementations of the Paxos register over a set of processes where register operations translate to a set of messages sent over asynchronous links to a quorum of acceptors.

Separating the read and write operations, instead of combining them as in Boichat et al.'s single propose operation [4], exposes an important component of Paxos-like protocols—the tokens. Tokens serve as guards to writes. Tokens require that for a value v and timestamp ts to be written, either v and ts first have to be read or $\perp$ and ts have to be read. Because of the Paxos register's consistency semantics, such a read guarantees that the write can proceed without violating agreement.

The tokens encapsulate the most difficult part in understanding the differences between Classic Paxos and Byzantine Paxos—how to guarantee agreement when proposers fail. We use plain tokens in a crash failure model because we assume proposers do not forge tokens. However, if proposers can be Byzantine, we use secure tokens. Secure tokens prevent players from invoking writes that violate Theorem 1 and consequently agreement.

By extracting tokens out from reads and writes, it is natural to explore different ways of implementing secure tokens. In the next sections, we show how Classic Paxos, Byzantine Paxos, and FaB Paxos have very different token implementations. Then in Section 8, we show a novel way to implement tokens using secrets instead of using cryptographic primitives.

Proposer p's implementation of read and write:

$localTS := 0$

procedure read()
  $currTS := (localTS, p)$
  $localTS := localTS + 1$
  send ⟨READ, $currTS$⟩ to acceptors
  wait until received ⟨READ-ACK, $currTS, lastVisible$⟩
      from a majority of acceptors
    let v be the value among the $lastVisible$s with
        highest timestamp
    return $(v, currTS)$
  on timeout
    return error

procedure write(v,token)
  let ts be the $token$'s timestamp
  send ⟨WRITE, v, ts⟩ **to** acceptors

Acceptor a's protocol:

$highestTS$:= (-1, NULL)
$lastVisible := (\perp, -1)$

on receive ⟨READ, ts⟩ **from** proposer p
  if $(ts > highestTS)$
    $highestTS$:= ts
    send ⟨READ-ACK, $ts, lastVisible$⟩ **to** p
  endif

on receive ⟨WRITE, v, ts⟩
  if $(ts \geq highestTS)$
    $highestTS := ts$
    $lastVisible := (v, ts)$
    send ⟨WRITE-ACK, v, ts⟩ **to** learners
  endif

Learner l's implementation of acknowledged:

procedure acknowledged():
  return the set of value-timestamp pairs (v, ts)
    such that l received ⟨WRITE-ACK, v, ts⟩ **from**
    a majority of acceptors

Figure 4: Crash Paxos register.

5 Crash Paxos Register

We show how to implement the Paxos register over a set of acceptors that can fail by crashing. Proposers issue read and write operations by sending requests over asynchronous links to the acceptors. Upon receiving a request, an acceptor may send an acknowledgment. Acceptors send read acknowledgments to proposers and send write acknowledgments to learners. Learners decide a value when they receive enough write acknowledgments for that value. Later in this section, we draw the parallels between our protocol and the Classic Paxos algorithm.

5.1 Assumptions

Our implementation assumes an asynchronous distributed system in which at least one proposer and over half the acceptors are correct. Processes communicate by passing messages over unreliable links and fail by permanently crashing[2].

5.2 Implementation

Proposers and learners reach consensus, as shown in Figure 3, by invoking read, write and acknowledged operations on a Paxos register. Figure 4 shows those operations when the acceptors collectively implement the Paxos register abstraction.

[2]We can easily handle processes that crash and recover by having each process write global variables to stable storage before sending any message. While recovering, a process initializes its variables based upon the contents in stable storage.

Classic Paxos message	Crash Paxos register message
prepare request	read
prepare response	read acknowledgment
accept request	write
accept response	write acknowledgment

Table 1: How messages in Classic Paxos map to messages in a crash-tolerant Paxos register.

A proposer p reads from the register by first constructing a unique timestamp, $currTS$, and sending $\langle \texttt{READ}, currTS \rangle$ to all acceptors. An acceptor a responds to such a request if $currTS$ is higher than any timestamp a has seen. If so, a sends $\langle \texttt{READ-ACK}, currTS, lastVisible \rangle$ back to p, where $lastVisible$ identifies the visible write with highest timestamp that a has seen so far. This acknowledgment is also a promise that any future request whose timestamp is lower than $currTS$ is ignored by a.

If p obtains read acknowledgments from a majority of acceptors, p can finish reading from the register by constructing a token with timestamp equal to $currTS$ and value equal to the value of the highest timestamped visible write among the received read acknowledgments.

Proposer p writes to the register by sending $\langle \texttt{WRITE}, v, ts \rangle$ to the acceptors, where ts is the timestamp of the passed in token. An acceptor a responds if ts is at least as large as the highest timestamp a has seen. If so, a considers the write for v, ts visible and sends $\langle \texttt{WRITE-ACK}, v, ts \rangle$ to the learners.

A learner tracks the progress of writes using the acknowledged operation, which returns the value-timestamp pairs of writes that a majority of acceptors consider visible.

5.3 Classic Paxos

Our protocol to implement a crash-tolerant Paxos register is nearly identical to the Classic Paxos algorithm for a single instance of consensus. We now give a brief overview of the Classic Paxos algorithm and show how it relates to the Crash Paxos register. The reader can find the full Classic Paxos protocol in [14].

Proposers, acceptors, and learners in Classic Paxos play the same roles as they did in the Crash Paxos register. Proposers propose values to acceptors, and learners decide a proposal's value once they have learned that enough acceptors have responded to that proposal.

In Classic Paxos, proposers issue proposals in two phases. In the first phase, a proposer p sends a *prepare request* containing a unique proposal number x to all the acceptors. An acceptor responds to a prepare request only if x is higher than any proposal number the acceptor has seen. A *prepare response* contains i) the highest numbered proposal the acceptor has accepted and ii) a promise to only accept proposals whose proposal numbers

are greater than x.

If p receives responses to its prepare request from a majority of acceptors, then p enters the second phase. In the second phase, a proposer selects the value of the highest numbered proposal among the received prepare responses. If there is no such value, then p selects an arbitrary value. p then sends an *accept request* containing x and the selected value to all acceptors.

An acceptor a responds to an accept request if the contained proposal number is at least as high as any other proposal number the acceptor has seen. If a accepts an accept request, a sends an *accept response* to the learners, echoing the proposal's value and number. A learner can decide a proposal's value once it has received accept responses for that proposal from a majority of acceptors.

Table 1 relates Crash Paxos register messages to Classic Paxos messages and shows that a proposer's first and second phases correspond to a read and write, respectively. Also, proposal numbers correspond to the Paxos register's timestamps and a proposal that can be decided corresponds to a total write.

6 Byzantine Paxos Register

In this section, we show how to implement a Byzantine fault-tolerant Paxos register over a set of processes. Similar to the crash-tolerant version, register operations send requests to acceptors and acceptors respond with acknowledgements.

6.1 Assumptions

We assume an asynchronous system in which processes can fail by arbitrarily deviating from the protocol. There are n_p proposers, at least one of which is correct, and $n_a > 3f_a$ acceptors, f_a of which may fail. Failed processes cannot subvert cryptographic primitives such as digital signatures.

Processes digitally sign messages to prevent message forgery. We use the notation $\langle M \rangle_i$ to indicate a message M signed by process i. Processes discard improperly signed messages.

6.2 Implementation

There are three key differences between a Byzantine Paxos register and a Crash Paxos register. First, an acceptor implementing the Byzantine Paxos register maintains its own timestamp and only acknowledges reads and writes for the current timestamp, discarding all other messages. Furthermore, for each timestamp ts, proposer p is the leader for ts if $p \equiv ts \bmod n_p$. Second, each write involves a pre-write step to guarantee that only one write per

Proposer p's implementation of read and write:

$estTS := p$

procedure read()
  send $\langle$READ, $estTS\rangle_p$ **to** acceptors
  wait until received $\langle$READ-ACK, $estTS, lastVisible\rangle$ **from**
      a quorum of acceptors
    let v be the value among $lastVisible$s with highest timestamp
    return $(v, estTS,$ quorum of READ-ACKs)
    on timeout
      return error

when receive $\langle$TIMESTAMP-CHANGE, $ts\rangle$ **from** quorum of acceptors
  if $ts > estTS$ AND $p \equiv ts \bmod n_p$
    $estTS := ts$
  endif

procedure write(v,token)
  let ts be the $token$'s timestamp
  send $\langle$PRE-WRITE, $v, ts, token\rangle_p$ **to** acceptors

Learner l's implementation of acknowledged:

procedure acknowledged()
  return the set of value-timestamp pairs (v, ts)
      such that l received $\langle$WRITE-ACK, $v, ts\rangle$ **from**
      a quorum of processes

Acceptor a's protocol:

$currTS$:= 0
$lastVisible$:= ($\perp$, -1, NULL)

on receive $\langle$READ, $ts\rangle_p$
  if ($ts = currTS$ AND p is the leader for ts)
    send $\langle$READ-ACK, $currTS, lastVisible\rangle_a$ **to** p
  endif

on receive $\langle$PRE-WRITE, $v, ts, token\rangle_p$
  if (($p = ts \bmod n_p$) AND
      ($ts \geq currTS$) AND
      (have not sent WRITE for ts) AND
      ($token$ shows this write is legal))
    if $ts > currTS$
      reset timeout
      $currTS := ts$
    endif
    send $\langle$WRITE, $v, ts\rangle_a$ **to** acceptors
  endif

when receive $\langle$WRITE, $v, ts\rangle$ **from** quorum of acceptors
  if ($ts \geq currTS$)
    if $ts > currTS$
      reset timeout
      $currTS := ts$
    endif
    $lastVisible := (v, ts,$ quorum of WRITEs)
    send $\langle$WRITE-ACK, $v, ts\rangle_a$ to learners
  endif

at time $timeoutVal$
  $currTS$:= $currTS + 1$
  $timeoutVal := 2 \times timeoutVal$
  $p := currTS \bmod n_p$
  send $\langle$TIMESTAMP-CHANGE, $currTS\rangle_a$ **to** proposer p

Figure 5: Byzantine Paxos register.

timestamp is visible. Third, $n_a - f_a$ acceptors constitute a quorum as compared to a simple majority in the Crash Paxos register. Figure 5 defines the protocols that proposers, acceptors, and learners follow.

A proposer p reads from the register by sending $\langle \texttt{READ}, estTS\rangle_p$ to all acceptors, where $estTS$ is p's estimate of the timestamp for a quorum of acceptors. An acceptor a responds if $estTS$ matches a's current timestamp and p is the leader for $estTS$. If so, then a responds with $\langle \texttt{READ-ACK}, estTS, lastVisible\rangle_a$, where $lastVisible$ contains the value and timestamp of the last visible write that a has seen. $lastVisible$ also contains a set of signed messages from a quorum of acceptors proving that such a write actually was visible and not just something that a concocted.

If p obtains and verifies read acknowledgments from a quorum of acceptors, then p can finish reading by constructing a token with timestamp $estTS$ and value equal to the value of the highest timestamped visible write among the $lastVisible$s. p also appends the quorum of read acknowledgments to the token as proof that p did not fabricate the result of the read.

Proposer p writes to the register by sending $\langle \texttt{PRE-WRITE}, v, ts, token \rangle_p$, where ts is the $token$'s timestamp and v is a value that can be legally written using $token$. An acceptor a accepts the pre-write if *i)* p is the leader for ts, *ii)* ts is at least as high as a's current timestamp, *iii)* a has not accepted another pre-write for ts, and *iv)* $token$ shows that this write is legal. If a accepts a pre-write, then a sends $\langle \texttt{WRITE}, v, ts \rangle_a$ to all acceptors and changes its current timestamp to ts if ts is higher.

A write is visible once a quorum of acceptors send write messages in response to it, meaning that two writes will never both be visible if they are for the same timestamp but different values. An acceptor a sends $\langle \texttt{WRITE-ACK}, v, ts \rangle_a$ to the learners if a observes that a write for v and ts is visible, where ts is greater than a's current timestamp.

As in the Crash Paxos register, each learner tracks the progress of writes using acknowledged(). The acknowledged operation returns the value-timestamp pairs of writes that a quorum of acceptors consider visible.

In case a proposer has failed, acceptors periodically increment their timestamps to give another proposer an opportunity to write a value. When an acceptor a changes its timestamp to ts, a sends $\langle \texttt{TIMESTAMP-CHANGE}, ts \rangle_a$ to the proposer p that leads ts. p updates its estimate of the current timestamp when it receives a quorum of such timestamp change messages.

For clarity, we presented an unoptimized Byzantine Paxos register. We now describe two simple optimizations that reduce the number of messages that need to be sent in some situations. First, the write for timestamp 0 does not require a token, meaning that a decision can be reached in three message delays when there are no failures and all messages are delivered on time, a situation that we expect to be the norm. Second, we can eliminate the read message by combining read acknowledgments with timestamp change messages. When an acceptor a increments its timestamp to ts, a can send $\langle \texttt{TIMESTAMP-CHANGE}, ts, lastVisible \rangle_a$ to the proposer p who leads ts. Therefore when p reads, p can immediately construct a token using the timestamp change messages instead of sending a read message and waiting for acknowledgments.

6.3 Byzantine Paxos

Castro and Liskov presented Byzantine Paxos as part of the larger Practical Byzantine Fault-Tolerance (PBFT) protocol [5]. PBFT is a Byzantine tolerant state-machine replication algorithm. It is hard to see the connection between PBFT and Byzantine Paxos because PBFT handles aspects of state-machine replication (like checkpoints and garbage collection) that quickly increases the protocol's complexity. We strip PBFT down to the elements necessary to achieve consensus and present this version as Byzantine Paxos.

Processes in Byzantine Paxos have unique ids from the set $\{0, \ldots, n-1\}$, where n is the number of processes. Each process maintains its *view*, which is a monotonically increasing natural number initialized to 0. The *primary*

Byzantine Paxos message	Byzantine Paxos register message
pre-prepare	pre-write (without token)
prepare	write
commit	write acknowledgment
view change	timestamp change + read acknowledgment
new view	pre-write (with token)
(optimized out)	read

Table 2: How messages in Byzantine Paxos map to messages in an optimized Byzantine Paxos register.

for view v is the process with id v mod n. A quorum in Byzantine Paxos consists of $n - f$ processes, where $f \leq \lfloor \frac{n-1}{3} \rfloor$ is the maximum number of processes that can fail.

Using the Paxos register terminology, views correspond to timestamps, primaries correspond to leaders, and each process is a proposer, acceptor, and learner.

In normal-case operation (without primary failures), Byzantine Paxos consists of three phases—pre-prepare, prepare, and commit—each of which contacts a quorum of processes.

In the pre-prepare phase, the primary p issues $\langle \texttt{PRE-PREPARE}, val, vue \rangle_p$, where vue is the current view and val is the value that p proposes. A process accepts a pre-prepare message provided the sender is the primary of vue, the process's current view is vue, and the process has not already accepted a pre-prepare message for vue. When a process i accepts $\langle \texttt{PRE-PREPARE}, val, vue \rangle_p$, it broadcasts a $\langle \texttt{PREPARE}, val, vue \rangle_i$ and enters the prepare phase.

In the prepare phase, a process waits and collects a quorum of prepare messages that have matching values and views. Once a process i has such a quorum of messages that match its current view, i considers that value to have *prepared* in view vue, broadcasts $\langle \texttt{COMMIT}, val, vue \rangle_i$ and enters the commit phase.

In the commit phase, a process waits for a quorum of commit messages that have matching values and views. Once a process i has such a quorum, i considers that value and view to have been *committed* and decides that value.

A value val that has prepared in view vue is analogous to a visible write for value val and timestamp vue. A similar analogy exists between a value and view that has committed and a write becoming total. Table 2 gives the mapping from messages in Byzantine Paxos to messages in the optimized Byzantine Paxos register.

If the primary is suspected to have failed, processes elect a new primary by incrementing their views. When a process increments its view to vue, it sends $\langle \texttt{VIEW-CHANGE}, vue, \mathcal{P} \rangle_i$ to the new primary, where $\mathcal{P}$ is the quo-

rum of prepare messages vouching for the most recent value and view to have prepared at i. When the primary p for view vue receives a quorum of valid view-change messages, p broadcasts $\langle \texttt{NEW-VIEW}, vue, \mathcal{V}, \langle \texttt{PRE-PREPARE}, vue, val \rangle_p$ where $\mathcal{V}$ is the quorum of valid view-change messages and val is the value among the prepare messages of $\mathcal{V}$ with highest view number. If all the $\mathcal{P}$ in the view-change messages are empty, then val can be any value.

When a process receives a new-view message, it verifies the contents including the $\mathcal{V}$ field and the appropriate selection of the value in the contained pre-prepare message. If the process can verify the contents, then it acts as if it received the pre-prepare message and continues executing the protocol, as before, but in the new view.

The $\mathcal{P}$ field of a view-change message corresponds to the last visible write that an acceptor in a Byzantine Paxos register has seen. Similarly, the $\mathcal{V}$ field of a new-view message is conceptually a token proving that the pre-prepare message in the new-view message is legal.

7 Fast Byzantine Paxos Register

We now describe a faster version of the Byzantine Paxos Register that in the failure-free synchronous case allows learners to decide after two message delays instead of three. This added speed comes at the cost of additional acceptors to guarantee safety.

7.1 Assumptions

The FaB Paxos register uses the same system assumptions as the Byzantine Paxos register (Section 6.1), except that the former requires $n_p > 3f_p$ proposers and $n_a > 5f_a$ acceptors, at most f_p proposers and f_a acceptors may fail. Also, the FaB Paxos register only uses digital signatures in reads, relying on authenticated channels for all other communications.

7.2 Implementation

There are four important differences between the FaB Paxos register and the Byzantine Paxos register. First, there is no pre-write step. Second, the FaB register's read operation differs significantly in that it does not select the value of the highest timestamped visible write. Third, each acceptor tracks the value for the last legal write that it has accepted, instead of tracking visible writes that it has seen. Fourth, proposers elect one another to move the timestamp past faulty proposers. A newly elected proposer is responsible for advancing the timestamps maintained by acceptors. Figure 6 gives the protocol that proposers, acceptors, and learners follow.

A proposer p reads from the register by sending $\langle \texttt{READ}, estTS, tsProof \rangle$ to all acceptors, where $estTS$ is p's estimate of the current timestamp, and $tsProof$ is a set of messages proving that enough proposers have

Proposer p's implementation of read and write:

$estTS := p$
$myTS := 0$
$tsProof := NULL$

procedure read()
  send ⟨READ, $estTS, tsProof$⟩ **to** acceptors
  wait until received ⟨READ-ACK, $estTS, lastLegal$⟩ **from**
      $n_a - f_a$ acceptors
    let v be a value among $lastLegal$s that appears
      a majority of times
    return (v, $estTS$, the $n_a - f_a$ READ-ACKs)
    on timeout
      return error

at time $timeoutVal$
  $myTS := myTS + 1$
  $timeoutVal := 2 \times timeoutVal$
  $q := myTS \mod n_p$
  send ⟨TIMESTAMP-CHANGE, $myTS$⟩$_a$ **to** proposer q

when receive ⟨TIMESTAMP-CHANGE, ts⟩ **from** $n_p - f_p$ proposers
  if $ts > estTS$ AND $p \equiv ts \mod n_p$
    $estTS := ts$
    $tsProof := n_p - f_p$ TIMESTAMP-CHANGE messages
  endif

procedure write(v,token)
  let ts be the $token$'s timestamp
  send ⟨WRITE, $v, ts, token$⟩ **to** acceptors

Acceptor a's protocol:

$highestTS := 0$
$lastLegal := \perp$

on receive ⟨READ, $ts, tsProof$⟩ **from** p
  if (($ts > highestTS$) AND
      (p is the leader for ts) AND
      (tsProof is valid for ts))
    $highestTS := ts$
    send ⟨READ-ACK, $currTS, lastLegal$⟩$_a$ **to** p
  endif

on receive ⟨WRITE, $v, ts, token$⟩ **from** p
  if (($p = ts \mod n_p$) AND
      ($ts \geq highestTS$) AND
      (have not sent WRITE-ACK for ts) AND
      ($token$ shows this write is legal))
    $highestTS := ts$
    $lastLegal := v$
    send ⟨WRITE-ACK, v, ts⟩ **to** learners
  endif

Learner l's implementation of acknowledged:

procedure acknowledged()
  return the set of value-timestamp pairs (v, ts)
      such that l received ⟨WRITE-ACK, v, ts⟩ **from**
      $\lceil \frac{n_a+3f_a+1}{2} \rceil$ acceptors

Figure 6: Fast Byzantine Paxos register.

advanced their timestamps to $estTS$.

An acceptor a responds if $estTS$ is higher than any timestamp a has seen, p is the leader for $estTS$, and $tsProof$ shows that enough proposers have advanced their timestamps to $estTS$. If so, then a responds with the signed message $\langle \texttt{READ-ACK}, estTS, lastLegal \rangle_a$ where $lastLegal$ contains the value of the last legal write that a has accepted. Note that $lastLegal$ is not a proof that some write in the past was actually legal. Rather, it is just a's testimony.

If p obtains and verifies read acknowledgments from $n_a - f_a$ acceptors, then p can finish reading by constructing a token with timestamp $estTS$ and value equal to the value that appears a majority of times among the $lastLegal$s. If no value appears a majority of times, then the read's value is $\perp$. As before, p appends the quorum of read acknowledgments to the token to prove that p did not fabricate the read.

Proposer p writes to the register by sending $\langle \texttt{WRITE}, v, ts, token \rangle$, where ts is the $token$'s timestamp and v is a value that can be legally written using $token$. An acceptor a accepts the write if *i)* p is the leader for ts, *ii)* ts is at least as high as the highest timestamp a has seen, *iii)* a has not accepted another write for ts, and *iv)* $token$ shows that this write is legal. If a accepts a write, then a sends $\langle \texttt{WRITE-ACK}, v, ts \rangle$ to all learners.

A learner's acknowledged operation returns the value-timestamp pair of any write that $\lceil \frac{n_a+3f_a+1}{2} \rceil$ acceptors

acknowledge. This number of acceptors is necessary and sufficient to guarantee that the result of subsequent reads return the written value.

In contrast to the Byzantine Paxos register, proposers in the FaB Paxos register are responsible for advancing the timestamp. Periodically, they increment their timestamps to give other proposers an opportunity to write. When a proposer p increments its timestamp to ts, p sends the signed message $\langle \texttt{TIMESTAMP-CHANGE}, ts \rangle_p$ to the proposer with id ts mod n_p. Proposers collect such messages to prove to acceptors that the timestamp has advanced.

Like before, the FaB Paxos register is unoptimized and we can make it more efficient in the common case by specifying that any write for timestamp 0 does not require a token. Further optimizations can be found in [18].

An advantage of using the Paxos register abstraction to describe FaB Paxos is that we can intuitively derive the above cryptic $\lceil \frac{n_a+3f_a+1}{2} \rceil$ value by using C2 of the consistency semantics. We use C2 to work backwards from how we defined the read operation. Remember that C2 dictates that a read is after a total write in the partial order if the write's timestamp is lower than the read's. To guarantee C2, a total write's value should be the majority of $lastLegal$ values in any subsequent read's $n_a - f_a$ acknowledgments. Out of $n_a - f_a$ acknowledgments, $\lceil \frac{n_a-f_a+1}{2} \rceil$ is the smallest majority, and after accounting for those acceptors that did not respond plus those that are Byzantine, a total write needs to be accepted by $\lceil \frac{n_a-f_a+1}{2} \rceil + 2f_a$ acceptors to guarantee C2. $\lceil \frac{n_a-f_a+1}{2} \rceil + 2f_a$ simplifies to $\lceil \frac{n_a+3f_a+1}{2} \rceil$.

7.3 Fast Byzantine Paxos

We now describe Martin and Alvisi's FaB protocol [18]. In FaB, each process is a proposer, acceptor, or learner. The FaB protocol uses $n_p > 3f_p$ proposers, $n_a > 5f_a$ acceptors, and $n_l > 3f_l$ learners, where f_p, f_a, f_l are the maximum number of proposers, acceptors, and learners that fail, respectively. However, we observe that a minimum number of learners is unnecessary to solve consensus and so elide those details in this presentation of FaB.

A proposer p proposes a value by first querying the acceptors and then issuing a proposal. Both the query and the proposal contain a proposal number n such that p is the leader for n, i.e. $p \equiv n$ mod n_p. p performs a query by sending $\langle \texttt{QUERY}, n, proof \rangle$ to the acceptors.

An acceptor a accepts the query if n is greater than the proposal number of any message that a has accepted, p equals n mod n_p, and $proof$ shows that enough proposers believe it is time for p to be the leader of n. The FaB protocol leaves the implementation of $proof$ unspecified. If a accepts the query, then a responds to p with $\langle \texttt{REPLY}, val, n \rangle_a$, where val is the value of the last proposal that a accepted.

If p receives $n_a - f_a$ replies to its query, then p can finish the query by constructing a *progress certificate*

Fast Byzantine Paxos message	Fast Byzantine Paxos register message
query	read
reply	read acknowledgment
propose	write
accepted	write acknowledgment

Table 3: How messages in Fast Byzantine Paxos map to messages in an Fast Byzantine Paxos register.

from the signed replies. A progress certificate vouches for a value v and proposal number n if all the contained replies are for n, and v appears a majority number of times among the replies. If no value appears a majority number of times, then the progress certificate vouches for every value.

A query is identical to the Paxos register's read operation: proposal numbers are analogous to timestamps, and progress certificates implement tokens. A progress certificate that vouches for only one value v corresponds to a token with value v. A progress certificate that vouches for multiple values corresponds to a token with value $\perp$.

After constructing a progress certificate PC, p issues a proposal for value val by sending $\langle \texttt{PROPOSE}, val, n, PC \rangle$ to all acceptors. An acceptor a accepts this proposal if n is at least as large as any message that a has accepted, p is the leader for n, a has not accepted any other proposal for n, and PC vouches for val and n. If a accepts this proposal, then a sends $\langle \texttt{ACCEPTED}, val, n \rangle$ to all learners

A learner decides a value v if it receives $\lceil \frac{n_a+3f_a+1}{2} \rceil$ accepted messages for v with the same proposal number.

Issuing a proposal for val and n is analogous to writing a value val for timestamp n to a Paxos register. Proposals that are later queried are essentially visible writes, and proposals that result in a learner deciding corresponds to total writes. Interesingly, the FaB protocol highlights a nuance of the Paxos register specification: that multiple writes for the same timestamp can be visible. The FaB Paxos register, in Figure 6, allows this situation as well. However, the register implementation still guarantees that a total write guarantees means that no other write for the same timestamp is or ever will be visible.

8 Information-Theoretically Secure Byzantine Paxos

Current deterministic asynchronous consensus algorithms rely on cryptographic primitives, i.e. digital signatures, to thwart Byzantine adversaries and guarantee safety. Given this, a natural question to ask is whether a version of Paxos is possible for computationally unbound Byzantine adversaries. Using the Paxos register, we now sketch an information-theoreticaly secure version of Byzantine Paxos: IT ByzPaxos. We highlight the key ideas below

and describe IT ByzPaxos fully in a technical report [2].

We draw two insights from the Paxos register to build IT ByzPaxos. First, if a write for value v and timestamp ts is total then a quorum of acceptors can prove that some proposer visibly wrote v, ts. And second, each token encapsulates a proof. A token for value v and timestamp ts proves two things: *i)* that some proposer visibly wrote v for timestamp $ts' < ts$ and *ii)* that every write for timestamp ts'' is not nor ever will be total, where $ts' < ts'' < ts$. A token for value $\perp$ and timestamp ts proves something slightly different: that every write for $ts' < ts$ is not nor ever will be total.

In IT ByzPaxos, we use secret sharing techniques to implement the above insights. Conceptually, we assign a secret S_{ts}^{v} for each value v that can be proposed and for each possible timestamp ts. We also assign a secret $S_{ts}^{\perp}$ for each timestamp ts . An honest dealer keeps these secrets hidden, but divides each secret into shares and distributes the shares to processes such that the following hold:

- A process i can reconstruct S_{ts}^{v} if and only if i observes a visible write for value v and timestamp ts.
- If a process can reconstruct $S_{ts}^{\perp}$ then any write for ts is not nor ever will be total.

Using the above properties, IT ByzPaxos implements tokens as sequences of secrets. Remember that a token for v, ts proves that a proposer visibly wrote v for $ts' < ts$ and that every write between ts' and ts is not nor ever will be total. A proposer assembles such a proof by revealing the secret $S_{ts'}^{v}$ and each secret $S_{ts''}^{\perp}$, where $ts' < ts'' < ts$. To implement a token for $\perp, ts$, a proposer reveals each secret $S_{ts'}^{\perp}$ where $ts' < ts$. In [2], we discuss how to divide secrets and distribute shares with or without an honest dealer.

IT ByzPaxos demonstrates the power and flexibility of the Paxos register abstraction. By isolating the properties common to several Paxos protocols, we found novel ways to implement the same structure as existing protocols but for computationally unbound adversaries.

9 Conclusion

The Paxos register abstraction provides a unified framework for simpler presentations of deterministic asynchronous consensus protocols like Classic Paxos and Byzantine Paxos. This abstraction clarifies the similarities between these protocols, while hiding protocol-specific details. We believe the Paxos register can express other deterministic asynchronous consensus protocols, like Disk Paxos [11] and Byzantine Disk Paxos [1], though this remains for future work. Moreover, the Paxos register has led to a novel Byzantine Paxos variant, IT ByzPaxos, that is secure even against a computationally unbound adversary. We suspect that analyzing the Paxos register further will yield additional interesting Paxos variants.

References

[1] I. Abraham, G. V. Chockler, I. Keidar, and D. Malkhi. Byzantine disk Paxos: optimal resilience with byzantine shared memory. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing*, pages 226–235. ACM Press, 2004.

[2] A. S. Aiyer, L. Alvisi, A. Clement, and H. Li. Information-Theoretially Secure Byzantine Paxos. Technical Report TR-07-21, The University of Texas at Austin, April 2007.

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

[4] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[6] G. Chockler and D. Malkhi. Active disk Paxos with infinitely many processes. In *Proceedings of the 21st Annual Symposium on Principles of distributed computing*, pages 78–87, New York, NY, USA, 2002. ACM Press.

[7] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.

[8] P. Dutta, R. Guerraoui, and M. Vukolic. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, EPFL/IC, February 2005.

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

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

[11] E. Gafni and L. Lamport. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 330–344, 2000.

[12] R. Guerraoui and M. Raynal. The Alpha and Omega of asynchronous Consensus. Technical Report PI-1676, IRISA, January 2005.

[13] L. Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[14] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

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

[16] B. Lampson. *The ABCDs of Paxos*. Presented at 20th Annual ACM Symposium on Principles of Distributed Computing, 2001.

[17] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 569–578, New York, NY, USA, 1997. ACM Press.

[18] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2005.

[19] D. Peleg and A. Wool. The availability of quorum systems. Technical report, Jerusalem, Israel, 1993.

[20] R. D. Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 111–125, 1997.

[21] C. Shao, E. Pierce, and J. Welch. Multi-writer consistency conditions for shared memory objects. In F. E. Fich, editor, *Distributed algorithms*, volume 2848/2003 of *Lecture Notes in Computer Science*, pages 106–120, Oct 2003.

[22] W. Vogels. Job openings in my group. http://weblogs.cs.cornell.edu/allthingsdistributed/ archives/000538.html.