

The Game of Paxos

Harry C. Li, Lorenzo Alvisi, and Allen Clement

University of Texas at Austin, Department of Computer Sciences

Abstract

We introduce the Game of Paxos to simplify the presentation of Paxos-style consensus protocols. We use this game to show how Lamport's Paxos and Castro and Liskov's PBFT are the same consensus protocol, but for different failure models. In this game, players try to store some value in a write-once register and quit only when they learn the register's final value. The write-once register contains two novel abstractions: (i) a Paxos register that captures how processes in both Paxos and PBFT propose and decide values and (ii) tokens that capture how these protocols guarantee agreement despite partial failures. We encapsulate the differences between Paxos and PBFT in the implementation details of these abstractions.

1 Introduction

You find a group of people engaged around a circular table. Intrigued, you edge closer to find that seat belts bind everyone into his or her chair. Each person busily presses flashing buttons on the table. You see the glint of a token in a lady’s hand, but she quickly inserts the token into a slot and continues pressing buttons. Next, a green light catches your eye and you turn your head just in time to see a man unlock his belt. Your curiosity finally overcomes your caution and you approach the table...

The description above captures moments from the Game of Paxos—a game designed to simplify the presentation of Paxos-style consensus protocols. We frame Lamport’s Paxos [11] and Castro and Liskov’s PBFT [3] as instances of this game and show that they are the same protocol, but for different failure models.

Since solving consensus in an asynchronous system with failures is impossible [7], Paxos gives us the next best thing for crash failures. It guarantees the safety properties of consensus and relies on synchrony only for liveness [6]. PBFT provides the same guarantees as Paxos for asynchronous systems with Byzantine faults.

At a high-level, these consensus protocols are intuitively similar. They both guarantee safety and liveness as explained above. In addition, both protocols use leaders to coordinate actions among quorums [5, 14, 18] of processes. And yet, while some refer to PBFT as Byzantine Paxos [13], 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 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 overwhelming¹. Second, PBFT is more complex than Paxos because PBFT assumes a weaker failure model. This additional complexity obfuscates the underlying similarities between these two protocols.

We provide two abstractions that help overcome

¹It 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. [21]

the above difficulties. Our first abstraction, the *Paxos register*, hides the details of quorum operations. 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 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 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.

Our second abstraction encapsulates in a *token* 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 requiring tokens as guards to every write, we obtain a write-once register. We claim that describing Paxos-style consensus protocols as operations on a write-once Paxos register simplifies the presentation of these protocols, and thus, makes algorithms like PBFT more accessible.

Differences between protocols like Paxos and PBFT manifest themselves in the implementation of the above abstractions, not in the specification. For example, under benign failures, we use a crash-tolerant Paxos register and issue plain tokens. Under Byzantine failures, we use a Byzantine-tolerant Paxos register and issue secure tokens to prevent foul play.

The Game of Paxos is the first register-based unified treatment of crash and Byzantine Paxos. Although registers simplify the exposition of asynchronous consensus protocols [8], the only existing unified presentation of these protocols does not utilize a register [13]. As for prior efforts that use a register to explain consensus, they are limited to either benign or Byzantine failures and yield non-standard consistency semantics [2, 4, 17]. The Paxos register handles both kinds of faults and provides semantics similar in flavor to the traditional notion of regular semantics [10].

We explain the Game of Paxos in Section 3. Section 4 defines the Paxos register’s and tokens’ semantics. Finally, in Sections 5 and 6, we show how specif-

ically Paxos and PBFT implement the Paxos register and token abstractions.

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) [19] and use it to model, verify, and analyze Paxos. Using the Clock GTA, they were the first to study the performance of Paxos both during failure-free executions and in the presence of failures.

Lamport’s own second take at Paxos [12] 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 [13], a version of Lamport’s original protocol, and derives Classic Paxos, Byzantine Paxos, and Disk Paxos [8] 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 encapsulate 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 [2]. Guerraoui and Raynal later refined this safety-liveness separation into the Alpha and Omega abstractions, respectively [9]. Both works use this separation to gain precious insight into the common internal structure of several crash tolerant Paxos. Our paper can be seen as complementary to these efforts: the Paxos register abstraction tries to elucidate one of the subtlest aspect 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* operation, respectively. We believe this addresses a crucial pedagogical step in understanding Paxos variants, especially those variants that tolerate Byzantine faults.

Dutta et al. focus on establishing complexity bounds for asynchronous Byzantine consensus [17]. Their treatment contains an construct, the *WriteProof*, that is akin to our token. The Paxos register and

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

1. Look to see whether the green light is on or off.
2. Push the *read* button and examine the token that drops into the tray.
3. If the light was on in step 1, the final value of the register is stamped on the token. The buckle has been unlocked. Game over.
4. If the light was off in step 1 and the token is stamped with a value, place the token in the slot, set the dial to the token’s value, and push the now blinking *write* button (it will turn back off). Go back to step 1.
5. If the light was off in step 1 and the token is blank, place the token in the slot, set the dial to any value, and push the blinking *write* button. Go back to step 1.

Figure 1: Instruction label at each seat.

token abstraction differentiate our work from theirs because our specifications enable us to unify the presentation of Paxos and PBFT.

Chockler and Malkhi’s *ranked register* [4] drew inspiration from Boichat et al.’s earlier work [1]. 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 uses regular consistency semantics (albeit using a partial order stronger than the one induced by real time alone), whereas the ranked register’s specification resembles neither safe, regular, nor atomic semantics.

Shao et al. [20] 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 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 Paxos 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, an inset metal tray, and a green light that is initially off. 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 (see Figure 1).

In the Game of Paxos, tokens serve as proof that a read returned a particular value. Inserting a token into the slot enables a player to issue a write. The green light indicates that a value has been written into the register.

For this game to help players reach consensus, all tokens in step 3 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 tokens in step 3 are stamped with the same value.

4.1 Chapter 1: Features & Safety

Your Paxos table comes with a Paxos 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 2. 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 token after the green light has turned on.

Agreement: All tokens in step 3 are stamped with the same value.

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

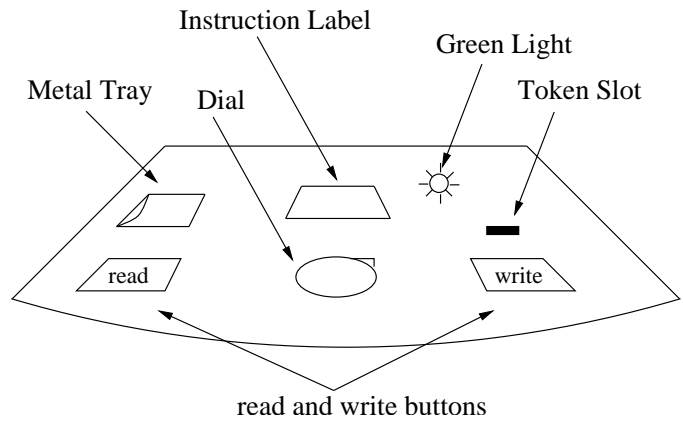


Figure 2: Example interface at each seat.

4.2 Chapter 2: Paxos Register Semantics

Your Paxos register stores value and timestamp pairs. For convenience, we use the syntactic convention that v is a value and ts is a timestamp. Your register provides *read* and *write* operations to access the value and timestamp. The register’s initial value is \perp , which cannot be written. 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 with no relation to the world clock.

Paxos Register Operations

Your Paxos register’s read operation takes no parameters and returns a value and timestamp pair. The write operation takes two parameters: a value and a timestamp. Your Paxos register departs from traditional registers [10, 14] in the following ways.

First, read and write operations have timestamps (not just values) associated with them. A read’s timestamp is the timestamp that the read returns; a write’s timestamp is the timestamp parameter the write uses.

Second, a read’s timestamp is at least as large as the timestamp of every previous non-overlapping operation.

Third, read operations only return values written by *visible* writes. The Paxos register defines visible writes in a declarative way by stating that no two visible writes have the same timestamp. The mechanisms used to guarantee this property depend on the

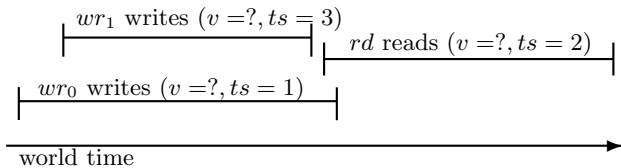


Figure 3: A sequence of disallowed Paxos register operations—-independent of the value they read or write. According to the specification, wr_0 should end when wr_1 ends and rd should return a timestamp of at least 3.

specific model of Paxos register. The La98 and CL99 models (explained in Sections 5 and 6) are the most common models. Henceforth, we only consider visible writes.

Fourth, 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. 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*.

Fifth, 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. Your Paxos register further guarantees that a write invoked under good conditions eventually becomes total.

We leave it to individual implementations to guarantee the above conditions. Figure 3 shows a sequence of operations that are disallowed by the Paxos register specification.

Consistency Semantics

The semantics of a Paxos register is similar to regular consistency semantics [10]. 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.

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. Your register uses real-time and

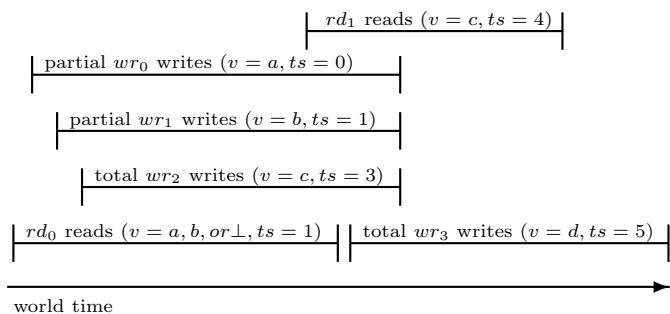


Figure 4: An illustration of regular consistency semantics under *our* partial order definition. rd_1 returns c because no write is concurrent with it and wr_2 immediately precedes rd_1 . rd_0 , however, can return either a , b , or \perp because it is concurrent with wr_0 and wr_1 .

timestamps to define a different partial order:

1. writes are ordered by increasing timestamp.
2. a write precedes a read if the read begins after the write ends in real time.
3. a total write precedes a read if the read returns a higher timestamp.
4. a read precedes a write (whether partial or total) if the read returns an earlier timestamp.

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 4 gives an example of how *our* 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. You cannot, however, request an acknowledged operation. The underlying circuitry periodically invokes the acknowledged operation on your behalf, checks the result, and turns on the green light if the returned set is non-empty.

When a read button is pressed, the circuitry reads the register and engraves a timestamp—and, when appropriate, a value—onto a token that is then output in the tray. If the circuitry reads value \perp from the register, then the circuitry engraves only the timestamp onto the token; we consider this a blank token.

When a write button is pressed, 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 token. If the token is blank, then insert it into the slot and the write button will immediately begin blinking. If the 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 token.

4.4 Chapter 4: Write-Once Register

By restricting the values that a write can use via the tokens, your Paxos table guarantees that all total writes write the same value. In fact, it guarantees the following stronger property:

Theorem 1. *If $\text{write}(v, ts)$ is the first write that is total, then all writes with a higher timestamp also write v .*²

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

Corollary 2: All reads invoked after a green light has turned on return the same non- \perp value.

Together, the components of your Paxos table — the Paxos register, consoles, and tokens — ensure that only one value is totally written to your Paxos register. The Paxos table is essentially a write-once register.

A write-once register stores a value, initially \perp , that changes at most once. Your table defines its value as the value of the last total write to the Paxos register. Since by Theorem 1 all total writes write the same value, the table's value changes at most once, and is therefore write-once.

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

Disclaimer. Unfortunately, because of budget constraints, your table cannot, in general, guarantee that eventually any write will become total.

²Because of space constraints, we include the proof in the Appendix.

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

4.5 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 distributed processes where register operations translate into a set of messages sent over asynchronous links to a quorum of processes.

Clearly separating the read and write operations, instead of combining them as in Boichat et al.'s single propose operation [2], exposes an important component of Paxos-like protocols—the tokens. Tokens serve as guards to writes. In Paxos-like protocols, writers need to be careful to only issue writes that will not violate agreement. Tokens allow only writes whose value-timestamp pairs agree with those that the console circuitry read from the Paxos register. By the consistency semantics of the Paxos register, these are precisely the pairs that a write can use without violating agreement.

The tokens encapsulate the most difficult part in understanding the differences between Paxos and PBFT—how to guarantee agreement when the leader fails. We use plain tokens on a crash-tolerant table because we assume players follow the rules of the Game. However, if players can be ill-willed, as in the Byzantine case, we use secure tokens that are unforgeable. This prevents a player from invoking a write that violates Theorem 1, and consequently agreement.

5 The La98 Paxos Table

In this section, we show how to build a La98 Paxos table. We begin by describing how to implement a fault-tolerant Paxos register over a set of processes. Players' read and write operations contact these processes by sending requests over asynchronous links. Upon receiving a request, a process may respond with acknowledgements. Each player's circuitry examines read and write acknowledgements to compute the results of read and acknowledged operations, respectively. We discuss how to interpret these responses when we present the La98 console circuitry and tokens.

```

currentTS := 0
lastWrite := ( $\perp$ , 0)

task on receive  $\langle$ WRITE-REQUEST, v, ts, c $\rangle$ 
  if (ts  $\geq$  currentTS)
    currentTS := ts
    lastWrite := (v, ts)
    send  $\langle$ WRITE-ACK, v, ts, i $\rangle$  to console c
  endif

task on receive  $\langle$ READ-REQUEST, ts, c $\rangle$ 
  if (ts  $\geq$  currentTS)
    currentTS := ts
    broadcast  $\langle$ READ-ACK, ts, lastWrite, i $\rangle$  to
      consoles
  endif

```

Figure 5: Process i 's protocol for the La98 register.

5.1 Operating Assumptions

Our implementation assumes an asynchronous distributed system of n processes of which fewer than half can fail; processes fail only by permanently crashing. Processes communicate via message passing over unauthenticated, unreliable links. To keep our discussion concise, however, it is convenient to describe the inner workings of the La98 model as if links were reliable—we do so in the following. The La98 Paxos table operates safely in an environment where there is no bound on message delay, clock drift, or on the time necessary to execute a computation step. However, the La98 relies on players using unique timestamps for each write invocation.

5.2 Implementation

Figure 5 describes the protocol a La98 process follows. Each process maintains a current timestamp, which is 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 with value v and timestamp ts from console c , i first checks whether ts is at least as large as the current timestamp. If so, i updates its current timestamp and broadcasts a write acknowledgement to all consoles.

If process i receives a read request with timestamp ts , i checks ts as in the write case. 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.

```

currTS := c

task on read button push
  let (v, ts) := read()
  engrave (v, ts) onto a new token tok
  return tok

task on blinking write button push
  let v and ts be dial's value and the timestamp
    on inserted token, respectively
  write(v, ts)

periodically:
  if acknowledged()  $\neq$  {} then
    turn on green light

procedure read()
  while(true) do
    currTS := currTS +  $n_c$ 
    broadcast  $\langle$ READ-REQUEST, currTS, c $\rangle$  to processes
    wait until received  $\langle$ READ-ACK, currTS, lastWrite, j $\rangle$ 
      from a majority of processes
    let v be the value among the lastWrites with
      highest timestamp
    return (v, currTS)
  on timeout
    continue

procedure write(v, ts)
  broadcast  $\langle$ WRITE-REQUEST, v, ts, c $\rangle$ 

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

```

Figure 6: Console c 's protocol for the La98 console circuitry.

The La98 model has room for n_c consoles, each with a unique identifier from the set $\{0, \dots, n_c - 1\}$. Console circuits do not fail—otherwise, players may be left with an unresponsive console. Each console circuit accesses the La98 register through read, write, and acknowledged procedures. Further, each circuit maintains a unique timestamp that the procedures use to communicate with the register.

When console c invokes a read procedure, c updates its current timestamp and broadcasts a read request, with the updated timestamp, to the processes. The update guarantees that a unique timestamp represents each read request. If c receives acknowledgements to its read request from a majority of processes, c examines the value-timestamp pairs in each read acknowledgement and returns the value with highest timestamp and the current timestamp as the result of the read.

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 console c invokes the write procedure, c broadcasts a write request message to the register’s processes. In the La98 Paxos table, every write is visible.

When console c invokes the acknowledged procedure, c checks its write acknowledgements from the processes and returns a set of pairs, where each pair corresponds to a write request that has garnered a majority of acknowledgements.

Periodically, the circuitry checks the result of the acknowledged operation. If the returned set is non-empty, indicating a majority of processes have acknowledged a write, then the circuitry turns on the green light.

Pushing the read button makes the circuitry invoke the read procedure and engraves the resulting value and timestamp onto a token.

Pushing a blinking write button causes a write request to be sent to the register timestamp with value equal to the dial’s current value and timestamp equal to the inserted token’s timestamp. The circuitry monitors the token slot and dial to determine when the write button should blink. Figure 6 gives pseudocode describing the actions performed by the La98 console circuitry.

5.3 Paxos and the La98 Paxos Table

We base the La98 Paxos table upon Lamport’s Paxos protocol [11].

Processes in Paxos play any of three roles: *proposers*, *acceptors*, and *learners*. Proposers propose 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 writing to the register and learn values by reading from the register after the green light is lit. Each process that participates in implementing the La98 register is an acceptor.

In Paxos, 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 selects 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. Table 1 relates Paxos messages to La98 messages.

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

This last condition is analogous to the one required for a write to become total in La98. Note that we could have optimized the console by examining the pairs returned by the acknowledged operation, thus obviating the need for subsequent reads of the register.

6 The CL99 Paxos Table

In this section, we show how to build a CL99 Paxos table. We describe how to implement a Byzantine fault-tolerant Paxos register over a set of processes. Similar to the La98 table, read and write operations send requests to these processes and processes respond with acknowledgements. As before, we explain how to interpret these acknowledgements for the read and acknowledged operations in our discussion of the console circuitry and tokens.

```

currentTS := 0
lastWrite := ⊥
primary := 0
timeoutVal := T

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 ts) ) then
    broadcast ⟨WRITE-REQUEST, v, ts, c⟩i to 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, nonce⟩c
  send ⟨READ-ACK, nonce, currentTS, lastWrite⟩i to j

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

```

Figure 7: Process i 's protocol for the CL99 register.

6.1 Operating Assumptions

We use the same operating assumptions as the La98 table with three exceptions. First, fewer than a third of processes fail. Second, processes fail by arbitrarily deviating from the protocol. Third, any number of players can deviate from the Game's rules. However, neither faulty processes nor players can subvert cryptographic primitives such as digital signatures.

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.

For simplicity, we first present an implementation where console circuits do not fail, which prevents players from invoking writes without the appropriate tokens. We then show how to modify the register if players manipulate their console's circuitry.

6.2 Implementation

The CL99 table has room for n_c consoles and uses n_r processes to implement the Paxos register. Consoles

and processes have unique identifiers. For convenience, we identify each console from the set $\{0, \dots, n_c - 1\}$.

Figure 7 describes the protocol that a correct process follows. The CL99 register defines a quorum as $\lceil \frac{2n_r}{3} \rceil$ processes. Each CL99 process maintains a current timestamp and last written variable. 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 respond to requests containing 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 can invoke a visible write for ts .

Despite console circuits never failing, players can still invoke multiple writes using the same timestamp but different values, also known as poisonous writes [16]. The CL99 register introduces a pre-write phase to solve this. A write begins by sending 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 . A write is visible if and only if it gathers 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 i first compares its current timestamp against ts . If they match, then i broadcasts an appropriate write acknowledgement to all consoles. In addition, i saves the quorum of write request messages as proof that the write was visible.

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 circuitry, in Figure 8, is very similar to the La98 circuitry; CL99 consoles access

```

currentTS := 0

task on read button push
  let  $(v, ts) := \text{read}()$ 
  engrave  $\langle v, ts \rangle_c$  onto a new token  $tok$ 
  return  $tok$ 

task on blinking write button push
  let  $v, ts$  be dial's value and token's timestamp
  write  $(v, ts)$ 

periodically:
  if acknowledged()  $\neq \{\}$  then
    turn on green light

procedure read()
  while(true) do
    let  $nonce$  be a fresh nonce
    broadcast  $\langle \text{READ-REQUEST}, nonce \rangle_c$  to processes
    wait until received a quorum of
       $\langle \text{READ-ACK}, nonce, ts, lastWrite \rangle_j$  with same  $ts$ 
    if  $(ts \geq currentTS)$  then
      let  $v$  be the value among the  $lastWrites$ 
        with highest timestamp
      currentTS :=  $ts$ 
      return  $(v, currentTS)$ 
    endif
  on timeout
    continue

procedure write( $v, ts$ )
  broadcast  $\langle \text{PRE-WRITE-REQUEST}, v, ts \rangle_c$ 

procedure acknowledged()
  return the set of value-timestamp pairs  $(v, ts)$ 
    such that  $c$  received  $\langle \text{WRITE-ACK}, v, ts \rangle_k$  from
    a quorum of processes

```

Figure 8: Console c 's protocol for the CL99 console circuitry.

the Paxos register via read and write procedures and construct tokens accordingly.

When a console c invokes the read procedure, c 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 secure value-timestamp pairs with highest timestamp and returns the value and current timestamp as the result of the read.

When a console c invokes the write procedure, c broadcasts a pre-write request message to the regis-

ter's processes.

When console c invokes the acknowledged procedure, c checks its write acknowledgements from the processes. The acknowledged procedure returns a set of value-timestamp pairs, each pair corresponding to a visible write that has collected a quorum of acknowledgements. As in the La98 console circuitry, the CL99 circuitry also monitors the output of the acknowledged operation and turns the green light on when the output is not the empty set.

Pushing the read button makes the circuitry follow almost the same actions as in the La98 console circuit. The difference is that the CL99 circuitry *securely* engraves the value-timestamp pair onto the token.

Pushing a blinking write button also makes the circuitry follow almost the same actions as in the La98 circuit. The departure here is that instead of sending a write request to the register processes, the console circuit sends a pre-write request. Figure 8 gives pseudocode describing the actions performed by the La98 console circuitry.

As presented, we can improve the CL99 table in at least three ways. First, the write for timestamp 0 does not require a token. Second, similar to the La98 optimization, the CL99 console circuitry can examine the pairs returned by the acknowledged operation so that subsequent read button presses do not trigger a read of the register.

Third, we can weaken our assumptions by allowing players to manipulate their console circuitry. Consequently, players could invoke writes without the appropriate token or with a forged token. To prevent forgery, the token should be the quorum of signed read acknowledgement messages from the read procedure. To resolve the problem of players invoking writes without matching tokens, we require that the token be included in the pre-write request.

Upon receiving a pre-write message, processes verify the included token with respect to the pre-write's value and timestamp. If the verification fails, then the process discards the pre-write request. This optimization is important in seeing the relationship between PBFT and the CL99 table in the next section.

6.3 PBFT and the CL99 Paxos Table

We base the CL99 Paxos table upon Castro and Liskov's PBFT protocol [3].

CL99 msg	PBFT msg
pre-write request (w/out token)	pre-prepare
write request	prepare
write ack	commit
read ack	view change
pre-write request (w/ token)	new view

Table 2: How CL99 messages map to PBFT.

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 additional aspects such as 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 $\{0, \dots, n - 1\}$, 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 $\langle \text{PRE-PREPARE}, vue, val \rangle_p$, 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 $\langle \text{PRE-PREPARE}, vue, val \rangle_p$ it broadcasts a corresponding message $\langle \text{PREPARE}, vue, val \rangle_i$ and enters the pre-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 $\langle \text{COMMIT}, vue, val \rangle_i$ 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 $\langle \text{VIEW-CHANGE}, vue, \mathcal{P} \rangle_i$ 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 $\langle \text{NEW-VIEW}, vue, \mathcal{V}, \langle \text{PRE-PREPARE}, vue, val \rangle_p \rangle_p$, \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 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

The Game of Paxos provides a unified framework for simpler presentations of asynchronous consensus protocols like Paxos and PBFT. We use this game to introduce the Paxos register and token abstractions. These abstractions elucidate the similarities between these protocols, while encapsulating protocol-specific details in each abstraction’s specification. We believe we can express other asynchronous consensus protocols like Disk Paxos [8] and Fast Byzantine Paxos [15] using the Game, though this remains for future work.

References

- [1] 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.
- [2] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui. Deconstructing Paxos. *SIGACT News*, 34(1):47–67, 2003.
- [3] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.
- [4] 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.
- [5] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Computing Surveys*, 17(3):341–370, 1985.
- [6] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.
- [7] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- [8] E. Gafni and L. Lamport. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 330–344, 2000.
- [9] R. Guerraoui and M. Raynal. The Alpha and Omega of asynchronous Consensus. Technical Report PI-1676, IRISA, January 2005.
- [10] L. Lamport. On interprocess communication, part I: Basic formalism. *Distributed Computing*, 1(2):77–85, 1986.
- [11] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [12] L. Lamport. Paxos made simple. *Distributed Computing Column of ACM SIGACT News*, 32(4):51–58, 2001.
- [13] B. Lamson. *The ABCDs of Paxos*. Presented at 20th Annual ACM Symposium on Principles of Distributed Computing, 2001.
- [14] 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.
- [15] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2005.
- [16] J.-P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine storage. In *Proceedings of the 16th International Conference on Distributed Computing*, pages 311–325, London, UK, 2002. Springer-Verlag.
- [17] M. V. Partha Dutta, Rachid Guerraoui. Best-case complexity of asynchronous Byzantine consensus. Technical Report 200499, EPFL/IC, February 2005.
- [18] D. Peleg and A. Wool. The availability of quorum systems. Technical report, Jerusalem, Israel, 1993.
- [19] 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.
- [20] 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.
- [21] 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 $n + 1^{\text{th}}$ 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 token with timestamp ts' or a token stamped with v' and ts' .
4. However, write(v', ts') cannot be issued with a blank token because a read that returns timestamp ts' cannot return \perp since write(v, ts) is total and $ts < ts'$.
5. Now consider the read that results in a 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. $v' = v$

The next two theorems state that the La98 and CL99 registers implement a Paxos register under different failure assumptions. Before presenting the corresponding proofs, we list important elements of what such a proof should contain:

- An explanation of how the implementation guarantees unique timestamps among visible writes.
- A proof that shows the register implementation satisfies the consistency semantics of a Paxos register.
- A definition of good conditions under which a write eventually becomes total.

Theorem 2. *The La98 register is a crash-tolerant Paxos register.*

Proof: The La98 register assumes that users select timestamps from disjoint sets and never use a timestamp for a write more than once. By assumption, therefore, every write meets the condition for being visible. We now prove that the La98's read operation satisfies the Paxos register's consistency semantics.

Lemma 1. *Reads only return the values of visible writes.*

Proof: Trivial because every write is visible.

Lemma 2. *Every read only returns a value written by an earlier or concurrent write.*

Proof:

1. Consider a read rd that returns v and ts .
2. wr is the only write that writes v .
3. Assume wr is later in the partial order than rd .
4. In general, wr can be later than rd if wr begins after rd ends or if wr 's timestamp is greater than rd 's.
5. wr cannot begin after rd ends because then rd could not return v .
6. Therefore, wr 's timestamp is greater than rd 's.
7. This is impossible because a read only returns values that were written with timestamps less than or equal to the read's timestamp.

Lemma 3. *Every read only returns a value written by the last total write or by a write since the last total one.*

Proof:

1. Consider a total write wr_t and a read rd later than wr_t in the partial order.
2. Let ts_t be the timestamp of wr_t .
3. Let rd return v_r and $ts_r \geq ts_t$.
4. Assume that no write since and including wr_t has written v_r .
5. rd returns v_r and ts_r because of a quorum of read acknowledgements.
6. At least one of the read acknowledgements contains a *lastWrite* field with timestamp at least ts_t .
7. In order for rd to return v_r , at least one of the read acknowledgements' *lastWrite* field is (v_r, ts) , where $ts \geq ts_t$.
8. Contradiction. No write since and including wr_t has written v_r .

Together, Lemmas 2 and 3 prove that the La98 register has the same consistency semantics as a Paxos register.

Lemma 4. *Under good conditions, the La98 register guarantees that a write eventually becomes total.*

The La98 register defines good conditions as follows. First, the system behaves synchronously. Second, henceforth exactly one user invokes register operations. Third, the privileged user invokes a write with a timestamp higher than all previous writes.

Proof:

1. Given the system behaves synchronously and only one user reads or writes to the register.
2. Consider the user who still invokes operations.
3. If this user invokes a write with a timestamp higher than all previous writes, then that write will eventually become total because the system is synchronous.
4. This user can select a high enough timestamp by first reading the current timestamp from the register and then choosing a higher one.

In order to satisfy these good conditions, users should elect a privileged user who is the only one allowed to read and write to the register. Electing such a user, however, is orthogonal to guaranteeing the register's semantics. \square

Theorem 3. *The CL99 register is a Byzantine-tolerant Paxos register.*

Proof:

A write in the CL99 register is visible if and only if it has passed the pre-write phase, which is when the write has gathered a quorum of corresponding write request messages.

When a user invokes a write, she broadcasts a pre-write request message to all register processes. The pre-write request contains a value and timestamp. Upon receiving a pre-write request message, a process broadcasts the corresponding write request only if the process has not yet sent a write request message for that timestamp. A write is visible exactly when it has generated a quorum of corresponding write request messages, which cannot be forged. A write becomes total when an acknowledged operation returns a set including that write's value-timestamp pair, meaning that a quorum of processes have acknowledged the write.

We now prove that the CL99's read operation satisfies the Paxos register's consistency semantics.

Lemma 5. *Every read only returns the values of visible writes.*

Proof:

1. Consider a read rd that returns v and ts .
2. rd returns v because of a read acknowledgement.
3. That read acknowledgement contains a quorum of write request messages vouching for a write of v .
4. By definition, that write is visible.

Lemma 6. *Every read only returns the values of earlier or concurrent writes.*

Proof: See proof of Lemma 2.

Lemma 7. *Every read only returns a value written by the last total write or by a write since the last total one.*

Proof:

1. Consider a total write wr_t and a read rd later than wr_t in the partial order.
2. Let ts_t be the timestamp of wr_t .
3. Let rd return v_r and $ts_r \geq ts_t$.
4. Assume that no write since and including wr_t has written v_r .
5. rd returns v_r and ts_r because of a quorum of unforgeable read acknowledgements.
6. At least one of the read acknowledgements contains a *lastWrite* field with timestamp at least ts_t .
7. In order for rd to return v_r , at least one of the read acknowledgements' *lastWrite* field vouches for a write(v_r, ts), where $ts \geq ts_t$.
8. Contradiction. No write since and including wr_t has written v_r .

Together, Lemmas 6 and 7 prove that the CL99 register has Paxos register consistency semantics.

Lemma 8. *Under good conditions, the CL99 register guarantees that a write eventually becomes total.*

The CL99 register defines good conditions as follows. First, the system behaves synchronously. Second, all users continually invoke reads and writes.

Proof:

1. Given the system behaves synchronously.
2. Each user has an infinite number of periods in which to invoke a write.
3. Each period of synchrony for a user to write is longer than the previous period.
4. Since all users continue to read and write to the register, some user will eventually have a sufficiently long window of synchrony in which to write to the register.
5. Such a user can select an appropriate timestamp to write by choosing the same timestamp as the one she just read.

Note that unlike the La98 register, a privileged user is not elected by the users. Instead, the register processes determine the current privileged user. This guards against ill-willed users. \square