The Game of Paxos

Technical Report TR-05-24

Harry C. Li, Lorenzo Alvisi, and Allen Clement

Abstract

We describe two abstractions that show how Lamport's Paxos and Castro and Liskov's PBFT are essentially the same consensus protocol, but for different failure models. The first abstraction is a regular register that captures how processes in both protocols propose and decide values. The second abstraction is tokens that capture how these protocols guarantee agreement despite partial failures. Together, the register and tokens provide the abstraction of a write-once regular register, which we claim is an intuitive way to conceptualize Paxos and PBFT. We also point out how details specific to Paxos and PBFT manifest themselves in the implementation of our abstractions.

1 Introduction

> *You find a group of people frantically engaged around a circular table. Intrigued, you edge closer to find that seat belts bind each person into his or her chair. Rather than struggle with their belts, they busily press flashing buttons on the table. You see the glint of a red token in a lady's hand, but she quickly inserts the token into a slot and continues pressing buttons. Next, a green sparkle catches your eye and you turn your head in its direction just in time to see a man holding a green token unlock his belt. Your curiosity finally overcomes your caution and you approach the table...*

The description above captures moments from the Game of Paxos. We use this game to show that Lamport's Paxos [7] and Castro and Liskov's PBFT [2] are the same protocol, but for different failure models.

Since solving consensus in an asynchronous system with failures is impossible [4], Paxos gives us the next best thing for crash failures. It guarantees the safety properties of consensus and relies on synchrony only for liveness. PBFT is a state-machine replication protocol for asynchronous systems with Byzantine faults. It demonstrates that Byzantine fault-tolerance can be made practical.

At a high-level, these protocols are intuitively similar. They both rely on synchrony only for liveness. In addition, both protocols use leaders to coordinate actions among quorums [3, 10, 11] of processes. While some refer to PBFT as Byzantine Paxos [9], the extent of the similarities between Lamport's protocol and Castro and Liskov's is not obvious.

It is difficult to characterize these similarities for two main reasons. First, Paxos and PBFT are non-trivial protocols that use message passing over asynchronous channels to obtain quorums. The subtleties of the corner cases in such a setting can quickly become overwhelming[1]. Second, the most elusive aspect in these two protocols is in how each guarantees agreement despite leader failures. We provide two intuitive abstractions that carve Paxos and PBFT into functionally identical parts that help overcome the above difficulties.

Our first abstraction is a register that hides the details of quorum operations. This register has regular consistency semantics [6] with respect to a partial order that we define. Processes issue read and write operations to this shared register. With a single correct leader, it is easy to see how to guarantee agreement; only the leader writes to the register and the leader writes only one value to the register. Non-leader processes wait until they read a non-$\perp$ value, and agree on the value read. Guaranteeing agreement becomes

[1] It is a testament to 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 Paxos by reading the original paper. [13]

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 partial ordering of register operations 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.

Our second abstraction is *tokens* that encapsulate proofs 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 requiring these tokens as guards to every write, we get a write-once regular register, which we claim is an intuitive way to conceptualize Paxos and PBFT.

Details specific to Paxos or PBFT manifest themselves in the implementation of these abstractions, not their specifications. Under benign failures, we use a crash-tolerant regular register and issue plaintext tokens. Under Byzantine failures, we use a Byzantine-tolerant regular register and issue secure tokens to prevent foul play.

The Game of Paxos lets us intuitively explain our abstractions. We explain the Game in Section 3. Section 4 defines our register's semantics and our tokens. Finally, in Sections 5 and 6, we show how Paxos and PBFT are specialized implementations of the register and the tokens.

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) [12] and use it to model, verify, and analyze Paxos. Using the Clock GTA, they analyze the performance of the protocol during periods of failures and also during more 'nice' times.

Lamport's own second take at Paxos [8] directly and concisely explains the protocol. Our goal is to maintain that clarity and simplicity while providing a characterization that encompasses both Paxos and Castro and Liskov's PBFT.

Lampson describes Abstract Paxos [9], a version of Lamport's original protocol, and derives Classic Paxos, Byzantine Paxos, and Disk Paxos [5] from it. In each derivation, he shows how a process chooses an appropriate value before trying to get enough processes to accept that value. Lampson highlights this choosing step as the key problem in implementing Paxos-like protocols and uses his formalism to explore how three Paxos variants accomplish it. We leverage the existing body of work on quorum systems so that the complexity in the choosing step is encapsulated in a regular register's read operation.

Boichat et al. propose a register abstraction to explain Paxos [1]. They separate Paxos's safety and liveness requirements into eventual register and leader election abstractions. From these abstractions, they construct Paxos variants such as Disk Paxos. They do not, however, address the Byzantine case. Our register abstraction is similar to their register, but departs in an important way; our register exposes two operations, read and write, whereas theirs exposes only a single *propose* operation. We believe that exposing the read and write operations is crucial to help reasoning about Paxos and PBFT.

3 The Game of Paxos

The Game of Paxos captures the protocol that correct processes execute in order to reach consensus.

A Paxos table is a circular table with a register embedded at its center. Upon sitting down, each player finds that a seat belt binds him to his chair. The only way to unlock the belt buckle is for the player to know the register's final value. And the only way to know the final value is to access the register via an interface located at each seat.

Each interface consists of two buttons, a dial that can be set to arbitrary values, a token slot, and an inset metal tray. One button is labeled 'read' and the other is labeled 'write.' The read button continually blinks, whereas the write button is initially dark. Each interface also has a tamper-proof seal on it so that the player can trust the results it gets back from the interface. At the center of the interface is an instruction label:

To learn the register's final value, follow the steps to play the Game of Paxos below.

1. Push the *read* button and examine the token that drops into the tray.
2. If the token is green, the buckle will unlock because the final value of the register is stamped on the token.
3. If the token is red and is stamped with a value, place the token in the slot, set the dial to the same value, and push the now blinking *write* button (it will turn back off). Go back to step 1.
4. If the token is red and blank, place the token in the slot, set the dial to any value, and push the blinking *write* button. Go back to step 1.

Tokens serve as proof that a read returned a particular value. Inserting a red token into the slot enables a player to issue a write. Green tokens are indicators that a value has been written into the register.

In order for this game to help players reach consensus, all green tokens must have the same value stamped on them. A user manual under each seat shows how the interface, tokens, and register guarantee this property. We open the manual in the next section.

4 The User Manual

This manual contains important information regarding the safe operation of your Paxos table. We urge you to read it carefully and become familiar with its contents. In so doing, you will understand how your Paxos table guarantees that all green tokens are stamped with the same value.

4.1 Chapter 1: Features & Safety

Your Paxos table comes with a register embedded at its center. You can access the register via an interface at each seat. It should look similar to the picture in Figure 1. Your interface guarantees the following:

Validity: If a token is stamped with value v, then some player pressed the write button with the dial set to v.

Integrity: A player following the rules obtains at most one green token.

Agreement: All green tokens are stamped with the same value.

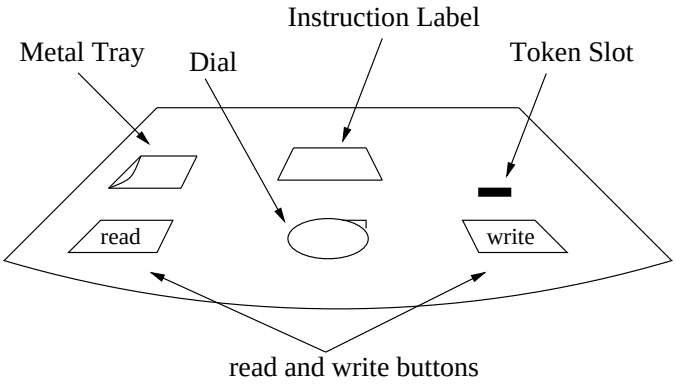

Figure 1: Example interface at each seat.

Warning! Check your table's operating assumptions. If you violate these assumptions, your Paxos table may behave unexpectedly.

4.2 Chapter 2: Register Semantics

Your embedded register stores value and timestamp pairs[2]. It provides two operations to access it: read and write. The register's initial value is $\perp$, which no one can write. Furthermore, each read or write has begin and end times measured by a world clock.

Caution. In many register implementations, the timestamp is a monotonically increasing natural number, which has no relation to the world clock.

Register Operations

Your register's read operation takes no parameters, but returns a value and timestamp pair. Your register's write operation takes two parameters: a value and a timestamp. A write begins as *partial* and ends either when it becomes *total* or when a write with higher timestamp ends, whichever occurs first[3]

This register departs from traditional registers in three ways. The first difference is that your register distinguishes between partial and total writes. A write may begin (as partial) but never be total because of an overlapping operation with higher timestamp. The second departure is that the timestamp returned by a read corresponds to the register's *cur-*

[2]For convenience, we use the syntactic convention that v is a value and ts is a timestamp.

[3]The conditions for when a write becomes total are implementation specific.

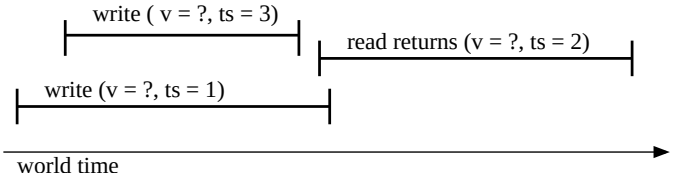

Figure 2: A disallowed sequence of register operations, independent of the values. First, the write with timestamp 1 should end when the other write ends. Second, the read should return a timestamp at least 3.

rent timestamp, not the timestamp used by the associated write. The purpose of this change will be evident later.

The third distinction is that the write has value and timestamp parameters, instead of just a value parameter. The timestamp parameter must satisfy two conditions.

The first condition is that no writer can have used the timestamp for a prior write. The La98 register model assumes that writers select timestamps from disjoint sets of numbers and use each timestamp at most once. The CL99 model, on the other hand, is more resilient; it tolerates clients that issue multiple writes with the same timestamp. The CL99 is more expensive because it employs a sophisticated pre-write phase, and guarantees that only one write per timestamp passes this phase. You can find details regarding this phase in the CL99 documentation (see Section 6).[4]

The second condition is that an operation's timestamp be at least as large as the timestamp of every previous non-overlapping operation; reads must satisfy this condition, as well. Figure 2 shows a sequence of register operations not allowed under our definitions.

Consistency Semantics

Your register has regular consistency semantics [6]. 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 a value that is concurrently being written.

[4]The La98 and CL99 tables are inspired by Lamport's protocol and Castro and Liskov's work, respectively.

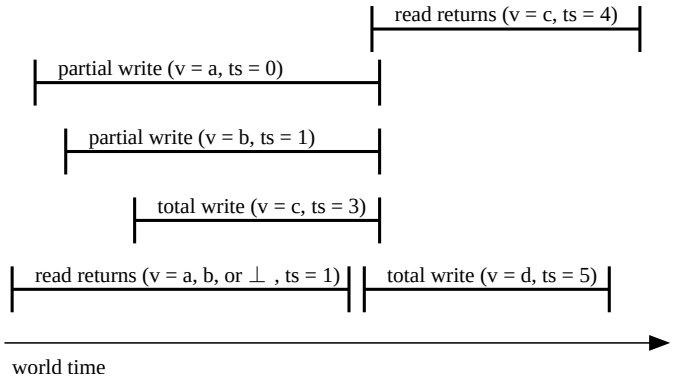

Figure 3: An illustration of regular consistency semantics under the partial order definition. The read with ts=4 returns c because no write is concurrent with it and the total write(c,3) immediately precedes the read according to the partial order. The read with ts=1, however, can return either a, b, or ⊥ because it is concurrent with the first two writes.

Traditionally, two operations are concurrent if they overlap in real time. Your register uses real time and timestamps to give a stronger partial ordering as follows:

1. Non-overlapping operations are ordered according to real time.
2. Overlapping operations are partially ordered according to their timestamp as follows:
   (a) writes are ordered by increasing timestamp.
   (b) a total write precedes a read if the read returns a higher timestamp.
   (c) a read precedes a write (whether partial or total) if the read returns an earlier timestamp.

Figure 3 gives an example of how this partial order affects reads under regular semantics.

4.3 Chapter 3: Console Circuitry

The console circuitry underneath each interface mediates communication between you and the register. For your safety, the register is embedded in the center of the Paxos table. You can request reads and writes via an interface at each seat.

In response to a press of a read button, the circuitry reads the register, stamps the resulting value and timestamp onto a token, and returns the token. The circuitry chooses the appropriate colored token based upon whether a total write has happened. If the register has notified the circuitry that a write has become total, then the circuitry selects a green token. Otherwise, the circuitry selects a red token. If the circuitry reads value $\perp$ from the register, then the circuitry stamps only the timestamp onto the token.

In response to a press of a write button, the circuitry first checks whether the button is blinking. If not, the button press has no effect. You can make the write button blink by using a red token. If the red token is blank, then insert it into the slot and the write button should immediately begin blinking. If the red token is not blank, set the dial to the same value as on the token, and then insert the token into the slot. When you press a blinking write button, the console circuitry issues a write with value v and timestamp ts to the register, where v is the dial's value and ts is the timestamp on the inserted red token.

4.4 Chapter 4: Write-Once Regular Register

Your table provides the abstraction of a write-once regular register. A write-once register stores a value, initially $\perp$, that changes at most once. Such a register is clearly useful to satisfy agreement.

Your register defines its value as the value written by the last total write. Therefore, for your register to be write-once, all total writes must write the same value. The Paxos table provides the following stronger property:

Theorem 1. *If write(v, ts) is the first write that is total, then all writes with a higher timestamp also write v.* [5]

Corollary 1: All reads after the first total write in the partial order return the same value.

Corollary 2: All green tokens have the same non-$\perp$ value stamped on them.

Warning! Reads issued before the first total write ends may return values that, strictly speaking, never end up actually written to your register. These values come from concurrent partial writes

Disclaimer. Unfortunately, due to budget constraints, your table cannot guarantee that a write will eventually become total.

We hope you enjoy your Paxos table. This concludes the user manual.

4.5 Discussion

The register abstracts away the details of asynchrony and quorum operations. In the next sections, we show an implementation of the register over a set of distributed processes where every read and write sends messages over asynchronous links to a quorum of processes.

The tokens play a key role in understanding how to guarantee agreement across mulitple writers. A writer needs to be careful to only issue writes that will not violate agreement. Because of the register's partial order definition, the result of a read indicates what value and timestamp pairs a write can use without violating agreement. Since the console circuitry stamps read results onto tokens, tokens serve as guards to writes, thus guaranteeing agreement.

Different failure assumptions affect the register's and tokens' implementations, not their specifications. For example, Paxos assumes a crash-failure model so it corresponds to a crash-tolerant regular register, plain-text tokens, and players that follow the Game's rules. PBFT, however, assumes a Byzantine failure model, and requires a Byzantine-tolerant register and secure tokens that curb ill-willed players. In the next sections, we show how Paxos and PBFT are optimized implemenations of the Paxos table for different failure models.

5 Paxos

5.1 Network Model

We consider an asynchronous distributed system of n processes. Processes communicate via message passing over unauthenticated links. We place no bound on message delay, clock drift, or the time necessary to execute a computation step.

[5] Because of space constraints, the proof is in the Appendix.

We assume reliable links to present the protocols more concisely. Any protocol that satisfies safety conditions using our network assumptions will still satisfy those same conditions using unreliable links. We also assume that failed processes do not recover.

5.2 The La98 Paxos Table

In this section, we show how to build the La98 Paxos table. We begin with a fault-tolerant implementation of the regular register. We implement the register over a set of processes. Player's reads and writes contact these processes by sending requests over asynchronous links. Upon receiving a request, a process may respond with an acknowledgement. We discuss how to interpret these acknowledgements when we present the La98 console circuitry and tokens.

Figure 4 describes the protocol a La98 process follows. Each process maintains a current timestamp, which is the the highest timestamp it has seen so far, and a value-timestamp pair representing the last write acknowledgement it sent.

If process i receives a write request for v and ts from console c, i first checks ts. If ts is at least the current timestamp, then i updates its current timestamp and sends a write acknowledgement back to c. A write becomes total when a majority of processes have acknowledged it.

If process i receives a read request with timestamp ts, i checks ts in the same manner as above. If the check succeeds, then i updates its current timestamp and responds with a read acknowledgement containing the value-timestamp pair of the last write acknowledgement it sent.

The La98 model has room for n_c consoles, each with a unique identifier from the set $\{0, \ldots, n_c - 1\}$. Console circuits do not fail. Otherwise, players may be left with an unresponsive console. Console circuits send read and write requests to the processes implementing the register. The circuitry maintains its own current timestamp and monitors the token slot and dial to determine when the write button should blink.

When a player pushes a blinking write button on console c, c sends a write request to the register with timestamp equal to the inserted red token's timestamp and value equal to the dial's current value. Upon receiving a set of write acknowledgements for the same timestamp from a majority of processes, c sets a flag, $valueWritten$, to indicate that a write has become total and the next read should return a green token.

```
currentTS := 0
lastWrite := (⊥, 0)

task on receive ⟨WRITE-REQUEST, v, ts, c⟩
  if (ts ≥ currentTS)
    currentTS := ts
    lastWrite := (v, ts)
    send ⟨WRITE-ACK, v, ts, i⟩ to console c
  endif

task on receive ⟨READ-REQUEST, ts, c⟩
  if (ts ≥ currentTS)
    currentTS := ts
    send ⟨READ-ACK, ts, lastWrite, i⟩ to console c
  endif
```

Figure 4: Process i's protocol for the La98 register.

La98 message	Paxos message
read request	prepare request
read ack	prepare response
write request	accept request
write ack	accept response

Table 1: How La98 messages map to Paxos.

When a player pushes console c's read button, c updates its current timestamp and broadcasts a read request, with the update timestamp, to the processes. The update guarantees that each read request is represented by a unique timestamp. If c receives acknowledgements to its read request from a majority of processes, c examines the value-timestamp pairs in each read acknowledgement. The value returned by the read is the value among the pairs with highest timestamp. The circuitry then selects a green or red token, based upon the $valueWritten$ flag, and stamps the value of the read and the current timestamp onto the token. Figure 5 gives the La98 console circuitry.

5.3 Paxos and the La98 Paxos Table

We now give a brief overview of the Paxos protocol and show how it relates to the La98 Paxos table. The reader can find the full Paxos protocol in [7].

Processes in Paxos play any of three roles: *proposers, acceptors, and learners.* Proposers propose

```
currentTS := c
valueWritten := false

task on read button push
  while(true) do
    broadcast ⟨READ-REQUEST, currentTS, c⟩ to processes
    currentTS := currentTS + n_c
    wait until received ⟨READ-ACK, currentTS, lastWrite, j⟩ from a majority of processes
      let v be the value among the lastWrites with highest timestamp
      if valueWritten then
        return ('green', v, currentTS)
      else
        return ('red', v, currentTS)
    on timeout
      continue

task on receive ⟨WRITE-ACK, v, ts, j⟩
  let msgs be the received ⟨WRITE-ACK, v, ts, k⟩ from a majority of processes
  if (msgs exists) then
    valueWritten := true
    forward msgs to every console {this line helps termination}
  endif

task on blinking write button push
  let v and ts be dial's value and timestamp on inserted red token
  broadcast ⟨WRITE-REQUEST, v, ts, c⟩
```

Figure 5: Console c's protocol for the La98 console circuitry.

values that acceptors then accept. If a learner discovers enough acceptors have accepted a value, then the learner can learn (or decide) that value.

These roles have analogues in the La98 table. Players propose values by pressing a blinking write button and learn values by picking up a green token. Each process implementing the La98 register is an acceptor.

Every proposer starts the protocol with a unique proposal number. A process issues a proposal in two phases. In the first phase, it sends a *prepare* request containing the current proposal number to all acceptors. An acceptor responds to a prepare request with i) the highest numbered proposal it has accepted so far and ii) a promise not to accept any more requests with numbers lower than the proposal number in the prepare request.

To enter the second phase, a proposer must receive responses to its prepare request from a majority of acceptors. In the second phase, a proposer selects the value in the responses with the highest proposal number. If there is no such value, then the proposer an arbitrary value. The proposer then sends an *accept* request containing the current proposal number and the selected value to all acceptors. After issuing an accept request, a proposer updates its current proposal number to the next unique number.

Proposal numbers correspond to the La98 register's timestamps. A proposer's prepare and accept phases correspond to La98 reads and writes, respectively. We provide Table 1 to relate Paxos messages to La98 messages.

An acceptor accepts a prepare or accept request provided that it has not accepted any other request so far with a higher proposal number. Finally, a learner can learn a value v once it realizes a majority of acceptors have responded to an accept request containing v.

This last step is analogous to a write becoming total. Note that we could have optimized the console by saving the value written by a total write in the $valueWritten$ variable, and obviating the need for subsequent reads of the register.

6 Byzantine Paxos

6.1 System Model

Faulty processes deviate arbitrarily from the protocol. However, they cannot subvert cryptographic

```
currentTS := 0
lastWrite := ⊥
primary := 0        {These two variables help to ensure that malicious players will}
timeoutVal := T     {not be able to indefinitely prevent a write from becoming total.}

task on receive ⟨PRE-WRITE-REQUEST, v, ts⟩_c
  if ( (c is the primary console) AND
       (ts = currentTS) AND
       (have not sent WRITE–REQUEST with timestamp ts) then
    broadcast ⟨WRITE-REQUEST, v, ts, c⟩_i to register processes
  endif

task on receive ⟨WRITE-REQUEST, v, ts, c⟩_j
  let msgs be the received ⟨WRITE-REQUEST, v, ts, c⟩_k from a register quorum
  if ( (ts = currentTS) AND
       (msgs exists) ) then
    lastWrite := msgs
    send ⟨WRITE-ACK, v, ts⟩_i to console c
  endif

task on receive ⟨READ-REQUEST⟩_c
  send ⟨READ-ACK, currentTS, lastWrite⟩_i to j

task at time timeoutVal
  currentTS := currentTS + 1
  primary := currentTS mod n_c
  timeoutVal := timeoutVal + (currentTS × T)
```

Figure 6: Process i's protocol for the CL99 register.

primitives such as digital signatures.

We use the same asynchrony and network assumptions as Section 5.1. To prevent message forgery, processes and consoles use private keys to sign messages that others then verify using the corresponding public key. We use the notation $\langle M \rangle_x$ to indicate a message M signed by console or process x. Improperly signed messages are discarded.

6.2 The CL99 Paxos Table

The CL99 table has room for n_c consoles and uses n_r processes to implement the regular register. Consoles and processes have unique identifiers. In particular, we identify each console from the set $\{0, \ldots, n_c - 1\}$. We first present an implementation where console circuits do not fail and later show how to modify the register if we relax this assumption. Any number of players can deviate from the rules.

Figure 6 describes the protocol that a correct process follows. The CL99 defines a quorum as $\lceil \frac{2n_r}{3} \rceil$ processes and tolerates $\lfloor \frac{n_r - 1}{3} \rfloor$ process failures. Each CL99 process maintains a current timestamp and last written variable as in the La98 register. CL99 processes, however, do not update their timestamps in response to receiving a request; they increment their timestamps based on a timeout value and only receive requests for their current timestamp. This prevents attacks that send requests with high timestamps with the intent to delay writes from becoming total. Further, CL99 processes use the notion of a primary for each timestamp. The primary for timestamp ts is the console with id ts modulo n_c; only the primary for ts is allowed to issue a write for ts.

The CL99 register introduces an additional phase for writes to prevent writes using different values but identical timestamps. Consoles send pre-write requests to CL99 processes. If a process i receives a pre-write request for v and ts from console c, i checks that 1) c is the primary for ts, 2) ts is the current timestamp, and 3) it has not responded to a pre-write request for ts yet. If all three checks are true, then i broadcasts a write request message, containing v, ts, and c. Note that at most one pre-write request can gather a quorum of corresponding write request messages.

If process i receives a quorum of properly signed write request messages containing v, ts, and c, then

```
currentTS := 0
valueWritten := false

task on read button push
  while(true) do
    broadcast ⟨READ-REQUEST⟩_c to processes
    wait until received a quorum of ⟨READ-ACK, ts, lastWrite⟩_j with same ts
      if (ts ≥ currentTS) then
        let v be the value among the lastWrites with highest timestamp
        currentTS := ts
        if valueWritten then
          return ('green', v, currentTS)
        else
          return ('red', v, currentTS, quorum of READ–ACKs)
      endif
    on timeout
      continue

task on receive ⟨WRITE-ACK, v, ts⟩_j
  let msgs be the received ⟨WRITE-ACK, v, ts⟩_k from a quorum
  if (msgs exists) then
    valueWritten := true
        currentTS:= ts + 1
    forward msgs to every console {this line helps termination}
  endif

task on blinking write button push  {implicitly verifies token}
  let v and ts be dial's value and timestamp on inserted red token
  broadcast ⟨PRE-WRITE-REQUEST, v, ts⟩_c
```

Figure 7: Console c's protocol for the CL99 console circuitry.

i first compares its current timestamp against ts. If they match, then i sends the appropriate write acknowledgement to c. In addition, i saves the quorum of write request messages as proof that it was allowed to send the write acknowledgement message.

If process i receives a read request from console c, i responds with a read acknowledgement containing the current timestamp and the last proof that it assembled for a write acknowledgement. The quorum of messages constituting this proof is an unforgeable version of an La98 process's value-timestamp pair, which represents the last write acknowledgement that that process sent.

The CL99 console implementation, in Figure 7, is very similar to the La98 model; it also maintains a timestamp variable and monitors the slot and dial.

When a player pushes a blinking write button on console c, c sends a pre-write request to the register with timestamp equal to the inserted red token's timestamp, and value equal to the dial's current value. Upon receiving a quorum of write acknowledgements for the same value and timestamp, console c sets the $valueWritten$ flag to true and sets its current timestamp to be greater than ts so that subsequent reads will return the totally written value (see Corollary 1).

When a player pushes console c's read button, the console broadcasts a read request to all processes. If c receives a quorum of read acknowledgements for timestamp ts greater than or equal to its current timestamp, then c examines the quorums of write request messages in each acknowledgement. Remember that conceptually, each quorum of write request messages is just a secure value-timestamp pair representing the last write a process acknowledged. As in the La98, c selects the value among these 'pairs' with highest timestamp and stamps the value and ts onto an appropriately colored token, again based upon the $valueWritten$ flag.

As presented, we can improve the CL99 table in at least three ways. First, the write for timestamp 0 does not require a red token. Second, similar to the La98 optimization, the CL99 console circuitry can save the value written by a total write so that sub-

CL99 table message	PBFT message
pre-write request (with no red token)	pre-prepare
write request	prepare
write ack	commit
read ack	view change
pre-write request (with red token)	new view

Table 2: How CL99 messages map to PBFT.

sequent read button presses do not trigger a read of the register.

Third, we can alter the CL99 register to tolerate players manipulating their console circuits. The implementation in Figure 7 relies on a console circuit correctly verifying a red token. If we allow consoles to fail, we can alter the pre-write message format to contain the inserted red token. Upon receiving a pre-write message, processes then verify the included red token with respect to the pre-write's value and timestamp. If the verification fails, then the process discards the pre-write request.

6.3 PBFT and the CL99 Paxos Table

In this section, we see how Castro and Liskov's PBFT is essentially the CL99 table with the above three optimizations.

The PBFT protocol is a Byzantine tolerant state-machine replication algorithm. It is hard to see the connection between PBFT and Byzantine Paxos because PBFT handles aspects such as failures, checkpoints, garbage collection, and quorums that quickly increase the protocol's complexity. We strip PBFT down to the elements necessary to achieve consensus and present this version below while explaining its similarities to the CL99 Paxos table.

Processes in PBFT 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* for view v is the process with id v modulo n. PBFT assumes at most $f \leq \lfloor \frac{n-1}{3} \rfloor$ process failures. Therefore, a quorum consists of $n - f$ processes.

Conceptually, each process in PBFT plays the roles of player, console circuit, and register process. PBFT's views correspond to the CL99 register's timestamps.

In normal-case operation (without primary failures), the PBFT protocol 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 $\langle \texttt{PRE-PREPARE}, vue, val \rangle_p$, 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 $\langle \texttt{PRE-PREPARE}, vue, val \rangle_p$ it broadcasts a corresponding message $\langle \texttt{PREPARE}, vue, val \rangle_i$ 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 vue, i broadcasts $\langle \texttt{COMMIT}, vue, val \rangle_i$ 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 can decide the corresponding value.

Table 2 gives the mapping from messages in PBFT to messages in the optimized CL99 table.

If the primary fails, processes elect a new primary to issue proposals. A process increments its current view if it has not decided within some timeout period. By incrementing their views, processes essentially elect a new primary.

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 quorum of prepare messages that triggered i to broadcast its last commit message. When the primary p for view vue receives a quorum of valid view-change messages, p broadcasts $\langle \texttt{NEW-VIEW}, vue, \mathcal{V}, \langle \texttt{PRE-PREPARE}, vue, val \rangle_p \rangle_p$, $\mathcal{V}$ is the triggering quorum of 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 is a special noop value.

The $\mathcal{P}$ field of a view-change message is essentially the quorum of write request messages in a CL99 read acknowledgement. Remember that conceptually this quorum is a secure value-timestamp 'pair.' The $\mathcal{V}$

field of a new-view message is conceptually a red token.

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.

7 Conclusion

We use the Paxos table to introduce the register and token abstractions, which we claim makes it easier to understand Paxos and PBFT's subtleties. Because implementation details manifest themselves only in these abstraction's implementations, the table provides a unified framework to conceptualize Paxos and PBFT. We are working on expressing variants like Disk Paxos and Fast Byzantine Paxos using the Paxos table.

References

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

[2] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, 1999.

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

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

[5] E. Gafni and L. Lamport. Disk paxos. In *DISC*, pages 330–344, 2000.

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

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

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

[9] B. Lampson. *The ABCDs of Paxos.* Presented at PODC, 2001.

[10] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, pages 569–578, New York, USA, 1997. ACM Press.

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

[12] R. D. Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the paxos algorithm. In *WDAG*, pages 111–125, 1997.

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

8 Appendix

Theorem 1: *If write(v, ts) is the first write that is total, then all writes with a higher timestamp also write v.*

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

Base Case: There are 0 writes after write(v, ts). Trivial.
Inductive Hypothesis: The first $n \geq 0$ writes after write(v, ts) all write the same value.

1. Consider the $n + 1^{th}$ write(v', ts') after write(v, ts).
2. ts' is greater than every timestamp used by the first n writes after write(v, ts).
3. In general, a write(v', ts') can only be issued with either a blank red token with timestamp ts' or a red token stamped with v' and ts'.
4. However, write(v', ts') cannot be issued with a blank red token because a read that returns timestamp ts' cannot return $\perp$ since write(v, ts) completed and $ts < ts'$.
5. Now consider the read that results in a red token stamped with v' and ts'. Because the register is regular, v' can either be the last written value or a value being concurrently written.
6. Observe that the last written value and any value being concurrently written can only be v due to the Induction Hypothesis.
7. $v' = v$