Faster Randomized Repeated Choice and DCAS

Dante Bencivenga, George Giakkoupis, Philipp Woelfel

▶ To cite this version:

Dante Bencivenga, George Giakkoupis, Philipp Woelfel. Faster Randomized Repeated Choice and DCAS. PODC 2024 - 43rd ACM Symposium on Principles of Distributed Computing, Jun 2024, Nantes, France. pp.454-464, 10.1145/3662158.3662828 . hal-04896495

HAL Id: hal-04896495
https://inria.hal.science/hal-04896495v1

Submitted on 20 Jan 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Faster Randomized Repeated Choice and DCAS

DANTE BENCIVENGA, University of Calgary, Canada
GEORGE GIAKKOUPIS, Inria, Univ Rennes, CNRS, IRISA, France
PHILIPP WOELFEL, University of Calgary, Canada

At STOC 2021, Giakkoupis, Giv, and Woelfel [9], presented an efficient randomized implementation of Double Compare-And-Swap (DCAS) from Compare-And-Swap (CAS) objects. DCAS is a useful and fundamental synchronization primitive for shared memory systems, which, contrary to CAS, is not available in hardware. The DCAS algorithm has $O(\log n)$ expected amortized step complexity against an oblivious adversary, where n is the number of processes in the system. The bottleneck of this algorithm is a building block, introduced in the same paper: A *repeated choice (RC) object*, which allows processes to propose values, and later agree on (and "lock in") one of the proposed values, which is roughly uniformly distributed among the "recently" proposed ones. The object can then be unlocked, and the process be repeated.

The RC implementation introduced by Giakkoupis et al. has step complexity $O(\log n)$. In this paper, we present a more efficient RC algorithm, with similar probabilistic guarantees, but expected step complexity $O(\log \log n)$. We then show how this improved RC object can be used to achieve an exponential improvement in the expected amortized step complexity of DCAS.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

Additional Key Words and Phrases: Shared memory, Double-Compare-And-Swap, DCAS, Randomized algorithms, Repeated choice

Contents

Abstract 1
Contents 1
1 Introduction 2
1.1 Preliminaries 4
1.2 DCAS and BDCAS 5
2 Repeated Choice 5
2.1 Implementation 6
2.2 Detailed Algorithm Description 8
2.2.1 Propose Method 8
2.2.2 Choose&Lock Method 9
2.2.3 Unlock Method 10
2.2.4 Clear Method 11
2.3 Correctness 11
2.3.1 Linearization Points 12
2.3.2 Correctness of `choose&lock()` and `unlock()` 13
2.3.3 Pivot Properties 14
2.3.4 Clear Method Properties 17
2.3.5 ABA-Freedom 20
2.3.6 Linearizability 21
2.4 Distribution of Agreement-Value 22
2.4.1 Proof of Theorem 2.27 24

Authors' Contact Information: Dante Bencivenga, University of Calgary, Calgary, Canada, drbenciv@ucalgary.ca; George Giakkoupis, Inria, Univ Rennes, CNRS, IRISA, Rennes, France, george.giakkoupis@inria.fr; Philipp Woelfel, University of Calgary, Calgary, Canada, woelfel@ucalgary.ca.

2.4.2 Proof of Theorem 2.28 29
2.5 Step Complexity 32
2.5.1 Clear Method 32
2.5.2 Unlock Method 36
2.5.3 Solo Execution 40
3 Bipartite DCAS 40
3.1 Implementation 41
3.2 High Level Idea 41
3.3 Detailed Algorithm Description 44
3.3.1 Read Method 44
3.3.2 Finish Method 44
3.3.3 BDCAS Method 44
3.4 BDCAS Correctness Statements 45
3.5 Step Complexity Preliminaries 46
3.5.1 Doomed and Successful Tasks 46
3.5.2 Strengthening the Adversary: BDCAS Traces 47
3.5.3 Extending E to Executions E' and E'' 48
3.5.4 Other Notation 48
3.6 Distribution of Agreement-Value τ_k 49
3.7 k-Fated Tasks 52
3.8 Lower Bounding the Sum of k-Fated Tasks 55
3.9 Upper Bounding the Sum of k-Fated Tasks 57
3.10 Completing the Proof of Theorem 3.25 58
3.11 Auxiliary Lemmas 58
4 Conclusion 66
Acknowledgments 66
References 66

1 INTRODUCTION

A double compare-and-swap (DCAS) object extends the standard compare-and-swap (CAS) operation to two memory locations. A `DCAS()` operation compares in one atomic step two given independent memory locations with two given 'old' values, and writes to both memory locations two given 'new' values if and only if the comparison yields a complete match. DCAS is an extremely useful and fundamental primitive that has numerous applications [2, 6, 12, 19, 23]. Unfortunately, it is generally not available on hardware (except for the highly inefficient implementation on Motorola's 68k series microprocessor [13]).

Over the years, many deterministic DCAS implementations have been developed using only synchronization primitives that are readily available on most common architectures, most prominently CAS [1, 4, 5, 8, 14–16, 18, 21, 24–26]. Most of these implementations are lock-free, and few are wait-free [1, 4, 8, 26]. Some implementations are efficient in the case of low or no contention between processes attempting a DCAS on overlapping memory locations [1, 5, 14, 21]. In the general case with no limit on contention, all implementations published prior to 2021 have a forbiddingly high (amortized or individual) step complexity of at least $\Omega(n)$.

Recently, Giakkoupis, Giv, and Woelfel [9] presented an efficient randomized DCAS algorithm with expected amortized step complexity of $O(\log n)$ from registers and CAS objects. It uses as building blocks a so-called *repeated choice* (RC) object, and a *bipartite double compare-and-swap* (BDCAS) primitive. BDCAS is the same

as DCAS, except that the applicable memory locations are partitioned into two parts, and if two locations are affected by an operation, they must be from different parts.

An RC object provides the operations `read()`, `choose&lock()`, `propose(`v`)` and `unlock(`u`)`, where v is from some domain V not containing $\bot$, and $u \in V \cup \{\bot\}$. The object stores a value in $V \cup \{\bot\}$, and operation `read()` simply returns that value. A process can *propose* a value $v \in V$ by calling `propose(`v`)`, which does not affect the value of the object and does not return anything. When a process calls `choose&lock()`, the value of the object is set to a randomly chosen, previously proposed value, or to $\bot$. After that, the value of the object cannot change anymore until it is unlocked with an `unlock(`u`)` call, where u must match the current value of the object. The benefit of this object comes from its probabilistic guarantees. Roughly, the value chosen in a `choose&lock()` call is likely to be one of the *recently* proposed values (meaning it has been proposed no earlier than two `choose&lock()` calls ago). Moreover, each specific value is chosen only with small probability, which, in [9], is roughly at most inversely proportional to the number of recently proposed values.

Thus, an RC object can be used as a form of probabilistic consensus, where the value that processes agree on is one of the proposed ones, but chosen at random instead of adversarially. The locking semantics allows processes to execute such a consensus protocol repeatedly.

Giakkoupis et al.'s RC implementation uses two arrays, each of size $\Theta(\log n)$, and in a `propose(`v`)` method a process writes value v into a random location in one of those arrays. The choice among the two arrays is uniformly random, and the position in the array is chosen according to a geometric distribution (i.e., position i is chosen with probability $\Theta(1/2^i)$). A linear search for the highest written position is employed to find a value that the processes then agree on in a `choose&lock()` call. This linear search, and the fact that in each `unlock()` call processes have to clear one of the two arrays, requires each process to take $\Theta(\log n)$ steps in each successful `choose&lock()` and `unlock()` call. This is also the bottleneck of the entire DCAS algorithm—everything else in the entire construction needs only amortized constant RC method calls and other shared memory steps in expectation.

In this paper, we reduce the step complexity of the RC object to expected amortized $O(\log\log n)$ (without increasing the logarithmic worst-case complexity). The probabilistic guarantees are technically slightly weaker, but there is no practical difference for applications. The probability that a specific value is chosen at random in a `choose&lock()` call is now bounded by a function that is inversely proportional to a random variable whose expectation is linear in the number of recently proposed values. Moreover, with small poly-logarithmic probability (in n), processes may agree on proposed values that are outdated, i.e, they were not proposed recently. This requires a new analysis of the BDCAS algorithm, but nothing changes for the DCAS algorithm. As a corollary, we obtain a DCAS object with expected amortized step complexity $O(\log\log n)$, which is an exponential improvement over [9].

Probabilistic shared memory algorithms are analyzed under the assumption of an adversary that knows the algorithm and schedules processes on-line, possibly using information about the current system state to decide which process takes the next shared memory step. The adversary also decides which implemented method the selected process calls and what arguments to use, if the process has no such pending method call. The result of Giakkoupis et al. was stated for an oblivious adversary, which schedules processes without any information about the system state (but internally, the analysis used a slightly stronger adversary, tailored to the proofs). For our analysis we define the *ip-aware adversary* (for "instruction pointer"-aware), which is stronger than the oblivious one in the sense that the adversary always knows the line in the program code containing the shared memory operation that each process has most recently executed, as well as the return values of all previous method calls invoked by the adversary. It does not know the return values of method calls made from within the implemented methods.

Corollary 1.1. *There is a randomized linearizable DCAS implementation from registers and CAS objects, where each* `read()` *operation has constant step complexity, and in an execution that is scheduled by an ip-aware adversary and that comprises ℓ* `DCAS()` *and* `read()` *invocations, the expected total number of steps is in $O(\ell \cdot \log\log n)$.*

Our new RC object employs two techniques: First, in method `choose&lock()` we add a binary search before a shorter linear search for the highest written position in the array. Second, in method `unlock()` we only clear the parts of the array that have likely been written to, and for that we use a randomized algorithm, which distributes the work over participating processes. For this purpose we devise a reusable and simplified version of the certified write-all (CWA) algorithm by Martel and Subramonian [22]. The binary search and the partial clearing of the array create technical challenges that are surprisingly difficult to deal with. For example, it is possible that when a number of processes write their proposed values to the array, they leave "holes", i.e., the array positions containing proposed values are not consecutive. The binary search may then return a position that is not the highest written one, and as a result, some high values may not get cleared from the array in the next `unlock()` call. Thus, some "outdated" proposals can be left behind. As a result, our RC algorithm and its randomized analysis become much more involved.

We remark that the resulting DCAS algorithm works under the same assumptions as the one from [9]: It requires unbounded sequence numbers and assumes that it is possible to allocate a constant number of CAS objects and registers with each `DCAS()` operation. (The total number of referenced objects is always bounded, and there are well-known techniques that can be used for bounding sequence numbers and for memory reclamation [3].) Finally, it is assumed that every `DCAS()` operation changes the value of both memory locations if there is a match, and not just one of them.

1.1 Preliminaries

We consider the standard asynchronous shared memory model in which n processes with unique IDs in $\{0, \ldots, n-1\}$ communicate through atomic operations on shared objects. Our algorithms use only atomic registers and compare-and-swap (CAS) objects.

A CAS object supports read and write operations, as well as `CAS`(old, new), which compares the value of the object to *old*, and if it matches, writes *new* to the object. The operation returns true and we say it is *successful* if there was a match; otherwise it returns false and we say it is *unsuccessful.*

Processes can generate private coin flips to make random decisions. Each process runs its own algorithm. An adversary generates a schedule by deciding, at each point in time, which process takes the next step, and which method call and arguments to use if the process does not have a pending next step. We assume an *ip-aware* adversary, which is described in Definition 2.24. The schedule, together with the processes' random choices, gives rise to an *execution*, which is simply the sequence of all steps taken by processes.

The standard correctness condition for shared memory objects is *linearizability* [17]. An object is linearizable, if for any execution, each of the object's complete and possibly some of its incomplete operations can be assigned a *linearization point* that occurs not before the operation's invocation and not after its response, such that the sequential execution obtained by ordering all operations by their linearization points is consistent with the object's sequential specification. That sequential execution is called *linearization* of the original execution on the object. An object is *strongly linearizable* [11], if each execution E on the object can be mapped to a linearization $lin(E)$, such that for any two executions E' and E'', $lin(E')$ is a prefix of $lin(E'')$ whenever E' is a prefix of E''. (This definition applies also to randomized objects.)

We assume without loss of generality that all shared memory operations of an execution E take place at integer time units; precisely, the k-th shared memory operation takes place at point k.

We use subscript t to denote the value of a shared variable at point t. If at t an operation is executed that changes the value of a shared variable y from a to $b \neq a$ (namely, a write or `CAS()` operation), then we define $y_t = b$.

1.2 DCAS and BDCAS

A double compare-and-swap (DCAS) object applies operations to an array $D[0 \ldots m-1]$. It supports read operations on elements of D as well as the $\texttt{DCAS}(\langle add_0, old_0, new_0\rangle, \langle add_1, old_1, new_1\rangle)$ operation, which writes new_i to $D[add_i]$ for both $i \in \{0,1\}$, if and only if immediately before the operation $D[add_j] = old_j$ for both $j \in \{0,1\}$. We also require $old_i \neq new_i$ for $i \in \{0,1\}$. Similar to `CAS()`, a `DCAS()` operation returns true if both old values matched, and false otherwise.

A BDCAS object assumes that the set $\{0, \ldots, m-1\}$ of addresses has a predefined partition into two subsets M_0 and M_1 such that $M_0 \cup M_1 = \{0, \ldots, m-1\}$ and $M_0 \cap M_1 = \emptyset$. It supports the operation $\texttt{BDCAS}(\langle add_0, old_0, new_0\rangle, \langle add_1, old_1, new_1\rangle)$, which is the same as `DCAS()` except that it requires $add_i \in M_i$ for both $i \in \{0,1\}$ and does not return a value. It also supports read operations.

We say that a `DCAS()` or `BDCAS()` operation is *successful* if the old values both match, and it is *unsuccessful*, otherwise.

2 REPEATED CHOICE

In this section, we describe the *repeated choice* (RC) type and a randomized implementation of it.

An RC object allows processes to *propose* values, and then to agree on a randomly chosen *agreement-value* among the proposed values. In order to allow this to happen repeatedly, an agreement-value is first *locked*, until some process *unlocks* that agreement-value.

In the following we give a sequential specification of RC. It is trivial to implement a linearizable object with these properties. However, our implementation also provides a non-trivial probabilistic property that is not part of the sequential specification, and which we describe later.

Let V be a set and $\perp$ a value with $\perp \notin V$. An RC object R stores an *agreement-value* in $V \cup \{\perp\}$, and it can be in one of two *locking states*, *locked* or *unlocked*. Initially, R's agreement-value is $\perp$ and R is unlocked. The object supports the methods $\texttt{propose}(v)$ for $v \in V$, `choose&lock()`, $\texttt{unlock}(u)$ for $u \in V \cup \{\perp\}$, and `read()`. We say a process *proposes* v when it calls $R.\texttt{propose}(v)$. We require that throughout an execution, no value gets proposed twice, i.e., if there are two method calls $R.\texttt{propose}(v)$ and $R.\texttt{propose}(v')$, then $v \neq v'$. This can easily be achieved by augmenting each proposed value with a pair comprising the proposer's process ID and a local sequence number.

Method $\texttt{propose}(v)$ does not change the agreement-value of R or its locking state, and it does not return anything; its semantics are defined through `choose&lock()` calls.

Method `choose&lock()` does not modify R if R is locked; in this case, we say the `choose&lock()` call is *unsuccessful*. If R is unlocked, then the method locks R and replaces the old agreement-value of R with some $v \in V \cup \{\perp\}$ such that either

(i) $v = \perp$, or
(ii) v was previously proposed and is different from all previous agreement-values of R.

We say the `choose&lock()` call is *successful* when it locks R and replaces the agreement-value as described above.

Method $\texttt{unlock}(v)$ does not modify R if R is unlocked or R's agreement-value is not v, in which case we say the call is *unsuccessful*. If R is locked and its agreement-value is v, then $\texttt{unlock}(v)$ unlocks R without changing its agreement-value, and we say the call is *successful*.

Methods `choose&lock()` and `unlock()` do not return anything.

Finally, method `read()` just returns the agreement-value of R.

Probabilistic Property Overview. Our RC implementation satisfies a probabilistic property which can be informally described as follows. Suppose that $CL_1, UL_1, CL_2, UL_2, CL_3$ are consecutive alternating successful `choose&lock()` and `unlock()` operations. A proposal is *recent* with respect to CL_3 if it was proposed after the linearization point of CL_1, and is *outdated* otherwise. The probabilistic property of our RC implementation approximates the ideal property that the agreement value chosen by CL_3 is uniformly random among all recent proposals. More concretely, there is a random variable g whose expectation is linear in the number of values proposed between the linearization points of UL_1 and UL_2 (this is a subset of all recent proposals). For any recent proposal, the probability that it is chosen is upper bounded by $O(1/g)$. There is also a non-zero probability that $\bot$ is chosen, which is exponentially small in g. Finally, there is an $o(1)$ probability that an outdated value is chosen, as long as a low-probability 'bad' event does not occur. This event depends on the last $O(n \log n)$ proposals, and occurs with probability $1/n^{1+\Omega(1)}$.

The formal description of this probabilistic property is given in Theorems 2.27 and 2.28.

The implementation has $O(\log \log n)$ expected amortized step complexity, and $O(\log n)$ worst-case step complexity per operation.

2.1 Implementation

In Figures 1–3, we present a randomized strongly linearizable [11] implementation of an RC object.

First, we define $\phi = \lceil \log \log n \rceil$, and $\lambda = 2^k(\phi - 1)$, where k is the unique integer satisfying $\log n \leq \lambda < 2 \log n$. We use a two-dimensional array $C[i][j]$, where $i \in \{0, 1\}$ and $j \in \{1, \ldots, \lambda\}$. We refer to arrays $C[0]$ and $C[1]$ as the two *sides* of C. To propose a value v in a `propose(`v`)` call, a process simply writes v to an array entry $C[\alpha][\beta]$, where α is chosen uniformly at random and β is (approximately) geometrically distributed.

The agreement-value and locking state of the RC object are interpreted from the contents of a CAS object S. This object has three components, $S.val$, $S.i$, and $S.\ell$.

The first component, $S.val$, stores the interpreted agreement-value of R. The third component, $S.\ell$, is a sequence number that gets incremented with every successful `choose&lock()` or `unlock()` call. The interpreted locking state of R is unlocked if $S.\ell$ is even, and locked if $S.\ell$ is odd. Finally, $S.i$ is a bit that describes the side of C that `choose&lock()` calls read to determine the interpreted agreement-value, and that `unlock()` calls erase before unlocking the object (as described later). The value of $S.i$ is flipped after each successful `unlock()` call, and always equals the second-least significant bit of $S.\ell$; we include it as a separate component to make the code easier to read.

The `read()` implementation is straightforward: A process simply returns the value of $S.val$.

When a process p calls `choose&lock()`, it reads $(S.val, S.i, S.\ell)$ from S in line 3. If R is unlocked (i.e., $S.\ell$ is even), then p performs a binary search followed by a linear scan, both in $O(\log \log n)$ time, over the array $C[S.i]$. The search roughly tries to find the non-$\bot$ value in $C[S.i]$ with the largest index (lines 5–14), under the assumption that if $C[S.i][j] = \bot$, then $C[S.i][j'] = \bot$ for all $j' > j$. While this assumption does not necessarily hold, the two-stage search is guaranteed to find a value stored in $C[S.i][j]$ such that either $j = \lambda$ or $C[S.i][j+1] = \bot$ at some point during the `choose&lock()` call. If it finds no non-$\bot$ entry then the search returns $\bot$. Also, if the search returns $\bot$, then $C[S.i][j] = \bot$ at some point during the call for all $1 \leq j \leq \phi$. In line 15, p tries to lock R with the value found in the binary search, using a `CAS()` operation on S.

A value proposed at point r is *recent* at point $t \geq r$ if at most one successful `choose&lock()` operation linearizes in the interval $(r, t]$. Because of the geometric distribution used in `propose()` calls, each recent value written to $C[S.i]$ is roughly equally likely the new interpreted agreement-value chosen by p. Outdated values (i.e., non-recent proposed values) are very likely overwritten before p's `choose&lock()` call (as discussed next).

An `unlock(`val`)` call by process p begins by reading S, and checking that the object is locked (i.e., $S.\ell$ is odd) and that its interpreted agreement-value is equal to the parameter val. Next, p checks the special "pivot value"

Shared Data and Global Constants:

- Let $\phi = \lceil \log \log n \rceil$ and $\lambda = 2^k(\phi - 1)$, where k is the unique integer satisfying $\log n \leq 2^k(\phi - 1) < 2 \log n$
- $C[i][j]$, for $i \in \{0, 1\}$ and $j \in \{1, \ldots, \lambda\}$, is a CAS object storing a value from $V \cup \{\perp\}$, and is initially $\perp$
- S is a CAS object storing a triple from $(V \cup \{\perp\}) \times \{0, 1\} \times \mathbb{N}$, and is initially $(\perp, 0, 0)$
- P is a CAS object storing a pair from $(V \cup \{\perp\}) \times \mathbb{N}$, and is initially $(\perp, 0)$

```
Method propose(v ∈ V)
   // Requirement: no two calls use the same argument v
1  Choose α ∈ {0,1} uniformly at random, and β ∈ {1,...,λ} independently at random such that
     Pr[β = j] = π_j, where π_j = 2^{-j} for j ∈ {1,...,λ-1} and π_λ = 2^{-λ+1}
2  C[α][β] ← v
```

```
Method choose&lock()
3  (val, i, ℓ) ← S
   // Check if object is already locked
4  if ℓ mod 2 = 1 then return
5  (low, high, v) ← (1, λ+1, ⊥)
   // Binary search
6  while high − low > φ − 1 do
7  |  mid ← (low + high)/2
8  |  v′ ← C[i][mid]
9  |  if v′ = ⊥ then high ← mid else (low, v) ← (mid, v′)
10 v′ ← C[i][1]
11 if C[i][φ] = ⊥ or low = 1 then (low, high, v) ← (1, φ, v′)
   // Linear scan
12 for j ← low + 1, ..., high − 1 do
13 |  v′ ← C[i][j]
14 |  if v′ ≠ ⊥ then v ← v′
   // Linearization point if successful
15 S.CAS((val, i, ℓ), (v, i, ℓ + 1))
```

Fig. 1. propose() and choose&lock() methods of faster repeated choice

stored in $C[S.i][\phi]$, which determines whether processes should clear all of $C[S.i]$ or just $C[S.i][1, \ldots, 5\phi]$. The CAS object P, consisting of a value from $V \cup \{\perp\}$ and a sequence number, enables the processes to agree on the pivot value. Process p attempts to erase the pivot value from C in line 21, then clears all the array entries in $C[S.i][j]$ for $j \in \{1, \ldots, 5\phi\} \setminus \{\phi\}$, by writing $\perp$ to them in lines 22–25.

If the agreed-upon pivot value is non-$\perp$ or $S.\ell$ is close to a multiple of λ, all active `unlock()` calls collectively erase the rest of $C[S.i]$. This clearing uses a reusable and simplified version of the certified write-all (CWA) algorithm by Martel and Subramonian [22]. We call this modified algorithm `clear()` and present it in Figure 3. Processes executing `clear(ℓ)` randomly choose positions in $C[S.i][5\phi, \ldots, \lambda]$ to clear. They keep track of progress using a balanced binary tree containing sequence numbers. Each node in the progress tree corresponds to a position in $C[S.i][5\phi, \ldots, \lambda]$, and processes update the sequence number of the node corresponding to their chosen position with the parameter of `clear(ℓ)` if both children are set to at least ℓ. Once the root is set to ℓ,

Method `unlock`(val)

```
16 (val′, i, ℓ) ← S
   // Check if object already unlocked or wrong val
17 if ℓ mod 2 = 0 or val ≠ val′ then return
18 pval ← C[i][φ]
   // Pivot-lock point if successful
19 P.CAS((⊥, ℓ − 1), (pval, ℓ))
20 (pval, ·) ← P
   // Clear pivot
21 C[i][φ].CAS(pval, ⊥)
   // Clear remaining values up to C[i][5φ]
22 for j ← 1, . . . , φ − 1, φ + 1, . . . , 5φ do
23 |  v ← C[i][j]
24 |  if S ≠ (val, i, ℓ) then return
25 |  C[i][j].CAS(v, ⊥)
   // Clear values above C[i][5φ]
26 if pval ≠ ⊥ or ℓ mod λ ≥ λ − 4 then clear(ℓ)
27 P.CAS((pval, ℓ), (⊥, ℓ + 1))
   // Linearization point if successful
28 S.CAS((val, i, ℓ), (val, 1 − i, ℓ + 1))
```

Method `read()`

```
29 (val, ·, ·) ← S
30 return val
```

Fig. 2. `unlock()` and `read()` methods of faster repeated choice

all processes running `clear(`ℓ`)` can return, and the progress tree can be re-used to track future invocations of `clear()` with larger parameters.

Both `unlock()` and `clear()` ensure that no value in $C[S.i]$ gets erased if it is proposed after the value of S changes, i.e., once the object gets unlocked by some other process. Over two consecutive successful `unlock()` calls, both sides of C get erased up to $C[S.i][5\phi]$ at least once, and so no outdated values in $C[S.i][1,\dots,5\phi]$ can be chosen anymore. Outdated values with positions above 5ϕ can only be chosen with a small probability since they require $C[S.i][\phi]$ to stay at $\perp$.

Note that if a process proposes a value, it is possible that the value gets erased immediately by an ongoing `unlock()` call. However, if the proposing process chooses side $1 - S.i$ to write to, then this won't happen. Since processes choose their side at random, they have at least a 1/2 probability of choosing a side that does not get erased before the process has a chance of being chosen.

2.2 Detailed Algorithm Description

2.2.1 Propose Method. In line 1 of a `propose(`v`)` call, the calling process p chooses $\alpha \in \{0, 1\}$ uniformly at random, and $\beta \in \{1, \dots, \lambda\}$ from an approximately geometric distribution. Then, p writes the proposed value v to $C[\alpha][\beta]$ in line 2.

Because the `unlock()` method, described in more detail below, erases values in C before unlocking the object, it is possible for the proposed value to be erased immediately after it is written to C. The adversary could use this

Shared Data and Global Constants:

- Let $h = \lambda - 5\phi$ (this is the size of the sub-array $C[i][5\phi+1,\ldots,\lambda]$ to be cleared)
- $T[j]$, for $j \in \{1,\ldots,h+1\}$, is an array of CAS objects which keeps track of progress in clearing $C[i]$ in a balanced binary tree structure (except for merging all leaves into a single common leaf). $T[j]$ is initialized to 0 for $j \in \{1,\ldots,h\}$ and to ∞ for $j = h+1$.

We require the following:

If `clear`(ℓ_1) and `clear`(ℓ_2) are both invoked in an execution with $\ell_1 < \ell_2$, then the first invocation of `clear`(ℓ_2) occurs after the first response of any `clear`(ℓ_1) operation. (1)

Method `clear`(ℓ)

```
   // Infer the side of C from ℓ
31 i ← ⌊ℓ/2⌋ mod 2
   // Efficiently clear C[i][j] for j ∈ {5φ+1,...,λ}
32 while T[1] < ℓ do
33     Choose j ∈ {1,h} uniformly at random
34     node ← T[j]
35     left ← T[min{2j, h+1}]
36     right ← T[min{2j+1, h+1}]
37     v ← C[i][j+5φ]
38     if T[1] ≥ ℓ then return
39     C[i][j+5φ].CAS(v, ⊥)
       // Propagate ℓ up the tree
40     T[j].CAS(node, min{ℓ, left, right})
```

Fig. 3. Efficient clearing method for faster repeated choice

to prevent a proposal from being selected, or boost the probability of choosing a different proposal. This motivates the two sides of C and the uniformly random choice of $\alpha \in \{0,1\}$: with probability 1/2, the proposed value survives until the next `choose&lock()` operation can (possibly) choose it as the next interpreted agreement-value. The approximately geometric distribution of β reduces the advantage of later proposals to be chosen, since they can overwrite earlier proposals. This limits the degree to which the adversary can increase the probability of any given proposal to be selected as an interpreted agreement-value.

2.2.2 Choose&Lock Method. A process p executing the method `choose&lock()` first reads the three fields of S in line 3. If $S.\ell$ is odd, then the interpreted locking state of R is locked, and so p immediately returns in line 4.

Since the `propose()` method chooses indices j of $C[i][j]$ according to an approximately geometric distribution, it is likely that if $C[i][j] = \perp$, then $C[i][j+1] = \perp$. Therefore, the `choose&lock()` method uses a binary search followed by a linear scan to approximately determine the non-$\perp$ value in $C[i]$ with the largest index j. Note that this does not guarantee that it will find the actual non-$\perp$ value with the largest index j. Instead, the method guarantees that it finds some non-$\perp$ value in $C[i][j]$ such that either $j = \lambda$ or $C[i][j+1] = \perp$ at some point during the method call. (The analysis only requires this weaker guarantee.) In line 5, p initializes the parameters for the binary search. The *low* and *high* variables store the smallest (inclusive) and largest (exclusive) indices for the binary search, initialized to 1 and $\lambda+1$, respectively, to cover the full range of indices $[1,\lambda]$ for $C[i]$. Variable

v is initialized to $\bot$, and is subsequently used to store the non-$\bot$ value with the highest index j found so far in the binary search.

The while-loop in lines 6–9 performs the binary search. The while-condition of the binary search in line 6 continues the binary search as long as the range has size more than $\phi - 1$. Since $\lambda = 2^k(\phi - 1)$ for an integer k, this ensures that it ends with a range of size exactly $\phi - 1$. Line 7 chooses the middle index *mid* halfway between *low* and *high*, and our choice of λ ensures that *mid* is always an integer. In line 8, p reads the value in $C[i][mid]$ to determine which half of the remaining range to search next. In line 9, p updates *high* or *low* accordingly, and in the case of $v' \neq \bot$ it updates v to v', since v' is the non-$\bot$ value with the largest index j found so far.

After the binary search, the interval $[low, high)$ has size $\phi - 1 \in \Theta(\log\log n)$, so p can perform a linear scan to find the highest non-$\bot$ value in the range and keep the step complexity at $O(\log\log n)$. In line 11, p checks the special pivot value $C[i][\phi]$; the `unlock()` method later clears the larger indices of $C[i]$ if $C[i][\phi] \neq \bot$. If p reads $\bot$ from $C[i][\phi]$, then p changes the linear scan interval to $[1, \phi)$ and sets v to the value in $C[i][1]$ that it reads in line 10. This ensures that the eventual interpreted agreement-value is erased by the next successful `unlock()` operation, as required by the specification. (This is the primary motivation for having a linear scan after the binary search, as it allows p to only check lower indices within $C[i]$ without revealing anything about the eventual index to the ip-aware adversary.) In all cases, immediately after line 11, v stores a value that p reads from $C[i][low]$. The linear scan occurs during the for-loop in lines 12–14, in which p updates v to the last non-$\bot$ value it reads in $C[i][low + 1, \ldots, high - 1]$ (or keeps v at the value from $C[i][low]$ if it only reads $\bot$ in the linear scan).

The method then concludes by attempting a `CAS()` on S in line 15. If successful, the `CAS()` changes the interpreted agreement-value to the final value of v, leaves the side of C the same (since `unlock()` then needs to clear it), and increments the sequence number.

2.2.3 Unlock Method. A process p executing `unlock()` begins similarly to a process executing `choose&lock()`, by reading S in line 16. If the sequence number $S.\ell$ is even, then the interpreted locking state is unlocked, so the method immediately returns in line 17. Similarly, if the argument *val* does not match the interpreted agreement-value $S.val$, the method immediately returns in line 17.

In the next part of `unlock()`, processes agree on whether to clear all of $C[i]$ or just the smallest indices, i.e., $C[i][1, \ldots, 5\phi]$. They make the decision by reading the pivot value $pval = C[i][\phi]$ in line 18, then attempting a `CAS()` on the object P in line 19 from $(\bot, \ell - 1)$ to $(pval, \ell)$. Since P comprises both a value and a sequence number, which follows the sequence number $S.\ell$ of S, `unlock()` operations that fully take place while $S.\ell = \ell^*$ use the same agreed-upon pivot value *pval*. Specifically, they agree on the unique value v such that $P = (v, \ell^*)$ at some point in the execution. The `CAS()` on $C[i][\phi]$ in line 21 attempts to clear $C[i][\phi]$ by replacing the agreed-upon pivot value with $\bot$. Since each lock/unlock cycle has a single agreed-upon pivot value and no two proposals write the same value, $C[i][\phi]$ is only cleared (i.e., replaced with $\bot$) at most once per lock/unlock cycle.

The next phase of `unlock()` clears the remaining small indices j of $C[i]$, i.e., $j \in \{1, \ldots, 5\phi\} \setminus \{\phi\}$. This consists of three steps within a for loop: first, process p reads the value of $C[i][j]$ in line 23, then it checks in line 24 if S matches the value that p read in line 16. If the value of S is different in the two lines, then p immediately returns, to prevent clearing values proposed after another process successfully unlocks the RC object. Otherwise, p attempts a `CAS()` in line 25 on $C[i][j]$ from the value it reads in line 23 to $\bot$. This ensures that all values stored in $C[i][1, \ldots, 5\phi]$ when R was last locked are erased or overwritten once the `unlock()` call returns.

In line 26, if $\ell \bmod \lambda \in \{\lambda - 4, \ldots, \lambda - 1\}$ or $P.pval \neq \bot$, then p invokes the `clear(ℓ)` method, which clears values in $C[i][5\phi + 1, \ldots, \lambda]$. Since overlapping `unlock()` operations which run while $S.\ell$ stays at one value read the value of $P.pval$ in line 20, they all make the same decision in line 26 and use the same argument ℓ. (If p reads $P.pval$ after another process executes line 27 during the same period when R is locked, it will either not run `clear(ℓ)` or run it and immediately exit the while-loop, since clearing already finished.) If the pivot

value is not $\perp$, then it is more likely that larger indices are also not $\perp$ and hence should be cleared. The check of $\ell \bmod \lambda \geq \lambda - 4$ ensures that both sides of C are fully cleared periodically, regardless of the pivot value. As we elaborate on below, all values present in $C[i][j]$ for $j \in \{5\phi + 1, \ldots, \lambda\}$ at the beginning of the first `clear(`ℓ`)` invocation are no longer present in $C[i]$ by the first response of a `clear(`ℓ`)` operation for the same argument ℓ.

Finally, the `CAS()` on P in line 27 increments the sequence number of P and replaces the pivot value with $\perp$ if successful, to indicate that `clear(`ℓ`)` is complete and does not need additional invocations. Line 28 then updates S by incrementing the sequence number and switching the side of C for the next lock/unlock cycle, while keeping the interpreted agreement-value unchanged.

2.2.4 Clear Method. A process p running `clear(`ℓ`)` first sets i to $\lfloor S.\ell/2 \rfloor \bmod 2$ in line 31, since it is always the case that $S.i = \lfloor S.\ell/2 \rfloor \bmod 2$. Next, in line 32, p checks that the value stored in $T[1]$, the root of the progress tree, is less than ℓ, and returns otherwise. (If the value in $T[1]$ is at least ℓ, this indicates that all `clear(`ℓ`)` operations can return.) In line 33, p chooses a random position to clear, among the $h = \lambda - 5\phi$ positions to clear. In line 34, p reads the value $T[h]$ of the progress tree corresponding to h, and in lines 35 and 36, p reads the values in the two children of $T[h]$, defined as $T[\min\{2j, h+1\}]$ and $T[\min\{2j+1, h+1\}]$, respectively. (This makes T almost a balanced binary tree, except that all its children are merged into a single node $T[h+1]$.) Lines 37–39 of `clear()` have a similar role to lines 23–25 of `unlock()`, respectively. Line 37 sets variable v to the value in $C[i][j + 5\phi]$, line 38 stops p from clearing values proposed after `clear(`ℓ`)` completes, and the `CAS()` in line 39 ensures that v is no longer present in C.

The final line, line 40, updates $T[j]$ to ℓ if both its children store values of at least ℓ. This has two purposes: first, it indicates that index j has been processed, and in particular that p attempted a `CAS()` on $C[i][j + 5\phi]$ to $\perp$. Second, because of the requirement of both children's values being set to at least ℓ first, $T[j] = \ell$ indicates that all nodes associated with the subtree rooted at $T[j]$ have been processed as well. Since the leaf node $T[h+1]$ is set to ∞ and never changes, this propagation of ℓ up the tree towards the root ensures that all indices $j \in \{1, \ldots, h\}$ are chosen in while-loop iterations before $T[1]$ is updated to ℓ. Hence, by the time the first `clear(`ℓ`)` operation returns, all values in $C[i][5\phi + 1, \ldots, \lambda]$ have changed since the first invocation of `clear(`ℓ`)`.

The expected total work by k processes to execute `clear(`ℓ`)` for a given value of ℓ is at most $O((\lambda + k)\log \lambda) = O((\log n + k)\log\log n)$. If there are $k < 2^{\phi} \in \Theta(\log n)$ recently proposed values, then the pivot value is non-$\perp$ with probability at most $k/2^{\phi} = O(k/\log n)$. Also, the periodically forced clearing when $S.\ell$ is near a multiple of λ occurs twice every approximately $\log n$ lock/unlock cycles. Together, these make the expected amortized complexity of `clear()` in $O(\log\log n)$. To achieve $O(\log n)$ worst-case complexity, the while-loop can alternate with iterations of lines 37–39 for $j \in \{1, \ldots, h\}$ ($h = \lambda - 5\phi$), but we do not include this in the code as it would distract from the main results.

2.3 Correctness

Let R be an RC object and E an execution on R. (An execution consists of a sequence of all steps taken by processes, together with all coin flips and local and shared variables.) We assume without loss of generality that all shared memory operations of an execution E take place at integer time units; precisely, the n-th shared memory operation takes place at point n. In particular, operations are invoked at the point of their first shared memory operation and respond at the point of their last shared memory operation.

We use the standard point operator '.' to access the fields of an object and the methods it supports, e.g., $o.x$ is field x of object o, and $o.f()$ is o's method $f()$. We will also consider *references* to objects, and use operator '.' to access the fields and methods of the referenced object, as well. E.g., if r is a reference to the object o above, then $r.x$ is the same variable as $o.x$.

We use subscript t to denote the value of a shared variable at point t. If at t an operation is executed that changes the value of a shared variable y from a to $b \neq a$ (namely, a write or `CAS()` operation), then we define $y_t = b$.

Two operations op_0 and op_1, where op_0's invocation is before op_1's invocation, *overlap* if and only if either op_0 does not respond or op_0's response is after op_1's invocation.

As with Section 2.1, we denote the three components of S by $S.val$, $S.i$, and $S.\ell$, respectively. The *interpreted agreement-value* of R is the value stored in $S.val$. $S.i$ indicates the side of C that `unlock()` and `choose&lock()` operations use. $S.\ell$ serves as a sequence number for successful `unlock()` and `choose&lock()` operations; if $S.\ell$ is even, the interpreted locking state of R is *unlocked*, and otherwise, it is *locked*. We also denote the two values stored P by $P.pval$ and $P.\ell$, respectively. $P.pval$ records a special *pivot value* used by the `unlock()` method, and $P.\ell$ is a sequence number which we later prove stays within a difference of one from $S.\ell$.

2.3.1 Linearization Points. Let op be one of the operations `propose()`, `choose&lock()`, `unlock()`, or `read()`, and let p be the process executing op. For each such operation we define a function $lin(op)$ during the execution E, where $lin(op)$ is either a point in the execution or is ∞. We will prove in Theorem 2.23 that $lin(op)$ is a strong linearization point when it is not ∞, and otherwise op does not linearize.

If op is a `propose()` operation, then $lin(op)$ is the point when op executes line 2. If op is a `read()` operation, then $lin(op)$ is the point when op executes line 29. Note that both methods consist of a single shared memory operation, and therefore their invocations and responses are this single point.

Next, consider the case where op is a `choose&lock()` [or `unlock()`] operation. If op reads a value from S at its invocation in line 3 [or line 16] that would cause it to return in line 4 [or line 17], then $lin(op)$ occurs at the point of op's invocation. If op's final `CAS()` on S in line line 15 [or line 28] is successful, then $lin(op)$ occurs at the point of this `CAS()`. Otherwise, $lin(op)$ occurs a small ϵ $(0 < \epsilon < 1)$ after the first point after op's read of S in line 3 [or line 16] in which the value of S changes; if no such point exists, then $lin(op) = \infty$.

Observation 2.1. *A* `choose&lock()` *or* `unlock()` *operation op is* successful *if and only if its final* `CAS()` *on S in line 15 or 28, respectively, succeeds at point t, in which case $lin(op) = t$.*

Proof. Suppose that a `choose&lock()` [or `unlock()`] operation op has a successful `CAS()` on S in line 15 [or 28] at point t. Therefore, op does not return early in line 4 [or 17 or 24]. It follows that $S.\ell$ is even [or odd] in op's execution of lines 3 and 15 [or lines 16 and 28], and so the interpreted locking state of R is unlocked [or locked, with the interpreted agreement-value $S.val$ equal to the argument of `unlock()`] at point t. From the sequential specification in Section 2, op is successful at point t. The first argument of op's `CAS()` operation is the value that op read from S in line 3 [or 16], and every successful `CAS()` on S increments $S.\ell$ by one. Therefore, t is also the point of the first shared memory operation that changes S after op's read of S, so $lin(op) = t$ by definition.

It remains to show that if a `choose&lock()` [or `unlock()`] operation does not have a successful `CAS()` on S in line 15 [or line 28], then it is not successful. If a `choose&lock()` operation op returns in line 4, then R's interpreted locking state is locked at $lin(op)$ in line 3, and therefore op is unsuccessful at point t (note that it does not modify S in this case). Similarly, if an `unlock(`val`)` operation op returns in line 17 [or line 24], then either the interpreted agreement-value is not val or R's interpreted locking state is unlocked at $lin(op)$ in line 16 [or line 24], and op does not modify S. The remaining case is that op's `CAS()` on S in its final line fails. Again, since the first argument of op's `CAS()` on S in line 15 [or 28] equals the value that op reads from S in line 3 [or 16] at op's invocation, S changes in between op's invocation and response. Since only a successful `CAS()` in line 15 or line 28 can modify S, by the above paragraph and the definition of lin, $lin(op) = lin(op') + \epsilon$ for some successful `choose&lock()` or `unlock()` operation op'. Since the interpreted locking state of R alternates with each successful `CAS()` on S, R is locked [or unlocked] at $lin(op)$. Therefore, op is unsuccessful. □

Observation 2.2. *If $lin(op) = \infty$, then op does not respond.*

Proof. Let p be the process executing op. Then op is not a `propose()` or `read()` operation, as otherwise $lin(op) < \infty$ by definition.

Now suppose op is a `choose&lock()` operation. For the purpose of proving a contradiction, assume that op responds. By definition of lin, S is unlocked when p reads S in line 3, and so p responds by executing line 15 at the end of op. The value of S changes either with the `CAS()` operation that p executes in line 15, or in between p's read of S in line 3 and before p's response in line 15. Since $lin(op)$ is at or immediately after the first such change of S, this leads to a contradiction.

Finally, suppose that op is an `unlock(`val`)` operation. As above, we assume that op responds, and will arrive at a contradiction. If op responds in line 17, then $lin(op)$ is the point of op's invocation. If op responds in line 24 due to S having changed, then $lin(op)$ is immediately after the first such change. Therefore, op responds in line 28. As in the `choose&lock()` method, either the final `CAS()` succeeds and $lin(op)$ is the point of op's response or S changes before the `CAS()` attempt and $lin(op)$ is immediately after the first such change. Thus, all cases imply that $lin(op) < \infty$, which is a contradiction. □

2.3.2 Correctness of `choose&lock()` and `unlock()`.

Observation 2.3. *If the interpreted agreement-value changes at point t, then a successful* `choose&lock()` *operation responds at point t.*

Proof. Recall that the interpreted agreement-value is defined as the first component of S, $S.val$. Only a successful `CAS()` in line 15 can change $S.val$. By Observation 2.1, a successful `choose&lock()` operation responds at point t. □

Lemma 2.4. *Let $w \in V$ and $t > 0$. If $S_t.val = w$, then there is a* `propose(`w`)` *operation op such that $lin(op) < t$.*

Proof. Initially, $S_0.val = \bot$. From Observation 2.3, some process p's successful `choose&lock()` operation op' executes line 15 at some point $lin(op') \leq t$, which sets $S_{t'}.val$ to w. Therefore, p previously reads w in line 8, 10 or 13 from some entry $C[i][j]$ before t'. Since $w \in V$ and so $w \neq \bot$, some process q writes w to $C[i][j]$ in line 2 before the point of p's read. Since only the `propose()` method can write non-$\bot$ values to C, for q's `propose(`w`)` operation op, $lin(op) < t' < t$. □

Lemma 2.5. *Between the responses of any two successful* `choose&lock()` *operations, a successful* `unlock()` *operation responds, and between the responses of any two successful* `unlock()` *operations, a successful* `choose&lock()` *operation responds.*

Also, the response of the first successful `choose&lock()` *operation happens before the response of the first successful* `unlock()` *operation.*

Proof. According to the conditional in line 4 and the parameters of the `CAS()` operation in line 15, `choose&lock()` can have a successful `CAS()` only if immediately before its `CAS()` $S.\ell$ is even. Similarly, in the `unlock()` method, the `CAS()` at line 28 succeeds only if $S.\ell$ is odd immediately before the `CAS()` operation due to the conditional in line 17.

The claim follows from the fact that initially $S.\ell$ is even (zero), and that each successful `choose&lock()` and `unlock()` operation increments $S.\ell$ at its response by Observation 2.1. □

Observation 2.6. *Let each of op_0 and op_1 be a* `choose&lock()` *or* `unlock()` *operation such that both op_0 and op_1 are successful. Then op_0 and op_1 do not overlap.*

Proof. Suppose for the purpose of proving a contradiction that op_0 and op_1 do overlap. Also suppose without loss of generality that op_0's first shared memory operation (a read of S in line 3 or 16) occurs before op_1's first shared memory operation (another read of S). By definition of overlapping operations, op_0's last shared memory operation (a successful `CAS()` on S) occurs after op_1's first shared memory operation.

Let op_i be the operation with the later successful CAS() among op_0 and op_1. S changes at least once after op_i's read of S in its first step and before its successful CAS() in its last step, at the time of op_{1-i}'s successful CAS() operation. The first argument of op_i's CAS() on S equals the values read from S in op_i's read from S in line 15 or 28, and S exhibits no ABAs due to $S.\ell$ strictly increasing with each successful CAS() operation. Therefore, op_i's final CAS() fails, contradicting that it is a successful choose&lock() or unlock() operation. □

2.3.3 *Pivot Properties.*

Observation 2.7. *If $P.\ell$ is even, then $P.pval = \bot$.*

Proof. Initially, $P = (\bot, 0)$. Only a successful CAS() operation on P in line 19 or 27 may change P, and both lines increment $P.\ell$ by one. Since any process executing unlock() has an odd local value of ℓ in order to progress past line 17, and since line 19 sets $P.\ell$ to ℓ and line 27 sets $P.\ell$ to $\ell + 1$, only line 27 can set $P.\ell$ to an even number. Since line 27 also sets $P.pval$ to $\bot$, the claim follows for all $P.\ell$. □

Recall from Observation 2.1 that a successful choose&lock() or unlock() operation is one whose final CAS() on S, in line 15 or 28 respectively, succeeds. Also, from Observation 2.1, a successful choose&lock() or unlock() operation op has $lin(op)$ exactly when its final CAS() on S succeeds, and no other operation can update S.

Definition 2.8 (Lock/unlock timeline points)**.** Let $OP_{x>t}$ indicate the set of all operations that execute line x after point t. We uniquely define the following points in an execution for all positive integers s. If the point does not exist, then we define it as ∞.

- Point t_s^{CL} occurs at the response of the s-th successful choose&lock() operation.
- Point t_s^{PL} occurs at the first execution of the CAS() in line 19 from an unlock() operation in $OP_{16>t_s^{CL}}$.
- Point t_s^{PC} occurs at the first execution of the CAS() in line 21 from an unlock() operation in $OP_{20>t_s^{PL}}$ such that the operation reads $S_{t_s^{CL}}.i$ from S in line 16.
- Point t_s^{PU} occurs at the first execution of the CAS() in line 27 from an unlock() operation in $OP_{16>t_s^{CL}}$.
- Point t_s^{UL} occurs at the first execution of the CAS() in line 28 from an unlock() operation in $OP_{16>t_s^{CL}}$.

For convenience in the analysis, we also fix $t_0^{CL} = t_0^{PL} = t_0^{PC} = t_0^{PU} = t_0^{UL} = 0$.

Lemma 2.9 (Lock/unlock timeline). *Let s be a positive integer. Then the following properties hold:*

1. *If $t_s^{CL} < \infty$, then t_s^{CL} is the point of the $(2s-1)$-th successful* CAS() *on S, which increments $S.\ell$ from $2s-2$ to $2s-1$. Also, $P_{t_s^{CL}}.\ell = 2s-2$.*
2. *If $t_s^{PL} < \infty$, then the* CAS() *on P at point t_s^{PL} is successful, and increments $P.\ell$ from $2s-2$ to $2s-1$. We will later refer to this point as a* pivot-lock point.
3. *If $t_s^{PC} < \infty$ and $P_{t_s^{PL}}.pval \neq \bot$, then C does not contain $P_{t_s^{PL}}.pval$ at any point at or after t_s^{PC}.*
4. *If $t_s^{PU} < \infty$, then the* CAS() *on P at point t_s^{PU} is successful, and increments $P.\ell$ from $2s-1$ to $2s$.*
5. *If $t_s^{UL} < \infty$, then point t_s^{UL} is the response of the s-th successful* unlock() *operation in line 28, which increments $S.\ell$ from $2s-1$ to $2s$.*
6. *If $t_{s+1}^{CL} < \infty$, then point t_{s+2}^{CL} occurs at the first execution of the* CAS() *in line 15 from a* choose&lock() *operation in $OP_{3>t_s^{UL}}$. This* CAS() *on S is successful, and increments $S.\ell$ from $2s$ to $2s+1$, while $P_{t_{s+2}^{CL}}.\ell = 2s$.*
   In particular, $t_s^{CL} \le t_s^{PL} \le t_s^{PC} \le t_s^{PU} \le t_s^{UL} \le t_{s+1}^{CL}$, using the convention that $\infty \le \infty$.

Proof. First, observe that $S.\ell$ begins at zero and is incremented by one with each successful CAS() on S in line 15 or 28, and that no other line modifies S. By Observation 2.1, $S.\ell$ is incremented exactly when a successful choose&lock() or unlock() operation responds, and it is initialized to zero. Therefore, at the s-th successful CAS() on S, $S.\ell = s$. Moreover, from Lemma 2.5, every other update of S beginning with the first is the response of a choose&lock() operation. Together, this proves (1) other than the claim that $P_{t_s^{CL}}.\ell = 2s-2$. From Lemma 2.5

and that S and P increment by one with each successful CAS(), it follows that property (6) for $s \geq 1$ implies property (1) for $s+1$.

We now prove $P_{t_s^{CL}}.\ell = 2s-2$ and (2) through (6) using induction on s.

Consider the point t_s^{CL} for the base case of $s = 1$, i.e., the response of the first successful choose&lock() operation. We want to show that $P_{t_0^{CL}}.\ell = 0$. Observe that only an unlock() operation can affect P, in line 19 or 27. Also, every unlock() operation returns in line 17 while $S.\ell = 0$, which is its initial value. Therefore, P does not change before t_1^{CL}, so $P.\ell$ remains at its initial value of zero, which proves that $P_{t_1^{CL}}.\ell = 0$.

Now we prove the inductive case. Given a positive integer s, suppose that (1) holds. We want to show that (2) through (6) hold for the given value of s.

To prove (5) and (6) other than the claim that $P_{t_{s+1}^{CL}}.\ell = 2s$, we observe that after a successful CAS() on S, the next process to read the updated value of S and then perform a CAS() on S using this updated value succeeds. From Lemma 2.5, point t_s^{UL} is the response of the s-th successful unlock() operation.

Observe that it follows immediately from Definition 2.8 that $t_s^{CL} \leq t_s^{PL} \leq t_s^{PU} \leq t_s^{UL} \leq t_{s+1}^{CL}$ and $t_s^{PL} \leq t_s^{PC}$. To show that $t_s^{PC} \leq t_s^{PU}$, first suppose that $t_s^{PU} < \infty$ and let op_{up} be the first unlock() operation in $OP_{16>t_s^{CL}}$ to execute line 27. Since $S.\ell = 2s-1$ throughout the interval $\left[t_s^{CL}, t_s^{UL}\right)$ and $S.\ell$ only increments with each CAS(), it follows that S does not change in the interval and so op_{up} reads $S_{t_s^{CL}}.i$ from $S.i$ in line 16. Because $op_{up} \in OP_{16>t_s^{CL}}$ it executes line 20 after t_s^{PL}, which makes $op_{up} \in OP_{20>t_s^{PL}}$. Therefore, op_{up}'s execution of line 21 is no earlier than t_s^{PC}, so $t_s^{PC} \leq t_s^{PU}$.

Next we show (2). Consider the first unlock() operation in $OP_{16>t_s^{CL}}$ to execute line 19, which we denote by op_{lp}, if it exists. The point when op_{lp} executes line 19 is $t_s^{PL} < \infty$ by definition. Since $S.\ell$ cannot decrease, and since $t_s^{PL} < t_s^{UL}$ by Definition 2.8, by (5) $S.\ell = 2s-1$ in the interval $\left[t_s^{CL}, t_s^{PL}\right]$. Therefore, op_{lp} is the first operation to execute line 19 with a local value of $\ell = 2s-1$. Recall that unlock() operations read an odd value of $S.\ell$ in line 16 if they progress past line 17. Thus, the only line that can change P when $P.\ell$ is even is line 19, which changes $P.\ell$ from a local value of $\ell - 1$ to ℓ. From (1), $P_{t_s^{CL}}.\ell = 2s-2$, which is even, so op_{lp} is the first process that attempts a CAS() on P with an old value for $P.\ell$ of $2s-2$. Therefore, $P.\ell = 2s-2$ throughout the interval $\left[t_s^{CL}, t_s^{PL}\right)$. It follows that op_{lp}'s CAS() succeeds and increments $P.\ell$ from $2s-2$ to $2s-1$, proving (2).

Next we show (4). Recall that we denote the first unlock() operation in $OP_{16>t_s^{CL}}$ to execute line 27 as op_{up}. The point when op_{up} executes line 27 is t_s^{PU}. Only line 27 can update P when $P.\ell$ is odd, by the same argument that only line 19 can update P when $P.\ell$ is even. Since op_{up} is invoked after t_s^{CL}, and since $t_s^{PU} < t_s^{UL}$ by Definition 2.8 when $t_s^{PU} < \infty$, op_{up} is the first operation to attempt a CAS() on P with the expected value of $P.\ell$ equal to $2s-1$. This CAS() is after t_s^{PL}, which updates $P.\ell$ to $2s-1$ as proven above, and op_{up} also executes line 20 after t_s^{PL} by definition of op_{up} and of line 20. Therefore, op_{up}'s local value of $pval$ matches $P.pval$, so its CAS() succeeds and increments the value of $P.\ell$ from $2s-1$ to $2s$, proving (4).

We now show (3). Consider the first unlock() operation in $OP_{20>t_s^{PL}}$ with a local value of i matching $S_{t_s^{CL}}.i$ to execute line 21; we denote this operation by op_{cp}. The point when op_{cp} executes line 21 is t_s^{PC}. Since $t_s^{PC} \leq t_s^{PU}$, P does not change since in the interval $\left[t_s^{PL}, t_s^{PC}\right]$. In particular, op_{cp} reads the same value from $P.pval$ in line 20 that op_{lp} reads from $C[S_{t_s^{CL}}.i][\phi]$ in line 18. If $C[S_{t_s^{CL}}.i][\phi]$ does not change in between op_{lp}'s read of it in line 18 and op_{cp}'s attempted CAS() on $C[S_{t_s^{CL}}.i][\phi]$ in line 21, then op_{cp}'s CAS() in $C[S_{t_s^{CL}}.i][\phi]$ to $\perp$ succeeds; otherwise, $C[S_{t_s^{CL}}.i][\phi]$ changes to some other value than $P_{t_s^{PL}}.pval$. Because only propose() can write non-$\perp$ values to C, and no two proposals use the same value, it follows that if $P_{t_s^{PL}}.pval \neq \perp$ then C does not contain $P_{t_s^{PL}}.pval$ at or after t_s^{PC}, which proves (3).

To complete the proof of (6), it remains to show that $P_{t_{s+1}^{CL}}.\ell = 2s$: since $P.\ell$ increments by one with each update, it suffices to show that P does not change in the interval $\left[t_s^{PU}, t_{s+1}^{CL}\right]$. Recall that only unlock() can change P, that

every `unlock()` operation that progresses past line 17 has an odd local value of ℓ, and that the `CAS()` attempts on P in lines 19 and 27 use an old value of $P.\ell$ that is at most ℓ. Thus, in order for a `CAS()` on P to successfully increment $P.\ell$ from $2s$ to $2s+1$ in $\left[t_s^{PU}, t_{s+1}^{CL}\right]$, the `unlock()` operation has a local value of ℓ of at least $2s+1$, since ℓ is even. This is not possible until after t_{s+1}^{CL}, which proves that $P_{t_{s+1}^{CL}}.\ell = 2s$.

Property (6) for s implies (1) for $s+1$, so the induction follows. □

Corollary 2.10. *If $P_{t_1} = (pval_1, \ell_1)$ and $P_{t_2} = (pval_2, \ell_2)$ such that $\ell_1 \neq \ell_2$ and neither $pval_1$ nor $pval_2$ are $\bot$, then $pval_1 \neq pval_2$.*

Proof. Assume without loss of generality that $t_1 < t_2$. Since $P.\ell$ can only increase, $\ell_1 < \ell_2$. Using Observation 2.7 and Lemma 2.9, $S_{t_1}.\ell = \ell_1$ and $S_{t_2}.\ell = \ell_2$, and they are both odd. From (3) of Lemma 2.9, C does not contain $P_{t_1}.pval$ after $t^{PC}_{(\ell_1+1)/2}$. Consider the operation op that first sets P to $(pval_2, \ell_2)$. From Lemma 2.9 and that $P.\ell$ increments with each successful `CAS()` on it, op makes this change at point $t^{PL}_{(\ell_2+1)/2}$. By definition of $t^{PL}_{(\ell_2+1)/2}$, op reads S in line 16 after $t^{CL}_{(\ell_2+1)/2}$, and from Lemma 2.9 $t^{CL}_{(\ell_2+1)/2} > t^{PC}_{(\ell_1+1)/2}$. Therefore, op reads some value different from $pval_1$ from $C[i][\phi]$ in line 18, and sets $P.pval$ to this different value $pval_2 \neq pval_1$ at $t^{PL}_{(\ell_2+1)/2}$. □

Claim 2.11. *Let $i^* \in \{0, 1\}$. If a `CAS()` on $C[i^*][\phi]$ from a non-$\bot$ value v to $\bot$ in $\left[t_s^{CL}, t_{s+2}^{CL}\right)$ is successful for some positive integer s such that $S_{t_s^{CL}}.i = i^*$, then it occurs at point t_s^{PC}, and $v = P_{t_s^{PL}}.pval$.*

Proof. From Lemma 2.9 and that $P.\ell$ increments by 1 with each successful `CAS()` on P, if $t^{CL}_{s'+1} < \infty$ then P changes exactly twice in the interval $\left[t^{CL}_{s'}, t^{CL}_{s'+1}\right)$ for any positive integer s', at points $t^{PL}_{s'}$ and $t^{PU}_{s'}$. (If $t^{CL}_{s'+1} = \infty$, then P changes at most twice in the interval.) From Observation 2.7, $P.pval$ can only be different from $\bot$ in the sub-interval $\left[t^{PL}_{s'}, t^{PU}_{s'}\right)$. Therefore, $P.pval$ takes on at most one non-$\bot$ value in the interval $\left[t^{CL}_{s'}, t^{CL}_{s'+1}\right)$, namely $P_{t^{PL}_{s'}}.pval$. From Lemma 2.9, $C[i^*][\phi]$ does not contain $P_{t^{PL}_{s'}}.pval$ at or after point $t^{PC}_{s'} \in \left[t^{PL}_{s'}, t^{PU}_{s'}\right]$.

Consider a successful `CAS()` on $C[i^*][\phi]$ from some value $v \neq \bot$ to $\bot$ at point $t \in \left[t_s^{CL}, t_{s+2}^{CL}\right)$. Because only line 21 can replace $C[i^*][\phi]$ with $\bot$, it follows that some `unlock()` operation op executes a successful `CAS()` in line 21 at point t. The first argument of the `CAS()` is op's local value of $pval$, which by the above paragraph is $v = P_{t^{PL}_{s'}}.pval$ for some s'. Therefore, op reads $P_{t^{PL}_{s'}}.pval$ from $P.pval$ in line 20 at point t' such that $t^{PL}_{s'} < t' < t$. From (3) of Lemma 2.9, C does not contain $P_{t^{PL}_{s'}}.pval$ at or after $t^{PC}_{s'}$, and so $t \leq t^{PC}_{s'}$. If $s' < s$, then $t^{PC}_{s'} < t^{CL}_s$, which contradicts that $t \geq t^{CL}_s$; therefore, $s' \geq s$.

Now suppose, for the purpose of proving a contradiction, that $t \neq t_s^{PC}$. Consider first the case where $t < t_s^{PC}$. Since $s' \geq s$ and $t^{PL}_{s'} < t < t^{PC}_s < t^{PL}_{s+1}$, it follows that $s' < s+1$ and so $s' = s$. Using Corollary 2.10, t^{PL}_s is the first point when $P.pval = P_{t^{PL}_s}.pval$, and it is the point of a successful `CAS()`, which sets $P.pval$ to $P_{t^{PL}_s}.pval$. We thus have $t^{PL}_s < t' < t < t^{PC}_s$, and since op's operations are on side i^*, it reads $i^* = S_{t^{CL}_s}$ from S in line 16. By definition of t^{PC}_s in Definition 2.8, op's execution of line 21 *is* t^{PC}_s, (the point when the first `unlock()` operation in $OP_{20>t^{PL}_s}$ whose local value of i matches $S_{t^{CL}_s}.i$ executes line 21). This contradicts that $t < t^{PC}_s$.

The remaining case is if $t > t^{PC}_s$. Since C does not contain $P_{t^{PL}_s}.pval$ after t^{PC}_s, it follows that $s' > s$. Also, since $t < t^{CL}_{s+2} < t^{PL}_{s+2}$, $s' < s+2$, so $s' = s+1$. Since $i^* = S_{t^{CL}_s}.i$ and $S.i$ alternates between zero and one at each successful `unlock()` operation's response, and using Lemma 2.5, $S_{t^{CL}_{s+1}}.i = 1 - i^*$. From Lemma 2.9, if $P_{t^{PL}_{s+1}}.pval \neq \bot$, then it is the value of some proposal written to $C[1-i^*][\phi]$. Since proposals are unique, $C[i^*][\phi]$ never contains $P_{t^{PL}_{s+1}}.pval$, so op's `CAS()` from $P_{t^{PL}_{s'}}.pval = P_{t^{PL}_{s+1}}.pval$ to $\bot$ is not successful, which contradicts our assumptions.

We have now proven that $t = t^{PC}_s$. From Definition 2.8, op reads value v from $P.pval$ in line 20 in the interval $\left(t^{PL}_s, t^{PC}_s\right)$, and therefore $v = P_{t^{PL}_s}.pval$. □

Lemma 2.12. *Let s be a positive integer such that $t_s^{PL} < \infty$. If $P_{t_s^{PL}}.pval = \bot$ and $2s - 1 \bmod \lambda < \lambda - 4$, then no process invokes* `clear(2s − 1)`. *If $P_{t_s^{PL}}.pval \neq \bot$ or $2s - 1 \bmod \lambda \geq \lambda - 4$, then at least one process responds from a* `clear(2s − 1)` *operation before $t_s^{PU} < t_s^{UL}$ if $t_s^{PU} < \infty$.*

Proof. A process can run `clear(2s − 1)` only if it is executing an `unlock()` operation that reads $2s - 1$ from $S.\ell$ in line 16 (the first line of `unlock()`). From Lemma 2.9, we only need to consider `unlock()` operations that are invoked in the interval $\left(t_s^{CL}, t_s^{UL}\right)$, since $S.\ell = 2s - 1$ exactly in the interval $\left[CLs, t_s^{UL}\right)$.

From Lemma 2.9, $P.\ell = 2s - 1$ in and only in the interval $\left[t_s^{PL}, t_s^{PU}\right)$.

Consider the case where $P_{t_s^{PL}}.pval \neq \bot$ or $2s - 1 \bmod \lambda \geq \lambda - 4$, and $t_s^{PU} < \infty$. From Definition 2.8, t_s^{PL} is the first point when an `unlock()` operation executes line 19 with a local value of ℓ matching $2s - 1$. Thus, the `unlock()` operation op_{up} that executes line 27 (incrementing $P.\ell$ to $2s$) at t_s^{PU} previously executes line 20 in $\left(t_s^{PL}, t_s^{PU}\right)$, so it reads a non-$\bot$ value from $P.pval$. Hence, the conditional in op_{up}'s execution of line 26 evaluates to true so op_{up} responds from a `clear(2s − 1)` operation in line 26 before t_s^{PU}, proving the last part of the lemma.

Now consider the case where $P_{t_s^{PL}}.pval = \bot$ and $2s - 1 \bmod \lambda < \lambda - 4$. For the purpose of showing a contradiction, suppose that an `unlock()` operation op' invokes `clear(2s − 1)` in line 26. Let t' be the point when op' reads P in line 20; by Definition 2.8 and because op' reads $S.\ell = 2s - 1$ after t_s^{CL}, $t' > t_s^{PL}$. Suppose first that $t' < t_s^{PU}$, so that $P.\ell = 2s - 1$ at point t'. Then op' reads $\bot$ from $P.pval$. It therefore does not run `clear()` in line 26. Suppose instead that $t' > t_s^{PU}$, so $P.\ell \geq 2s$ at the point of op''s read of P. If $P.\ell$ is even at t', by Observation 2.7 $P.pval = \bot$. In this case, op' sets its local $pval$ variable to $\bot$ and therefore does not execute `clear()` in line 26. Otherwise, if $P.\ell$ is odd at t', its last change is from a successful `CAS()` in line 19. By Lemma 2.9, this can only happen if $S.\ell$ also changed since t_s^{PU} and before t'. Therefore, op' returns in line 24, and does not reach line 26 to run the `clear()` method. Since both cases lead to a contradiction, the first part of the lemma follows. □

Corollary 2.13. *If some process invokes* `clear(2s − 1)` *and $t_s^{PU} < \infty$, then at least one process responds from a* `clear(2s − 1)` *operation before $t_s^{PU} < t_s^{UL}$.*

Proof. Suppose that some process invokes `clear(2s − 1)`. From Lemma 2.9, t_s^{PL} occurs before any `unlock()` operation that reads $2s-1$ from $S.\ell$ reaches line 26, so $t_s^{PL} < \infty$. By the first part of Lemma 2.12, either $P_{t_s^{PL}}.pval \neq \bot$ or $2s - 1 \bmod \lambda \geq \lambda - 4$. The corollary then follows using the second part of Lemma 2.12. □

Claim 2.14. *When used in the context of the RC object, invocations of the* `clear()` *method satisfy (1).*

Proof. Suppose that `clear(ℓ₁)` and `clear(ℓ₂)` are both invoked in an execution with $\ell_1 < \ell_2$. Since any `unlock()` operation that reads an even value of $S.\ell$ in line 16 returns in line 17 and processes can only invoke `clear()` from line 26 of the `unlock()` method, ℓ_1 and ℓ_2 are odd. Thus, $\ell_1 = 2s_1 - 1$ and $\ell_2 = 2s_2 - 1$ for integers $s_1 < s_2$. Consider the first invocation of `clear(ℓ₂)`, by some process p_2 at point t_2. As p_2 invokes `clear(ℓ₂)` from an `unlock()` operation, ℓ_2 equals the value of $S.\ell$ that p_2 reads in line 16 at point $t_1 < t_2$. By Lemma 2.9, $t_{s_1}^{PU} < t_{s_2}^{CL} < t_1$. By Corollary 2.13, some `clear(2s₁ − 1)` operation responds before $t_{s_1}^{PU}$, and hence before t_2. □

2.3.4 Clear Method Properties. In this section, we prove several statements about the `clear()` method. Recall from Figure 3 that $h = \lambda - 5\phi$, which is the size of sub-array $C[i][5\phi + 1, \ldots, \lambda]$.

Claim 2.15. *If $T[j] = \ell > 0$ at point t, where $1 \leq j \leq h$, then a* `clear(ℓ)` *operation is invoked before t.*

Proof. First, consider the case where $h/2 < j \leq h$. Both of $T[j]$'s children, defined as in lines 35 and 36 as $T[\min\{2j, h+1\}]$ and $T[\min\{2j+1, h+1\}]$, are $T[h+1]$ in this case. Since no line modifies $T[h+1]$, and it is initialized to ∞, $T[h+1] = \infty$ at all points of an execution. Since $T[j]$ is initialized to 0, it changes to ℓ in a successful `CAS()` operation in line 40 of some `clear(ℓ′)` operation. Since $left$ and $right$ both equal ∞ in this case, $\min\{\ell', left, right\} = \min\{\ell', \infty, \infty\} = \ell'$, which implies that $\ell' = \ell$ and proves the claim for this case.

Otherwise, suppose that $1 \leq j \leq h/2$, i.e., $T[j]$ has at least one non-leaf child. Again, $T[j]$ changes in line 40 of some `clear(ℓ′)` operation. This time, $\ell = \min\{\ell', \mathit{left}, \mathit{right}\}$, which is one of ℓ', left, or right. If $\ell = \ell'$, the claim is immediately satisfied. Otherwise, ℓ equals either left or right, so one of the non-leaf children of $T[j]$, $T[2j]$ or $T[2j+1]$, respectively, is set to ℓ at the time of the read in line 35 or 36, which occurs before point t. In this case, we repeat the argument for this larger value of j, which eventually leads to some $j > h/2$ and proves the claim. □

Lemma 2.16. *For each node $T[j]$ where $1 \leq j \leq h+1$, the value of $T[j]$ is non-decreasing over time. Also, the value in $T[j]$ is not larger than the value of its children $T[\min\{2j, h+1\}]$ and $T[\min\{2j+1, h+1\}]$ at any point. (In other words, T satisfies the min-heap property, although we are not using traditional heap operations.)*

Proof. We will prove the following two claims by induction over points in time t, using the fact that only line 40 can change $T[j]$ for $1 \leq j \leq h$ and that $T[h+1]$ is always set to its initial value of ∞. (Recall that we define points so that at most one shared memory operation takes place at each point, and that each shared memory operation takes place at a positive integer point in time.)

For all $1 \leq j \leq h+1$,

$$T[j] \text{ never decreases at any point in } [0, \dots, t], \text{ and} \tag{2}$$

$$T[j] \leq \min\{T[\min\{2j, h+1\}], T[\min\{2j+1, h+1\}]\} \text{ at all points in } [0, \dots, t]. \tag{3}$$

At point $t = 0$, (2) holds trivially and (3) follows from the initialization of T. Also, both claims hold for all t for $j = h+1$, since $T[h+1]$ always equals ∞ and using the convention that $\infty \leq \infty$.

Now, suppose that (2) and (3) hold for point $t = s-1$. If point s is not the point of a successful `CAS()` operation in line 40, then (2) and (3) also hold for point $t = s$ because only a successful `CAS()` in line 40 can modify T.

Suppose then that some process p executes a successful `CAS()` in line 40 at point s as part of a `clear(ℓ)` operation for some $\ell > 0$ and some j in $\{1, \dots, h\}$. This `CAS()` successfully changes $T[j]$ from node to $\min\{\ell, \mathit{left}, \mathit{right}\}$, where node, ℓ, left, and right refer to p's local variables. Process p executes lines 34–36 at points $t_1 < t_2 < t_3 < s$, respectively, i.e., $(\mathit{node}, \mathit{left}, \mathit{right}) = \big(T_{t_1}[j], T_{t_2}[\min\{2j, h+1\}], T_{t_3}[\min\{2j+1, h+1\}]\big)$. Using the inductive hypothesis for $t = s-1$,

$$\begin{aligned} \mathit{node} &= T_{t_1}[j] \\ &\leq \min\left\{T_{t_1}[\min\{2j, h+1\}], T_{t_1}[\min\{2j+1, h+1\}]\right\} \\ &\leq \min\left\{T_{t_2}[\min\{2j, h+1\}], T_{t_3}[\min\{2j+1, h+1\}]\right\} \\ &= \min\{\mathit{left}, \mathit{right}\}, \end{aligned}$$

where the first inequality follows from (3) and the second inequality follows from (2). Therefore, p can only decrease $T[j]$ at point s in line 40 if $\ell < \mathit{node}$.

Suppose for the purpose of proving a contradiction that $\ell < \mathit{node}$: using Claim 2.15, a `clear(node)` operation is invoked before t_1. From (1), some `clear(ℓ)` operation responds before the first invocation of `clear(node)`, which in turn is before t_1. From the conditionals in lines 32 and 38, we have that $T[1] \geq \ell$ before the first response of `clear(ℓ)`, which again is before t_1. Finally, applying (2) for the point before s when p executes line 38 implies that $T[1]$ is still at least ℓ at that point, so p returns and does not execute line 40 at point s. This contradicts that $\ell < \mathit{node}$ and therefore proves (2) for point $t = s$.

To show that (3) also holds for point $t = s$, observe that

$$
\begin{aligned}
T_s[j] &\leq \min\{left, right\} \\
&= \min\left\{T_{t_2}[\min\{2j, h+1\}], T_{t_3}[\min\{2j+1, h+1\}]\right\} \\
&\leq \min\{T_s[\min\{2j, h+1\}], T_s[\min\{2j+1, h+1\}]\},
\end{aligned}
$$

where the last inequality follows from applying (2) for $t = s$ to $T[\min\{2j, h+1\}]$ and $T[\min\{2j+1, h+1\}]$. □

Corollary 2.17. *At the time t of the first response of any* `clear(ℓ)` *operation, for a given* ℓ*, all* $T[j]$ *for* $1 \leq j \leq h$ *are set to exactly* ℓ*.*

Proof. First, using the min-heap property of T as proven in Lemma 2.16 and that `clear(ℓ)` can only respond when $T[1] \geq \ell$ (in line 32 or 38), it immediately follows that $T[j] \geq \ell$ for all $1 \leq j \leq h$ at point t.

Next, suppose for the purpose of proving a contradiction that, for some $T[j]$ with $1 \leq j \leq h$, $T[j] = \ell_2 > \ell$ at point t. Using Claim 2.15, some `clear(ℓ₂)` operation is invoked before t. However, this contradicts (1), since $\ell_2 > \ell$ and `clear(ℓ₂)` is invoked before the first response of `clear(ℓ)`. □

Lemma 2.18. *Suppose that* $C[i][j + 5\phi] = v \neq \bot$ *at the time t of the first invocation of* `clear(ℓ)` *for a given integer* ℓ *such that* $i = \lfloor \ell/2 \rfloor \bmod 2$ *and* $j \in \{1, \ldots, h\}$*. If a* `clear(ℓ)` *operation responds at point* $t_2 > t$*, then* $C[i][j + 5\phi] \neq v$ *at all points at and after* t_2*.*

Proof. Let $t_1 \in (t, t_2]$ be the point of the first response of any `clear(ℓ)` operation. From Corollary 2.17, at point t_1, $T[j] = \ell$. From Claim 2.15, $T[j]$ may only be set to ℓ after point t. Therefore, $T[j]$ changes its value to ℓ at some point in the interval (t, t_1).

Consider the process p that first changes $T[j]$ to ℓ in the interval (t, t_1) and call this point t_j. Only line 40 can change $T[j]$, so p executes that line as part of a `clear(ℓ_p)` operation. In line 40, p performs a successful `CAS()` on $T[j]$ to $\min\left\{\ell_p, left, right\right\} = \ell \leq \ell_p$. From (1) and that $t_j < t_1$ (i.e., $T[j]$ changes to ℓ before the first response of any `clear(ℓ)` operation), $\ell_p \leq \ell$; therefore, $\ell_p = \ell$. By definition of t, p invokes `clear(ℓ)` at or after point t.

It thus follows that at some point in $\left(t, t_j\right)$, p executes line 39 of its `clear(ℓ)` operation, i.e., it attempts a `CAS()` operation to $\bot$ on the corresponding array element $C[i][j + 5\phi]$ where $i = \lfloor \ell/2 \rfloor \bmod 2$. The `CAS()` on $C[i][j + 5\phi]$ either succeeds, in which case $C[i][j + 5\phi]$'s value changes to $\bot$, or fails, in which case $C[i][j + 5\phi]$ changes in between p's execution of lines 37 and 39, both of which occur after point t. Therefore, $C[i][j + 5\phi]$ changes at some point in $\left(t, t_j\right) \subset (t, t_2)$. Because only `propose()` operations can write non-$\bot$ values to C, and because no two `propose()` operations can propose the same value, the lemma follows. □

Claim 2.19. *If an* `unlock(v)` *operation modifies* S*,* P*, or* T *at point t, then* $S_t.val = v$*.*

Proof. Suppose that an `unlock(v)` operation op modifies S, P, or T at point t. Then, op executes either 19, 27, 28, or 40 at point t. Each of these lines consists of a `CAS()` operation for which the new value includes a sequence number no larger than op's local value of $\ell + 1$, which it reads in line 16 at point $t' < t$.

Now suppose for the purpose of proving a contradiction that $S_t.val \neq v$. Since $S.val = v$ and $S.\ell$ is odd at point t', and since only a successful `CAS()` on S in line 15 can change $S.val$, which also increments $S.\ell$ from an even to an odd number, it follows that $S_t.\ell \geq S_{t'}.\ell + 2$. This immediately contradicts the case in which op executes a successful `CAS()` in line 28, since this `CAS()` would change $S.\ell$ to $\ell + 1 = S_{t'}.\ell + 1$.

From Lemma 2.9, $P.\ell$ is incremented to $S_{t'}.\ell + 1$ before the point when $S.\ell$ is incremented to $S_{t'}.\ell + 2$, which in turn is no later than point t. This contradicts the case in which op executes a successful `CAS()` in line 19 or 27, since such a `CAS()` increments $P.\ell$ to a value at most $S_{t'}.\ell + 1$.

Finally, from Corollary 2.13 and Lemma 2.9, some `clear(ℓ)` operation responds before t. Using Lemma 2.16 and Corollary 2.17, all values in T are at least ℓ at and after point t, and which contradicts the case where op changes T to a value at most ℓ at point t.

Since all cases lead to a contradiction, the claim follows. ◻

2.3.5 ABA-Freedom. In this section we will prove that the sequence of non-$\bot$ values stored in $S.val$ exhibits no ABAs, i.e., each successful `choose&lock()` either changes the interpreted agreement-value to a new non-$\bot$ value in V or sets or leaves it at $\bot$.

Claim 2.20. *At all points in an execution,* $S.i = \lfloor S.\ell/2 \rfloor \bmod 2$.

Proof. First, recall that $S.i$ and $S.\ell$ are both initialized to zero. Second, note that $\lfloor S.\ell/2 \rfloor \bmod 2$ is the second-least significant bit of $S.\ell$. Third, note that each update of S, in line 15 or 28, increments $S.\ell$ by one. Fourth, using Observation 2.1 and Lemma 2.5, every second update of S occurs at the response of a successful `unlock()` operation, which also flips the value of $S.i$ to $1-S.i$, and no other operation can change $S.i$. The claim follows from the fact that the second-least significant bit of an integer flips with every second increment of that integer. ◻

Claim 2.21. *Let s be a positive integer such that $t_s^{UL} < \infty$. If $P_{t_s^{PL}}.pval \neq \bot$ or $2s-1 \bmod \lambda \geq \lambda - 4$, then let $\lambda_s = \lambda$; otherwise, let $\lambda_s = 5\phi$.*

If $v \neq \bot$ is written to entry $C[S_{t_s^{CL}}.i][j]$ before t_s^{PC}, where $j \in \{1, \ldots, \lambda_s\} \setminus \{\phi\}$, then v is not stored in array C at any point after t_s^{UL}. If $v \neq \bot$ is written to entry $C[S_{t_s^{CL}}.i][\phi]$ before t_s^{CL}, then v is not stored in array C at any point after t_s^{UL}.

Proof. Recall that v is only written to C when it gets proposed in line 2, and only to one entry in C, and no value can be proposed twice. It thus suffices to show that either

- $C[S_{t_s^{CL}}.i][j]$ changes at some point in $\left(t_s^{PC}, t_s^{UL}\right)$, or $\left(t_s^{CL}, t_s^{UL}\right)$ in the case of $j = \phi$, or
- $C[S_{t_s^{CL}}.i][j] = \bot$ throughout that same interval.

Only `unlock()`, `clear()` and `propose()` can modify C. Line 2 of a `propose()` operation can change $C[S_{t_s^{CL}}.i][j]$ if $(\alpha, \beta) = (S_{t_s^{CL}}.i, j)$. Lines 21, 25 and 39 can also change $C[S_{t_s^{CL}}.i][j]$, by executing a successful $C[S_{t_s^{CL}}.i][j]$.`CAS`$(v, \bot)$.

First, consider the case where $j \in \{1, \ldots, 5\phi\} \setminus \{\phi\}$, and let op be the successful `unlock()` operation that responds at t_s^{UL}. Since the `unlock()` operation is successful, op completes the entire for-loop in lines 22–25. If $C[S_{t_s^{CL}}.i][j] = v$ when op reads $C[S_{t_s^{CL}}.i][j]$ in line 23 and $C[S_{t_s^{CL}}.i][j] = v$ when op executes its `CAS()` in line 25, then $C[S_{t_s^{CL}}.i][j]$ changes to $\bot$. By Lemma 2.9, op starts the for-loop after t_s^{PC} and (since op's response is t_s^{UL}) finishes the loop before t_s^{UL}. Hence, if $C_{t_s^{PC}}[S_{t_s^{CL}}.i][j] = v \neq \bot$, then at some point in $\left(t_s^{PC}, t_s^{UL}\right)$ the value v stored in C gets erased or overwritten.

Next, consider the case where $\lambda_s = \lambda$ and $j > 5\phi$. By definition of λ_s, either $P_{t_s^{PL}}.pval \neq \bot$ or $2s-1 \bmod \lambda \geq \lambda - 4$. From Lemma 2.12, a `clear`$(2s-1)$ operation responds before t_s^{UL}. From Definition 2.8 and that `clear`$(2s-1)$ can only be invoked in line 26 from an `unlock()` operation that reads $2s-1$ from S in line 16, the first `clear`$(2s-1)$ operation is invoked after t_s^{PC}. Therefore, using Claim 2.20 and Lemma 2.18, either $C[S_{t_s^{CL}}.i][j]$ changes in $\left(t_s^{PC}, t_s^{UL}\right)$ or $C_{t_s^{CL}}[S_{t_s^{CL}}.i][j] = \bot$.

Finally, consider the case where $j = \phi$. From Lemma 2.9, the first `unlock()` operation invoked after t_s^{CL} to execute line 19 has a successful `CAS()` on P at point t_s^{PL} (operation op_{lp} in that lemma's proof), and so $P_{t_s^{PL}}.pval$ is the value that op_{lp} reads from $C[S_{t_s^{CL}}.i][\phi]$ in line 18. From property (3) of Lemma 2.9, if $P_{t_s^{PL}}.pval \neq \bot$ then C does not contain $P_{t_s^{PL}}.pval$ after t_s^{PC}. Since the value of $C[S_{t_s^{CL}}.i][\phi]$ was $P_{t_s^{PL}}.pval$ at some point after t_s^{CL}, it follows that either $C[S_{t_s^{CL}}.i][j]$ changes in $\left(t_s^{CL}, t_s^{UL}\right)$ or $C_{t_s^{CL}}[S_{t_s^{CL}}.i][j] = \bot$. ◻

Lemma 2.22. *Let $t_1 < t_2$ be two points in time such that $S_{t_1}.val = S_{t_2}.val = val \neq \bot$. Then*

(a) no successful `choose&lock()` *operation responds in $(t_1, t_2]$, and*
(b) $S_t.val = val$ for all $t \in (t_1, t_2]$.

Proof. Part (b) follows from (a), because according to Observation 2.3 $S_t.val$ may only change at the response of a successful choose&lock() operation. Hence, it suffices to prove (a).

For the purpose of proving a contradiction, assume that a successful choose&lock() operation responds in $(t_1, t_2]$, and let $t' \in (t_1, t_2]$ be the response of the last such choose&lock() operation d. The process executing d reads val from $C[i][j]$ for some $i \in \{0, 1\}$ and $j \in \{1, \ldots, \lambda\}$ and at point t' it writes it into $S.val$ (in line 15). Hence, C contains val at a point after the first shared memory operation of d executes.

Since $S.val$ begins as $\perp$, a successful choose&lock() operation d' prior to d sets the interpreted agreement-value to val, at point $t^{CL}_{s'}$ for some s'. Since no value is proposed twice, d' also reads val from $C[i][j]$. From Lemma 2.5, a successful unlock() operation responds in between the responses of d and d'. Let u denote the first such successful unlock operation, which responds at point $t^{UL}_{s'}$ from Lemma 2.9. From Observation 2.6, u is invoked after d' responds and responds before d is invoked, so the interpreted agreement-value at the invocation of u is val.

If $j \leq 5\phi$, then from Claim 2.21, val is not stored in C by the response of u, which contradicts that C contains val at the point of d's invocation. If $j > 5\phi$, then d''s local value of low is higher than 1 after line 11, so d' reads $C[i][\phi] \neq \perp$ in that same line. From Claim 2.11, $C[i][\phi] \neq \perp$ until at least $t^{PC}_{s'} > t^{CL}_{s'}$. Thus, by Definition 2.8 and Lemma 2.9, $P_{t^{PL}_{s'}}.pval \neq \perp$, and so by Claim 2.21 C does not contain val after $t^{UL}_{s'}$, again contradicting that C contains val at the point of d's invocation. ◻

2.3.6 Linearizability.

Theorem 2.23. *The algorithm in Figures 1–3 is a strongly linearizable implementation of an RC object.*

Proof. Recall that the sequential specification of RC is specified in Section 2.

A read() operation op_R returns $S_t.val$ at $lin(op_R)$, and does not change the status or interpreted agreement-value of R. Similarly, a propose() op_P does not change the status or interpreted agreement-value at $lin(op_P)$, and it also does not return anything.

Now consider a choose&lock() operation op_C. It follows immediately from Section 2.3.1, the CAS() in line 15, and if-condition in line 4 that a choose&lock() is successful if and only if the interpreted locking state of R is unlocked at point $lin(op_C)$. (Recall that, for unsuccessful choose&lock() and unlock() operations op_T, $lin(op_T)$ is immediately after $lin(op_S)$ when op_S is the operation which first changes S after op_U's invocation.) Since by Lemma 2.5 a successful choose&lock() operation changes $S.\ell$ from an even to an odd value, and by Section 2.1 R is locked exactly when $S.\ell$ is odd, a successful choose&lock() operation changes R's status to locked. If successful, then by Lemma 2.4 the new interpreted agreement-value was previously proposed, and by Lemma 2.22 it is either $\perp$ or different from all previous interpreted agreement-values. If unsuccessful, then it does not change the status of R, and by Observation 2.3 also not the interpreted agreement-value.

Next consider an unlock() operation. If an unlock(val) operation op_U is successful, then from line 17 the status of R is locked and the interpreted agreement-value is val at op_u (in line 28 by Observation 2.1) and the operation unlocks R at op_U.

If an unlock(val) operation op'_U is unsuccessful, $lin(op'_U)$ could be in op'_U's execution of line 16 when R is unlocked or has an interpreted agreement-value different from val. Otherwise, $lin(op'_U)$ is immediately after the first shared memory operation which changes S after line 16, at which point R becomes unlocked. In either case, R is unlocked or has an interpreted agreement-value different from val at point $lin(op'_U)$. Thus, an unlock(val) operation is successful and unlocks R if and only if the status of R is locked and with interpreted agreement-value equal to val at point $lin(op'_U)$.

From all the above we conclude that if we order all operations op with $lin(op) < \infty$ by $lin(op)$, then we obtain a valid sequential history. By the definition of lin and by Observation 2.2, $lin(op)$ is between the invocation and response of op. This proves linearizability.

lin also defines strong linearization points: consider a prefix E' of an execution E that ends at point $t = lin(op)$. From the definition of lin, op also linearizes at point t in E'. Hence, each operation in E' linearizes at the same point in E' as in E. Since this is true for all prefixes E' of E, strong linearizability follows. ◻

2.4 Distribution of Agreement-Value

In this section, we analyze the distribution of the interpreted agreement-value of an RC object after a successful `choose&lock()` call. Since Theorem 2.23 shows that lin defines strong linearization points, we can refer to $lin(op)$ as the linearization point of op.

We consider an execution E on an RC object R scheduled by the *instruction pointer-aware*, or *ip-aware*, adversary as follows.

Definition 2.24 (ip-aware adversary)**.** We define a *step* of an execution as a single shared memory operation (i.e., a single read, write, or `CAS()`) and all preceding local operations by the same process. At each step in the execution, the adversary gets to choose the process p to take the next step based on the adversary's available information. (The adversary's choices can be probabilistic.) If p has no pending next line, then the adversary chooses also a method for p to invoke, including the arguments of that operation. For each process that has taken at least one step, the adversary can see the process's most recently executed line number in the program code, and whether it is the response point of an implemented method. If a process responds from a method call invoked by the adversary, then the adversary can also see the return value. The adversary cannot directly see the value of shared memory variables, processes' local variables including the outcomes of their random coin flips, or the return value of helper methods. (We relax some of these restrictions in the analysis.)

Our definition helps overcome a shortcoming of other adversary definitions: Usually such definitions limit what information about the past execution an adversary is allowed to take into account when deciding which process gets scheduled to take the next step. But if a randomized algorithm uses atomic objects, then the adversary has less information available than if the algorithm uses (e.g., linearizable) implemented objects. This means the analysis of a randomized algorithm must, in general, take into account the exact implementation of any object that is used by the algorithm. It is very difficult to specify types in such a way that any object satisfying the specification behaves the same way under a given adversary model. Strong linearizability fixes this problem for the strong adaptive adversary model, in the sense that all strongly linearizable objects can be assumed to be atomic for the purpose of analyzing a randomized algorithm using them. But no similar correctness condition is known for any other adversary model. Even worse, e.g., for the oblivious adversary, any such correctness condition would be infeasible to obtain [7]. To overcome this impasse, our ip-aware adversary definition explicitly states what information the adversary can learn from implemented method calls. This carefully chosen definition allows us to analyze the BDCAS algorithm, even though it uses an implemented RC object.

For RC, the `propose()`, `choose&lock()`, `unlock()`, and `read()` methods are directly invocable by the RC adversary, whereas the `clear()` method is a helper method, and can only be invoked by an `unlock()` operation in line 26. For the BDCAS adversary we use in Section 3.5, the `BDCAS()` and `read()` methods of BDCAS are directly invokable by the BDCAS adversary, and the `finish()` method and all the RC methods are helper methods that can only be invoked within other methods.

In the analysis, we assume that the adversary has some extra information on the past execution. We define the *trace* for an execution with respect to an RC object, and in Definition 3.28 with respect to a BDCAS object.

Definition 2.25 (RC Trace)**.** For an execution E on an RC object R, we define the *trace* $\mathcal{T}_R(E)$ to be a sequence that indicates for each step $\sigma \geq 1$ of E:

(i) the process p that executes step σ;

(ii) if step σ is the first step (i.e., the invocation) of an implemented method call of *RC*, the name and arguments (if any) of that call;
(iii) the line number corresponding to step σ, including information on whether σ is the final step (i.e., the response) of its method call;
(iv) the value of $R.S.val$ at step $\sigma - 1$;
(v) the values of $R.S.\ell$, $R.S.i$, $R.P$, and $R.T[1]$ at σ; and
(vi) if step σ is the response of a `propose(`v`)` operation, the value chosen for α.

Because the analysis for RC always involves a single execution E on a single RC object R at a time, we use the notation $\mathcal{T}(t)$ as shorthand for $\mathcal{T}_R(E_t)$, where E_t is the t-step prefix of E. We also define $\mathcal{T}()$ without arguments as shorthand for $\mathcal{T}_R(E)$.

Definition 2.25 does not explicitly specify the return values of methods invoked by the adversary. However, because only the `read()` method returns a value, which is the value of $S.val$, and at most one shared memory operation takes place per point in time, the adversary can deduce the return value of `read()` from (iv).

Recall from Definition 2.8 and Lemma 2.9 that, for each $s \geq 1$, t_s^{CL} is the s-th linearization point of a successful `choose&lock()` operation and t_s^{UL} is the s-th linearization point of a successful `unlock()` operation. For both points, they are ∞ if they do not exist, and we additionally defined that $t_0^{CL} = t_0^{UL} = 0$. If $v \in V$ is proposed in execution E, then we let r_v denote the point when v is proposed; otherwise, $r_v = \infty$. Finally, for $s \geq 1$, p_s denotes the number of proposals v such that $t_s^{UL} < r_v < t_{s+1}^{UL}$.

Definition 2.26 (Outdated/recent values)**.** Value $v \in V$ is *outdated* in execution E if $r_v < t_s^{CL}$ and $t_{s+1}^{UL} < \infty$ for some $s \geq 1$. If v is not outdated and $r_v < \infty$ then v is *recent.*

Main Theorem Statement. The main result we prove in this section is the following theorem, which gives an upper bound for the probability of selecting any given recent proposal as the interpreted agreement-value and an upper bound for selecting $\bot$. Recall that $S.val$ is the first component of $R.S$, and $S.val \in V \cup \{\bot\}$.

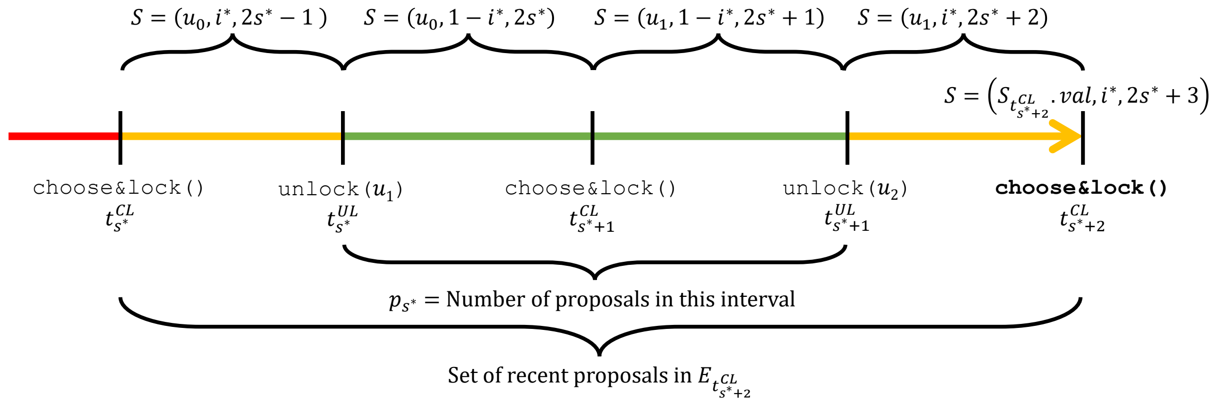

Fig. 4. Setup of Theorem 2.27

Theorem 2.27 (Selection of Recent Proposals). *Let E be an infinite random execution in which the ip-aware adversary as described above schedules calls to the methods of an RC object R, and let E_t be the prefix of the first t steps of E. We additionally impose the following restriction on the adversary:*

(R) For every $s \geq 0$, a bounded number of method calls of the RC object are invoked between points t_s^{UL} and t_{s+1}^{UL}.

Let $s^* \geq 0$. There exists an integer random variable $g_{s^*} = g_{s^*}\left(\mathcal{T}\left(t^{UL}_{s^*+1}\right)\right)$ satisfying

$$0 \leq g_{s^*} \leq p_{s^*} \text{ and } \mathbf{E}\left[g_{s^*} \,\middle|\, E_{t^{UL}_{s^*}}\right] = (1/2) \cdot \mathbf{E}\left[p_{s^*} \,\middle|\, E_{t^{UL}_{s^*}}\right], \tag{4}$$

such that the following hold. If v is a recent proposal of execution $E_{t^{CL}_{s^*+2}}$, then

$$\Pr\left[S_{t^{CL}_{s^*+2}}.val = v \,\middle|\, \mathcal{T}\left(t^{CL}_{s^*+2}\right)\right] < \frac{11}{\max\{g^*_s, 11\}} + \frac{3}{n} \tag{5}$$

and

$$\Pr\left[S_{t^{CL}_{s^*+2}}.val = \bot \,\middle|\, \mathcal{T}\left(t^{CL}_{s^*+2}\right)\right] \leq (1/\log n)^{g_{s^*}}. \tag{6}$$

The next result bounds the probability of selecting an outdated value as the interpreted agreement-value.

Theorem 2.28 (Selection of Outdated Proposals). *Let E be an execution under the same assumptions as Theorem 2.27. Let $c = \lfloor 2s/\lambda \rfloor$. There is a rare 'bad' event $\mathcal{B}_{s^*} = \mathcal{B}_{s^*}\left(E_{t^{UL}_{s^*}}\right)$ satisfying*

$$\Pr\left[\mathcal{B}_{s^*} \,\middle|\, E_{t^{CL}_{c\lambda/2+1}}\right] \leq 1/n^2,$$

such that if D is the set of outdated proposals in $E_{t^{CL}_{s^*+2}}$, then

$$\Pr\left[\left(S_{t^{CL}_{s^*+2}}.val \in D\right) \wedge \overline{\mathcal{B}_{s^*}} \,\middle|\, \mathcal{T}\left(t^{CL}_{s^*+2}\right)\right] \leq 6/\log^2 n.$$

2.4.1 Proof of Theorem 2.27. In the following we omit the prefix R when we refer to variables of R, e.g., we write S instead of $R.S$. From Lemma 2.9 and that $S.i$ alternates between 0 and 1 at each successful `unlock()` operation's linearization point, there exist values $u_0, u_1, u_2 \in V \cup \{\bot\}$ and an integer $i^* = (s^*+1) \bmod 2$, such that:

$$S_t = \begin{cases} (u_0, i^*, 2s^*-1), & \text{if } t \in \left[t^{CL}_{s^*}, t^{UL}_{s^*}\right) \\ (u_0, 1-i^*, 2s^*), & \text{if } t \in \left[t^{UL}_{s^*}, t^{CL}_{s^*+1}\right) \\ (u_1, 1-i^*, 2s^*+1), & \text{if } t \in \left[t^{CL}_{s^*+1}, t^{UL}_{s^*+1}\right) \\ (u_1, i^*, 2s^*+2), & \text{if } t \in \left[t^{UL}_{s^*+1}, t^{CL}_{s^*+2}\right) \\ (u_2, i^*, 2s^*+3), & \text{if } t = t^{CL}_{s^*+2}. \end{cases} \tag{7}$$

(7) follows from observing when each component of S can change. $S.val$ may change only at the linearization point of a successful `choose&lock()` call. $S.i$ changes precisely at the linearization point of a successful `unlock()` call. Lastly, $S.\ell$ increments by one at the linearization point of each successful `choose&lock()` and `unlock()` call.

Values u_0 and u_1 can be inferred from $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$, but, in general, this is not the case for u_2, as $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$ does not indicate u_2.

Recall that each value $u \in V$ is proposed at most once, and let $W \subseteq V$ be the set of all values proposed in $E_{t^{CL}_{s^*+2}}$. For each $u \in W$, let $C[\alpha_u][\beta_u]$ be the array entry that u in written to at point r_u, in line 2. Note that r_u and α_u can be inferred from $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$ for any $u \in W$, but, in general, β_u cannot. We define sets of proposals L and Q as

$$\begin{aligned} L &= \left\{u \in W : \alpha_u = i^*, r_u \in \left(t^{CL}_{s^*}, t^{CL}_{s^*+2}\right)\right\}, \\ Q &= \left\{u \in W : \alpha_u = i^*, r_u \in \left(t^{UL}_{s^*}, t^{UL}_{s^*+1}\right)\right\}. \end{aligned} \tag{8}$$

Q consists of the proposals to $C[i^*]$ in the green interval in Figure 4, and L consists of the union of Q with the proposals to $C[i^*]$ in the yellow intervals of Figure 4. L is also the set of recent proposals to side $C[i^*]$ in execution $E_{t^{CL}_{s^*+2}}$.

We now prove (5). Recall from Definition 2.8 and Lemma 2.9 that t_s^{PL} is the unique point in (t_s^{CL}, t_s^{UL}) in which a process executes a successful CAS() on P in line 19, which we refer to as a *pivot-lock point*.

LEMMA 2.29. *Let $a \in \{0,1\}$ and $b \in \{1,\ldots,\lambda\}$. Suppose a* propose() *operation writes some value $u \neq \bot$ to $C[a][b]$ at point r_u, and let $t_{s^*}^{PL}$ be the first pivot-lock point after r_u such that $S_{t_{s^*}^{PL}}.i = a$. If at most one successful* unlock() *operation linearizes in the interval $[r_u, t_{s^*}^{PL}]$, then $C_t[a][b] \neq \bot$ for all $t \in [r_u, t_{s^*}^{PL}]$.*

PROOF. For the purpose of proving a contradiction, suppose there is a point in $(r_u, t_{s^*}^{PL}]$ at which $C[a][b] = \bot$. The value of $C[a][b]$ can change to $\bot$ only as a result of a successful CAS() operation in line 25, 39, or (when $b = \phi$) 21.

Consider the case where the successful CAS() occurs in line 25 [or 39]. There is a point $t_{p@25} \in (r_u, t_{s^*}^{PL}]$ at which some process p executes a successful $C[a][b]$.CAS$(u', \bot)$ operation in line 25 [or 39] for some value $u' \neq \bot$ (possibly $u' = u$). Let $r_{u'} < t_{p@25}$ be the point when u' is written to $C[a][b]$ for the first time. Then $r_{u'}$ is the linearization point of a propose(u') operation. Since the type's specification requires that each value is proposed at most once, we have

$$C_t[a][b] = u' \text{ if and only if } t \in [r_{u'}, t_{p@25}). \tag{9}$$

In particular, since u is written to $C[a][b]$ at point r_u and $t_{p@25}$ is a point in $(r_u, t_{s^*}^{PL})$ at which $C[a][b]$ changes from u' to $\bot$, we have

$$r_u \leq r_{u'} < t_{p@25} < t_{s^*}^{PL}. \tag{10}$$

Prior to $t_{p@25}$, process p reads S in line 16 at point $t_{p@16}$ and reads S again in line 24 at point $t_{p@24}$ [or reads $T[1]$ at point $t_{p@38}$]. Due to the if-condition in line 24 and that S does not experience ABAs, the value of S does not change in $[t_{p@16}, t_{p@24}]$. [In the case of line 38 within the clear() method, by Lemma 2.16 $T[1]$ becomes permanently at least s before any clear(s) operation responds, which by Corollary 2.13 occurs before any unlock() operation can increment $S.\ell$ past s. Therefore, the value of S does not change in $[t_{p@16}, t_{p@38}]$.] Since $t_{p@23}$, the point when p reads u' from $C[a][b]$ in line 23 [or 37], is in that interval, and from the if-condition in line 17 it follows that

$$R_{t_{p@23}} \text{ is locked, and } S_{t_{p@23}}.i = a. \tag{11}$$

By (9) and (10), $t_{p@23} \in [r_{u'}, t_{p@25}] \subseteq [r_u, t_{s^*}^{PL})$. Since an unlock() operation performs a successful CAS() on P in line 19 at point $t_{s^*}^{PL}$, and using Lemma 2.9, R is locked at $t_{s^*}^{PL}$; by the lemma's definition of $t_{s^*}^{PL}$, $S_{t_{s^*}^{PL}}.i = a$. Recall that

- at most one successful unlock() operation linearizes in the interval $[r_u, t_{s^*}^{PL}] \supseteq [t_{p@23}, t_{s^*}^{PL}]$,
- S does not change in the interval $[t_{p@16}, t_{p@24}]$ [or in the interval $[t_{p@16}, t_{p@38}]$],
- $S.i = a$ at both $t_{p@24}$ [or $t_{p@38}$] and $t_{s^*}^{PL}$,
- and $S.i$ changes when and only when a successful unlock() operation linearizes.

Therefore, no successful unlock() operation linearizes in the interval $[t_{p@16}, t_{s^*}^{PL}]$.

Recall that $t_{s^*}^{PL}$ is the point of a successful CAS() on P in line 19. From Observation 2.7 and Lemma 2.9, since S does not change in $[t_{p@16}, t_{p@24}]$ (or $[t_{p@16}, t_{p@38}]$), and $t_{p@19}$ is in that interval, it follows that $P.\ell \geq S.\ell$ immediately after $t_{p@19}$. Since the CAS() on P at $t_{s^*}^{PL}$ succeeds, it follows from Lemma 2.9 that S becomes unlocked and locked again in between $t_{p@19}$ and $t_{s^*}^{PL}$ in order for $P.\ell$ to become less than $S.\ell$ immediately before the pivot-lock point. This contradicts that no unlock() operation linearizes in $[t_{p@16}, t_{s^*}^{PL}]$ and rules out that $C[a][b]$ changed to $\bot$ as a result of a successful CAS() in line 25 or 39.

The successful CAS() operation therefore occurs at an execution of line 21 from some process p at point $t_{p@21}$, which implies that $b = \phi$. From Lemma 2.9 and Claim 2.11, the successful CAS() occurs at point $t_{s'}^{PC}$ in the lock/unlock cycle for some s', while $S.i = a$. Since point $t_{s'}^{PC}$ is after $t_{s'}^{PL}$, the next pivot-lock point at $t_{s^*}^{PL} > t_{p@21}$

occurs after two successful `unlock()` operations linearize, at which point $S.i = a$ again. This contradicts the assumption that at most one successful `unlock()` operation linearizes in $\left[r_u, t^{PL}_{s^*}\right] \supseteq \left[t_{p@21}, t^{PL}_{s^*}\right]$. □

We next show that for any $v \in L$, it is possible that $u_2 = v$ only if, for all proposals $u \in Q$, $\beta_v + 1 \neq \beta_u$.

Lemma 2.30. *If $v \in L$ and $u_2 = v$, then $\beta_v + 1 \neq \beta_u$ for all $u \in Q$.*

Proof. If $\beta_v = \lambda$, the lemma follows trivially from the non-existence of $C[i^*][\lambda + 1]$. Suppose that $v \in L$ and $u_2 = v$, and assume, for contradiction, that there exists some $u \in Q$ such that $\beta_v + 1 = \beta_u$.

Let λ^* be the successful `choose&lock()` call that linearizes at $t^{CL}_{s^*+2}$, and p the process that executes λ^*. From (7), $S_{t^{CL}_{s^*+2}}.i = i^*$, so p reads value v from $C[i^*][j]$, for some $1 \le j \le \lambda$. Since value v is written to C only once, at point r_v, it follows that $\alpha_v = i^*$ and $\beta_v = j$.

Then, since $u \in Q$, value u is written to $C[i^*][\beta_u]$ at point $r_u \in \left(t^{UL}_{s^*}, t^{UL}_{s^*+1}\right)$. We can then apply Lemma 2.29 to obtain that $C_t[i^*][\beta_u] \neq \bot$ for all $t \in \left[r_u, t^{CL}_{s^*+2}\right]$. From Observation 2.6, λ^* is invoked after $t^{UL}_{s^*+1}$. Thus, if p reads the value of $C[i^*][\beta_u]$ while executing λ^*, then it reads a non-$\bot$ value.

The binary search in the `choose&lock()` method in lines 5–9 begins with a range of length λ, which equals a power of two times $\phi - 1$, and stops when the range length becomes at most $\phi - 1$. Since the range is divided by two with each iteration, the range $[low, high)$ has length exactly $\phi - 1$ by line 10, and $high$ is one more than a multiple of $\phi - 1$. If β_v is neither a multiple of $\phi - 1$ nor one plus a multiple of $\phi - 1$, then p reads v from $C[i^*][\beta_v]$ in line 13 and then reads $C[i^*][\beta_v + 1] = C[i^*][\beta_u]$ in the next for-loop iteration in line 13. If β_v is one plus a multiple of $\phi - 1$, then either p reads v in line 8 and the if-condition in line 11 evaluates to false, or p reads v in line 10 and the if-condition in line 11 evaluates to true. In both cases, p reads $C[i^*][\beta_v + 1] = C[i^*][\beta_u]$ in its first execution of line 13. In all cases discussed so far, by the previous paragraph $C[i^*][\beta_u]$ contains a non-$\bot$ value when p reads it, and by line 14 p does not choose v as the new interpreted agreement-value, which is a contradiction.

Finally, if β_v is exactly a multiple of $\phi - 1$, then $\beta_v + 1 = high = \beta_u$ immediately after line 11. Since $high$ is initialized to $\lambda + 1$, and $\beta_u \le \lambda$, $high$ first changes to β_u in either line 9 or line 11. In both cases, p reads $\bot$ from $C[i^*][\beta_u]$ before setting $high$ to β_u. This once again contradicts that p reads a non-$\bot$ value from $C[i^*][\beta_u]$. □

The next key lemma states that the conditional probability that a proposal v with $r_v > t^{PL}_{s^*}$ is written to entry $C[i^*][j]$, given $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$, the entries where all the other proposed values were written, and that $\alpha_v = i^*$, is exactly π_j.

Lemma 2.31. *If $t^{PL}_{s^*} < r_v < t^{CL}_{s^*+2}$ and $\alpha_v = i^*$, then for any $1 \le j \le \lambda$,*

$$\Pr\left[\beta_v = j \;\middle|\; \mathcal{T}\left(t^{CL}_{s^*+2}\right), \left\{(u, \beta_u) : u \in V \setminus \{v\} \text{ and } r_u < t^{CL}_{s^*+2}\right\}\right] = \pi_j. \tag{12}$$

Proof. Recall that $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$ includes neither the outcome of process's coin flips nor the contents of C, but it does include the values α_v for all proposals v such that $r_v < t^{CL}_{s^*+2}$. Therefore, the probability that a `propose(v)` operation selects $\beta_v = j$, given $\mathcal{T}(r_v)$, is π_j. It thus suffices to show that the adversary does not learn anything about β_v in the interval $\left[r_v, t^{CL}_{s^*+2}\right]$ for all v such that $t^{PL}_{s^*} < r_v < t^{CL}_{s^*+2}$ and $\alpha_v = i^*$.

The only information included in the trace that can be directly affected by the choice of β_v consists of the values of $S.val$ and $P.pval$. From Lemma 2.9 and Definition 2.25, the only points in $\left[r_v, t^{CL}_{s^*+2}\right]$ in which $S.val$ and $P.pval$ can change and be visible to the adversary by $t^{CL}_{s^*+2}$ are $t^{PC}_{s^*}$, $t^{CL}_{s^*+1}$, $t^{PL}_{s^*+1}$, and $t^{PC}_{s^*+1}$. From Observation 2.7, $P.pval = \bot$ at $t^{PC}_{s^*}$ and $t^{PC}_{s^*+1}$, so it does not reveal anything about any proposals. Using Lemma 2.9, the values of u_1 and $P_{t^{PL}_{s^*+1}}.pval$ are read from $C[1 - i^*]$. Therefore, any proposal written to $C[i^*]$ in $\left(t^{PL}_{s^*}, t^{CL}_{s^*+2}\right)$ does not affect the selection of u_1 and $P_{t^{PL}_{s^*+1}}.pval$. The adversary thus does not learn anything about the value β_v of proposals to $C[i^*]$ written in the interval $\left(t^{PL}_{s^*}, t^{CL}_{s^*+2}\right)$, and so the lemma follows. □

Recall from line 1 that $\pi_j = 2^{-j}$, for $j \in \{1, \ldots, \lambda - 1\}$, and $\pi_\lambda = 2^{-\lambda+1}$.

Lemma 2.32. *Let $X_1, \ldots, X_k$ be random variables that take values in the set $\{1, 2, \ldots, \lambda\}$. Suppose that for each $1 \le i \le k-1$ and $1 \le j \le \lambda$, $\Pr[X_i = j \mid X_1, \ldots, X_{i-1}] = \pi_j$. Then, $\Pr[X_k + 1 \neq X_i$ for all $i \in \{1, \ldots, k-1\}] < 7/\max\{7, k\} + 2^{-\lambda+1}$.*

Proof. Suppose that $k \ge 8$ and $\lambda \ge 2$, otherwise the claim holds trivially. Let $\pi_{\lambda+1} = 0$ for convenience.

$$\begin{aligned}
&\Pr[X_k + 1 \neq X_i \text{ for all } i \in \{1, \ldots, k-1\}] \\
&= \sum_{j=1}^{\lambda} \Pr[X_k = j \text{ and } X_i \neq j+1 \text{ for all } i \in \{1, \ldots, k-1\}] \\
&= \sum_{j=1}^{\lambda} \pi_j \cdot \left(1 - \pi_{j+1}\right)^{k-1} \\
&= \sum_{j=1}^{\lambda-2} 2^{-j}\left(1 - 2^{-j-1}\right)^{k-1} + 2^{-\lambda+1}\left(1 - 2^{-\lambda+1}\right)^{k-1} + 2^{-\lambda+1} \cdot 1 \\
&< \sum_{j=1}^{\infty} 2^{-j}\left(1 - 2^{-j-1}\right)^{k-1} + 2^{-\lambda+1} \\
&< \sum_{j=1}^{\infty} 2^{-j} e^{-2^{-j-1}(k-1)} + 2^{-\lambda+1} \\
&= \sum_{j'=-\lfloor \log(k-1) \rfloor}^{\infty} 2^{-j'-\lfloor \log(k-1) \rfloor - 1} e^{-2^{-j'-\lfloor \log(k-1) \rfloor - 2}(k-1)} + 2^{-\lambda+1} \\
&< \sum_{j' \in \mathbb{Z}} 2^{-j'-\log(k-1)} e^{-2^{-j'-\log(k-1)-2}(k-1)} + 2^{-\lambda+1} \\
&= \frac{1}{k-1} \sum_{j' \in \mathbb{Z}} 2^{-j'} e^{-2^{-j'-2}} + 2^{-\lambda+1} \\
&< \frac{6}{k-1} + 2^{-\lambda+1} \\
&< 7/k + 2^{-\lambda+1}
\end{aligned}$$

where for the last inequality we used that $k \ge 8$. □

Lemma 2.33.

$$\Pr\left[u_2 = v \,\middle|\, \mathcal{T}\left(t^{CL}_{s^*+2}\right)\right] < \frac{11}{\max\{11, |Q|\}} + \frac{3}{n}.$$

Proof. From Lemma 2.31, conditionally on $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$ and that $r_v > t^{PL}_{s^*}$, the collection of random variables $\beta_u, u \in Q \cup \{v\}$, satisfy the conditions of Lemma 2.32, which gives

$$\Pr[\beta_v + 1 \neq \beta_u \text{ for all } u \in Q] < 7/|Q \cup \{v\}| + 2^{-\lambda+1}.$$

The analysis for the case where $t^{CL}_{s^*} < r_v < t^{PL}_{s^*}$ is slightly different, because at point $t^{PL}_{s^*}$ the adversary learns the value of $P.pval$, which can be influenced by previous proposals written to $C[i^*][\phi]$. If $P_{t^{PL}_{s^*}}.pval \neq \bot$, then by Claim 2.21 v is not present in C by point $t^{UL}_{s^*}$, and so by Observation 2.6 $u_2 \neq v$ with probability 1 given $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$.

Otherwise, if $P_{t^{PL}_{s^*}}.pval = \bot$, then by Claim 2.11 the adversary knows that no proposal in $\left(t^{CL}_{s^*}, t^{PL}_{s^*}\right)$ writes to $C[i^*][\phi]$, and in particular that v is not written to $C[i^*][\phi]$. Since the adversary does not know anything else about β_v, using a slightly modified argument from Lemma 2.31 it follows that

$$\Pr\left[\beta_v = j \,\middle|\, \mathcal{T}\!\left(t^{CL}_{s^*+2}\right), \left\{(u, \beta_u) : u \in V \setminus \{v\} \text{ and } r_u < t^{CL}_{s^*+2}\right\}, P_{t^{PL}_{s^*}}.pval = \bot\right] = \frac{\pi_j}{1 - \pi_\phi}.$$

Since the distribution for proposals in Q are unaffected, we can modify the result from Lemma 2.32 by multiplying the upper bound by $1/(1 - \pi_\phi)$. From

$$\begin{aligned}\pi_\phi &= 2^{-\lceil \log\log n \rceil} \\ &\le 2^{-\log\log n} \\ &= 1/\log n, \\ 1/(1 - \pi_\phi) &\le 1/(1 - 1/\log n).\end{aligned}$$

Therefore, the upper bound of the probability that $\beta_v + 1 \neq \beta_u$ for all $u \in Q$ for the case where $t^{CL}_{s^*} < r_v < t^{PL}_{s^*}$ and $P_{t^{PL}_{s^*}}.pval = \bot$ is

$$\begin{aligned}\frac{7/\max\{7, |Q \cup \{v\}|\} + 2^{-\lambda+1}}{1 - 1/\log n} &\le \frac{7/\max\{7, |Q \cup \{v\}|\} + 2^{-\log n+1}}{1 - 1/\log n} \\ &= \frac{7/\max\{7, |Q \cup \{v\}|\} + 2/n}{1 - 1/\log n}.\end{aligned}$$

Since the result is trivial when $|Q \cup \{v\}| \le 7$, we can use that $\log n \ge 3$ and $|Q \cup \{v\}| \ge |Q|$ to simplify the upper bound to

$$\frac{7/\max\{7, |Q \cup \{v\}|\} + 2/n}{1 - 1/\log n} \le \frac{11}{\max\{11, |Q|\}} + \frac{3}{n}.$$

Combining these cases with Lemma 2.30 proves the lemma. □

To complete the proof of (5), we show that $|Q|$ satisfies the given requirements for the random variable g_{i^*} in (4).

Claim 2.34.

$$0 \le |Q| \le p_{s^*} \text{ and } \mathbf{E}\left[|Q| \,\middle|\, E_{t^{UL}_{s^*}}\right] = (1/2) \cdot \mathbf{E}\left[p_{s^*} \,\middle|\, E_{t^{UL}_{s^*}}\right].$$

Proof. Since Q is a subset of the set of proposals in $\left(t^{UL}_{s^*}, t^{UL}_{s^*+1}\right)$, it follows that $0 \le |Q| \le p_{s^*}$.

Next, consider the random variable Δ_t for $t \in (t^{UL}_{s^*}, t^{UL}_{s^*+1}]$, where we define Δ_t as the number of proposals to $C[i^*]$ in $\left(t^{UL}_{s^*}, t\right]$ minus the number of proposals to $C[1 - i^*]$ in $\left(t^{UL}_{s^*}, t\right]$, i.e., $\Delta_{t^{UL}_{s^*+1}} = |Q| - (p_{s^*} - |Q|)$. For any proposal $u \in W$, the value of α_u is chosen uniformly at random between 0 and 1 in line 1 at point r_u, independently of all events prior to r_u. Since we define a step as consisting of a shared memory operation and all preceding local operations by the same process, lines 1 and 2 of the `propose()` operation take place in a single step. Thus, the execution does not contain any information about the chosen value of α_u until the point r_u when u is written to $C[\alpha_u]$ in line 2, so it follows that $\mathbf{E}[\Delta_t \mid E_{t-1}] = \Delta_{t-1}$, making Δ_t a martingale. Since executions of line 2 change the absolute value of Δ_t by 1 and all other lines do not change Δ_t, we also have that $|\Delta_t - \Delta_{t-1}| \le 1$. Recall from (R) that the adversary schedules a bounded number of method calls in between $t^{UL}_{s^*}$ and $t^{UL}_{s^*+1}$. We can therefore

use the Optional Stopping Theorem, which tells us that $\mathbf{E}\left[\Delta_{t^{UL}_{s^*+1}} \;\middle|\; E_{t^{UL}_{s^*}}\right] = 0$. Therefore,

$$\begin{aligned}
\mathbf{E}\left[|Q| \;\middle|\; E_{t^{UL}_{s^*}}\right] - \mathbf{E}\left[p_{s^*} - |Q| \;\middle|\; E_{t^{UL}_{s^*}}\right] &= 0 \\
2 \cdot \mathbf{E}\left[|Q| \;\middle|\; E_{t^{UL}_{s^*}}\right] &= \mathbf{E}\left[p_{s^*} \;\middle|\; E_{t^{UL}_{s^*}}\right] \\
\mathbf{E}\left[|Q| \;\middle|\; E_{t^{UL}_{s^*}}\right] &= (1/2) \cdot \mathbf{E}\left[p_{s^*} \;\middle|\; E_{t^{UL}_{s^*}}\right].
\end{aligned}$$

□

It remains to show (6). Suppose that $|Q| > 0$, otherwise (6) is immediate.

Lemma 2.35. *If $u_2 = \bot$, then for every proposal $u \in Q$, $\beta_u > \phi$.*

Proof. For the purpose of proving a contradiction, suppose that $u_2 = \bot$ and that, for some proposal u' to $C[i^*]$ in $\left(t^{UL}_{s^*}, t^{UL}_{s^*+1}\right)$, $1 \le \beta_{u'} \le \phi$. Using Lemma 2.29, no value proposed to $C[i^*]$ after $t^{UL}_{s^*}$ can be replaced with $\bot$ in the interval $\left(t^{UL}_{s^*}, t^{CL}_{s^*+2}\right]$. Thus, if λ^* reads $C[i^*][\beta_{u'}]$, it reads a non-$\bot$ value in that position.

We first show that $low = 1$ after λ^* executes line 11. Suppose for the purpose of proving a contradiction that λ^*'s local value of low is greater than 1 by the end of line 11. Then, λ^* reads a non-$\bot$ value from $C[i^*][mid]$ in line 8 and sets low to mid in line 9, for some $1 < mid < \lambda$. Thus, λ^* sets v to a non-$\bot$ value in line 9 and does not overwrite low and v in line 11. The conditional in line 14 guarantees that v does not change to $\bot$ again, so $u_2 \neq \bot$; by contradiction, $low = 1$ by the end of line 11.

Since $low = 1$ by the end of λ^*'s execution of line 11, λ^* reads $\bot$ from $C[i^*][\phi]$ at least once. λ^* can read $\bot$ from $C[i^*][mid]$ every time it executes line 8, in which case $mid = \phi$ in the last while-loop iteration from λ being a power of two times $\phi - 1$. Otherwise, λ^* reads $\bot$ from $C[i^*][\phi]$ in line 11, and resets low to 1 and v to the value it reads in $C[i][1]$. Therefore, $\beta_{u'} \neq \phi$. Moreover, lines 10–11 and the for loop in lines 12–14 check all values of $C[i^*][j]$ for $1 \le j < \phi$, and set v to the last non-$\bot$ entry found. Since $u_2 = \bot$, λ^* reads $\bot$ from all $C[i^*][j]$ entries in this range. Since no value in $C[i^*]$ is erased in $\left(t^{UL}_{s^*}, t^{CL}_{s^*+2}\right]$, this contradicts that $1 \le \beta_{u'} \le \phi$. □

From Lemmas 2.31 and 2.35,

$$\begin{aligned}
\Pr\left[u_2 = \bot \;\middle|\; \mathcal{T}\left(t^{CL}_{s^*+2}\right)\right] &\le \Pr\left[\forall u \in Q, \beta_u > \phi \;\middle|\; \mathcal{T}\left(t^{CL}_{s^*+2}\right)\right] \\
&= \prod_{u \in Q} \left(\sum_{j=\phi+1}^{\lambda} \pi_j\right) \\
&= \left(2^{-\phi}\right)^{|Q|} \\
&\le \left(\frac{1}{\log n}\right)^{|Q|},
\end{aligned}$$

This proves (6) and competes the proof of Theorem 2.27.

2.4.2 Proof of Theorem 2.28. Recall that for each value $v \in W$ (where $W \subset V$ is the set of values proposed in E), v is written to $C[\alpha_v][\beta_v]$ at point r_v, and if $v \in V \setminus W$, then $r_v = \infty$.

To set up the proof of Theorem 2.28, we define two subtypes of outdated proposals. Intuitively,

- a *dangerous* proposal might not be cleared when it should be, and the adversary can tell it exists with high probability, and
- an *inconvenient* proposal might not be cleared when it should be, but the adversary can't tell that it exists with high probability.

Definition 2.36 (Offset point)**.** An *offset point* is a point t_s^{PC} for a non-negative integer s such that either $P_{t_s^{PL}}.pval \neq \bot$, $2s - 1 \bmod \lambda \geq \lambda - 4$, or $s = 0$. We say that t_s^{PC} is an offset point for side α if $S_{t_s^{PL}}.i = \alpha$.

Remark 2.37. Since all the conditions for a point to be an offset point are visible in $\mathcal{T}()$, the adversary is aware of all offset points.

Observation 2.38. *If $C[\alpha][\phi]$ changes from a non-$\bot$ value to $\bot$ at point t, then t is an offset point for side α.*

Proof. From Claim 2.11, $t = t_s^{PC}$ for some s, and $C[\alpha][\phi]$ was equal to $P_{t_s^{PL}}.pval$ immediately before point t. Therefore, $P_{t_s^{PL}}.pval \neq \bot$, which makes t satisfy Definition 2.36. □

Observation 2.39. *Suppose that $t_s^{PC} < \infty$ is an offset point for side α. Then, any value that is in $C[\alpha][\beta]$ for some $\beta \neq \phi$ before point t_s^{PC}, as well as any value in $C[\alpha][\phi]$ before point t_s^{CL}, is no longer present in C by point t_s^{UL}.*

Proof. This follows immediately from the combination of Definition 2.36, Lemma 2.12, and Claim 2.21. □

Definition 2.40 (Dangerous proposals)**.** Proposal $v \in W$ is *dangerous* if $r_v < \infty$, $\beta_v > 5\phi$, $C_{r_v}[\alpha_v][\phi] = \bot$, and the last offset point on side α_v before r_v, $t_{s_v}^{PL}$, is such that $\left|\left\{u : t_{s_v}^{PL} < r_u \leq r_v\right\}\right| > 6\log^2 n$.

Definition 2.41 (Inconvenient proposals)**.** Proposal $v \in W$ is *inconvenient* if $r_v < \infty$, $\beta_v > 5\phi$, and $C_{r_v}[\alpha_v][\phi] = \bot$, but it is not dangerous (i.e., it is one of the first $6\log^2 n$ proposals after the last offset point before r_v on side α_v).

Lemma 2.42 (Frequency of dangerous proposals). *Let $s \geq 0$ and $\alpha \in \{0, 1\}$. The probability of the event O_s that t_s^{PC} is the offset point of some dangerous proposal is* $\Pr\left[O_s \;\middle|\; E_{t_s^{PC}}\right] < 1/n^2$.

Proof. By Definition 2.40 and Observation 2.38, value v proposed at point $r_v > t_s^{PC}$ is dangerous with t_s^{PC} as its offset point only if there are more than $\left(6\log^2 n + 1\right)$ proposals in the interval $\left[t_s^{PC}, r_v\right]$, and no proposal in the interval writes to $C[\alpha][\phi]$. Accordingly, the probability that t_s^{PC} is the offset point for some dangerous proposal, given $E_{t_s^{PC}}$, is at most the probability that the first $\left(6\log^2 n + 1\right)$ proposals after t_s^{PC} do not write to $C[\alpha][\phi]$. (If $C_{t_s^{PL}}[\alpha][\phi] \neq \bot$, then the conditional probability is zero by Claim 2.11.)

For each of the first $\left(6\log^2 n + 1\right)$ proposed values u after point t_s^{PC}, the probability that u is proposed to $C[\alpha][\phi]$ given $\mathcal{T}(r_u - 1)$ is exactly $2^{-\phi-1}$ and independent of other proposals. Thus, the probability that all the $6\log^2 n + 1$ proposals do not write to $C[\alpha][\phi]$ given $E_{t_s^{PC}}$ is less than

$$\begin{aligned}
\left(1 - 2^{-\phi-1}\right)^{6\log^2 n} &= \left(1 - 2^{-\lceil\log\log n\rceil-1}\right)^{6\log^2 n} \\
&< \left(1 - 2^{-\log\log n-2}\right)^{6\log^2 n} \\
&= \left(1 - \frac{1}{4\log n}\right)^{6\log^2 n} \\
&= \left(1 - \frac{1}{4\log n}\right)^{4\log n\cdot(3/2)\log n} \\
&< 1/e^{(3/2)\log n} \\
&= 1/n^{(3/2)\log e} \\
&< 1/n^2.
\end{aligned}$$

□

Lemma 2.43 (Frequency of inconvenient proposals). *Let E be an execution such that t_s^{PC} is an offset point on side α, and let $s' > s$ be such that there is no offset point on side α in the interval $\left(t_s^{PC}, t_{s'+2}^{CL}\right]$ and that $S_{t_{s'}^{CL}}.i = \alpha$.*

Then, the probability that there exists an inconvenient proposal v such that $\alpha_v = \alpha$ and $r_v > t_s^{PC}$, given $\mathcal{T}\left(t_{s'+2}^{CL}\right)$, is at most $6/\log^2 n$.

Proof. Consider for this lemma a stronger adversary that knows the value β_v of all proposals v that write to $C[1-\alpha]$, to $C[\alpha][\beta_v]$ with $\beta_v \leq \phi$, or that occur before t_s^{PC}. Thus, all the proposals that the adversary does *not* know about occur after t_s^{PC} to $C[\alpha][\beta_v]$ with $\beta_v > \phi$. The value of these unknown proposals do not affect $P.pval$ by point $t_{s'+2}^{CL}$, since they do not write to $C[i][\phi]$ for any $i \in \{0,1\}$. Similarly, since they do not write to $C[i][\phi]$, they do not affect whether `clear()` is called, and hence do not affect $T[1]$. The values of unknown proposals also do not affect $S.val$ before point $t_{s'+2}^{CL}$, which we show by contradiction.

Suppose that an unknown proposal affects $S.val$ before point $t_{s'+2}^{CL}$: this can only happen if a successful `choose&lock()` operation op in $\left(t_s^{PC}, t_{s'+2}^{CL}\right)$ chooses an unknown proposal as the new interpreted agreement-value for side α. Since unknown proposals v have $\beta_v > \phi$, it follows that op's value of low is greater than 1 after line 11. Therefore, in line 11, op reads $C[\alpha][\phi] \neq \bot$, which by Observation 2.6 is after t_s^{PC} and before op's linearization point in line 15 at point $t_{s''}^{CL} < t_{s'+2}^{CL}$, so $s'' < s'+2$. By Claim 2.11, $C[\alpha][\phi] \neq \bot$ throughout the interval after op's read of $C[\alpha][\phi]$ until just before $t_{s''}^{PC} > t_{s''}^{PL}$. By Definition 2.8 the successful `CAS()` on P at point $t_{s''}^{PL}$ sets $P.pval$ to a non-$\bot$ value, and by Definition 2.36 this makes $t_{s''}^{PC} < t_{s'+2}^{CL}$ an offset point. This contradicts the assumption that there is no offset point in the interval $\left(t_s^{PC}, t_{s'+2}^{CL}\right]$.

Since the remaining visible fields are $S.\ell$, $P.\ell$, and $S.i$, none of which are affected by proposed values, the adversary cannot learn about the location of any unknown proposals by $t_{s'+2}^{CL}$.

From Definition 2.41, an inconvenient proposal v with $\alpha_v = \alpha$ and $r_v > t_s^{PC}$ has $\beta_v > 5\phi$ and is one of the first $3\log^2 n$ proposals to $C[\alpha]$ after t_s^{PC}. For any given unknown proposal, its probability of writing to $C[\alpha][j]$ such that $j > 5\phi$, given the stronger adversary's knowledge at point $t_{s'+2}^{CL}$, is $2^{-5\phi-1}/2^{-\phi-1} = 2^{-4\phi}$. Using the union bound, the probability that any of the first $6\log^2 n$ proposals after PCs writes to $C[\alpha][j]$ such that $j > 5\phi$ is at most

$$
\begin{aligned}
6\log^2 n \cdot 2^{-4\phi} &= 6\log^2 n \cdot 2^{-4\lceil \log\log n \rceil} \\
&\leq 6\log^2 n \cdot 2^{-4\log\log n} \\
&= \frac{6\log^2 n}{\log^4 n} \\
&= \frac{6}{\log^2 n}.
\end{aligned}
$$

□

We can now prove Theorem 2.28. First, we specify that the 'bad event' after $E_{t_{c\lambda/2-1}^{CL}}$ is the event where some proposal in the interval $\left(t_{c\lambda/2-1}^{CL}, t_{s^*}^{CL}\right)$ is dangerous and present in C by point $t_{s^*+1}^{UL}$.

Let $s_o \geq 0$ be the largest integer at most s^* such that $t_{s_o}^{PC}$ is an offset point for side i^*. By Definition 2.36 and Lemma 2.9, $s_o \geq c\lambda/2 - 1$. From Observation 2.39, any value written to $C[i^*][j]$ with $j \neq \phi$ before $t_{s_o}^{PC}$ and any value written to $C[i^*][\phi]$ before $t_{s_o}^{CL}$ is not present in C after point $t_{s_o}^{UL}$.

We now consider a stronger adversary, similar to the one described in the proof of Lemma 2.43. For each proposal with $\alpha_v = 1 - i^*$ or $\beta_v \leq \phi$, we reveal the value β_v at the time r_v of the proposal. Additionally, at each offset point, we reveal the value β_v for all proposals made before that offset point.

Since the result in Lemma 2.42 is conditioned on the full execution E up to the relevant offset point, it also holds for this stronger adversary: thus, the probability that $t_{s_o}^{PC}$ is the offset point for a dangerous proposal, given the stronger adversary's knowledge at point $t_{s_o}^{PC}$, is at most $1/n^2$. Since any dangerous proposal in C at point $t_{s^*+1}^{UL}$ has $t_{s_o}^{PC}$ as its offset point, it follows that the probability of the bad event, given $E_{t_{s_o}^{PC}} \supseteq E_{t_{c\lambda/2-1}^{CL}}$, is at most $1/n^2$.

By Claim 2.21, all values v such that $r_v < t^{PC}_{s_o}$, $\alpha_v = i^*$, and $\beta_v > 5\phi$ are not in C by point $t^{UL}_{s_o}$. By the same claim, all values v such that $r_v < t^{CL}_{s^*}$, $\alpha_v = i^*$, and $\beta_v \leq \phi$ are not in C by point $t^{UL}_{s^*}$. Thus, the only outdated values in $C[i^*]$ at point $t^{UL}_{s^*+1}$ are values v such that $t^{PC}_{s_o} < r_v < t^{CL}_{s^*}$, $\alpha_v = i^*$, and $\beta_v > 5\phi$. If $s_o = s^*$, then $t^{CL}_{s^*} < t^{PC}_{s_o}$ and so there are no outdated values in $C[i^*]$ at point $t^{UL}_{s^*+1}$. Otherwise, conditioned on the bad event not happening, by Claim 2.21 the set of outdated values in $C[i^*]$ at point $t^{UL}_{s^*+1}$ is exactly the set of inconvenient proposals written in the interval $\left(t^{PC}_{s_o}, t^{CL}_{s^*}\right)$. By Lemma 2.43, the probability that this set is non-empty, given $\mathcal{T}\left(t^{CL}_{s^*+2}\right)$, is at most $6/\log^2 n$. Theorem 2.28 thus follows from Observation 2.6 and that s_2 is read from $C[\alpha]$ by a successful `choose&lock()` operation.

2.5 Step Complexity

The `propose()` and `read()` methods take a single shared memory step each, and the `choose&lock()` method's worst-case step complexity is in $O(\log\log n)$. The `clear()` method takes a variable number of shared memory steps (and hence so does the `unlock()` method), so we discuss the expected amortized step complexity of `clear()` next.

2.5.1 Clear Method. The following lemma proves a result first derived by Michael Luby [20] and used similarly to Martel and Subramonian [22]. We include this proof in order for the paper to be self-contained.

In this section, we use the term *colouring* to mean irreversibly changing a node from an *uncoloured* state to a *coloured* state, as opposed to the definition used in the vertex colouring problem.

Lemma 2.44. *Let G be a directed graph consisting of the disjoint union of a set of m paths, each consisting of at most d nodes. We want to colour all the nodes according to the following process: in each step, for each node we select it with a probability greater than or equal to a parameter $0 < q \leq 1$. The probability of selecting a particular node in a given step is independent of the node selections in prior steps, and not necessarily independent of other node selections in the same step. For every node selected in a given step, if it is the first node in the directed path or its predecessor in the path was coloured before the step began, then we colour the node.*

The expected number of steps to colour all nodes is in $O((d + \log m)/q)$.

Proof. For any fixed m the lemma follows from the fact that it takes at most d/q steps in expectation to colour a single path of d nodes, so let $m \geq 3$. We show the lemma by proving that the probability that the process takes more than $c(d + \ln m)/q$ steps is exponentially small in c for $c \geq 4$.

First, consider a single directed path of at least $c(d + \ln m)/q$ nodes. In each step, at most one node may become coloured: the first uncoloured node in the path is coloured in a step if and only if that node is selected in the step. Since node selections in a given step are independent of prior step selections, in $c(d + \ln m)/q$ steps the expected number of colourings is at least $c(d + \ln m)$.

Let X denote a random variable counting the number of steps in which a node is coloured after taking $c(d + \ln m)/q$ steps in a path of at least $c(d + \ln m)/q$ nodes. We will find an upper bound for the probability that $X \leq d - 1$. Since in a path of d nodes we stop colouring once we've coloured d nodes, the probability that $X \leq d - 1$ is equal to the probability that we do not fully colour a path of d nodes. This probability is also an upper bound for the probability of not fully colouring a path of less than d nodes.

Using Chernoff bounds,

$$\begin{aligned}\Pr[X \leq d - 1] &= \Pr\left[X \leq \frac{d-1}{c(d + \ln m)} c(d + \ln m)\right] \\ &= \Pr\left[X \leq \left(1 - \frac{c(d + \ln m) - (d-1)}{c(d + \ln m)}\right) c(d + \ln m)\right]\end{aligned}$$

$$
\begin{aligned}
&\leq \Pr\left[X \leq \left(1 - \frac{c(d+\ln m) - (d+\ln m)}{c(d+\ln m)}\right) c(d+\ln m)\right] \\
&= \Pr\left[X \leq \left(1 - \frac{(c-1)(d+\ln m)}{c(d+\ln m)}\right) c(d+\ln m)\right] \\
&= \Pr\left[X \leq \left(1 - \frac{c-1}{c}\right) c(d+\ln m)\right] \\
&\leq \exp\left(-c(d+\ln m)\left(\frac{c-1}{c}\right)^2 /2\right) \\
&= \exp\left(-\frac{(c-1)^2 (d+\ln m)}{2c}\right) \\
&= \left(m^{-\frac{c-1}{2c}} \exp\left(-\frac{c-1}{2c} d\right)\right)^{c-1}.
\end{aligned}
$$

Using that $c \geq 4$, it follows that $\frac{c-1}{2c} \geq 3/8$, so

$$
\left(m^{-\frac{c-1}{2c}} \exp\left(-\frac{c-1}{2c} d\right)\right)^{c-1} \leq \left(m^{-\frac{3}{8}} \exp\left(-\frac{3}{8} d\right)\right)^{c-1}.
$$

Now consider all m paths. Using the union bound and that each path has at most d nodes, the probability that at least one path does not have all nodes coloured is at most

$$
\begin{aligned}
m\left(m^{-\frac{3}{8}} \exp\left(-\frac{3}{8} d\right)\right)^{c-1} &= m^{1-\frac{3}{8}(c-1)} \exp\left(-\frac{3}{8} d\right)^{c-1} \\
&\leq m^{-\frac{1}{8}} \exp\left(-\frac{3}{8} d\right)^{c-1},
\end{aligned}
$$

again using that $c \geq 4$. Finally, the expected number of steps to colour all nodes is at most

$$
\begin{aligned}
&\frac{d+\ln m}{q}\left(4 + \sum_{c=4}^{\infty} Pr[\text{more than } c(d+\ln m)/q \text{ steps}]\right) \\
&\leq \frac{d+\ln m}{q}\left(4 + \sum_{c=4}^{\infty} m^{-\frac{1}{8}} \exp\left(-\frac{3}{8} d\right)^{c-1}\right) \\
&= \frac{d+\ln m}{q}\left(4 + \frac{1}{m^{\frac{1}{8}} \exp\left(\frac{9}{8} d\right)\left(1 - \exp\left(-\frac{3}{8} d\right)\right)}\right) \\
&\leq 5\frac{d+\ln m}{q}
\end{aligned}
$$

where in the last line we use that $m \geq 3$ and $d \geq 1$. ◻

Lemma 2.45. *Let T be a rooted binary tree with depth d, and suppose that we want to colour all nodes of T using a slightly modified form of the process in Lemma 2.44: again, in each step each node has a probability of selection of at least q, where the probability is independent of prior steps' selections and is not necessarily independent of other node selections in the same step. We colour a selected node in a step if and only if either it is a leaf of the tree or all of its children were coloured before the current step. The expected number of steps to colour the root of T using this process is in $O(d/q)$.*

Proof. We prove this lemma by showing a coupling between the process of colouring the tree T and the process of colouring a graph G to which Lemma 2.44 applies. For each directed path in T from a leaf to the root, we couple it with a disjoint directed path in G following the same direction (from leaf to root) and with the same number of nodes. We thus couple each non-leaf node in T with multiple nodes in G, one for each path from a leaf of T to its root that passes through that node.

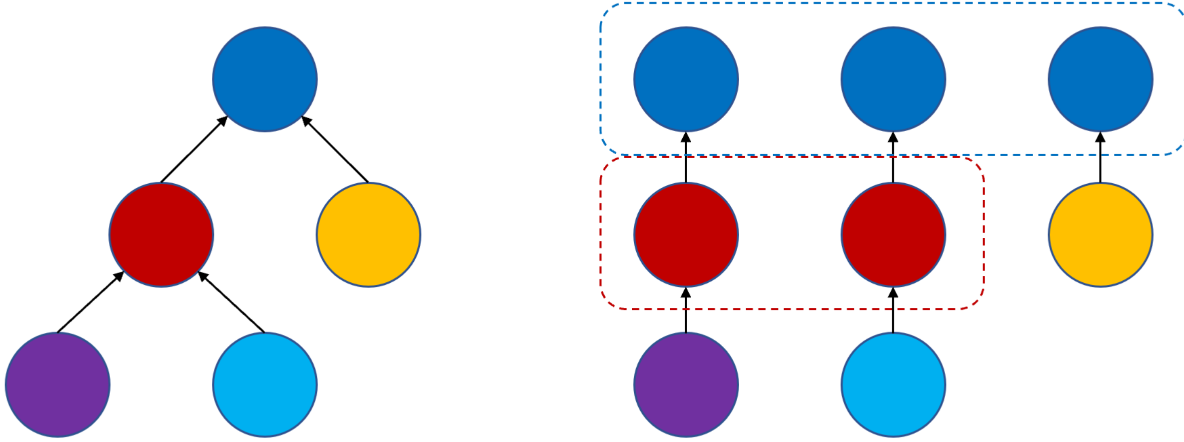

Fig. 5. Coupling used in the proof of Lemma 2.45

We couple the process of colouring T with the process of colouring G as follows: for every step in which we select a node N in T, we select *all* nodes associated to N in T, and call this set of associated nodes N_G. As in Lemma 2.44, we colour a node in G in a step if we select the node and if all its children (if applicable) were coloured in a previous step. This process of colouring G satisfies the assumptions of Lemma 2.44, because each node in G is selected with probability q independently of selections in prior steps (recall that neither lemma requires independence of selections *within* a step).

We will now show by induction from leaves towards the root that the following statement holds for all nodes N in T:

$$N \text{ is coloured in a given step if and only if all nodes in } N_G \text{ are coloured in the same step.} \tag{13}$$

For the base case, suppose that N is a leaf node. Then N_G is a singleton set consisting of the first node in its path, and this node is selected in G exactly when N is selected in T. Since we colour leaf nodes in T as soon as we select them, and we colour initial nodes on paths in G as soon as we select them, (13) follows for any leaf node N.

For the inductive case, suppose that N is not a leaf. Suppose that (13) holds for the children of N in T, i.e., for each child of N in T, the child is coloured if and only if all its associated nodes in G are also coloured.

Consider the first step s in which N becomes coloured: in this step we select N in T, and by definition of the coupling we also select all its associated nodes (the set N_G) in G. Moreover, both the children of N are coloured before step s, and by the inductive hypothesis all the children of all nodes in N_G are coloured before step s. It therefore follows that all nodes in N_G are also coloured in step s.

Conversely, consider the first step s in which all nodes in N_G become coloured: since we select at least one node in N_G in step s, by the coupling we select N in T and all nodes in N_G in G. Since we finish colouring all nodes N_G in step s, all of their children are coloured before step s, and by the inductive hypothesis both children of N are coloured before step s. Since we selected N in T in step s, we therefore also colour N in step s.

It therefore follows from the above induction that all nodes are coloured in T exactly when all nodes are coloured in G, and (from the tree structure) that the last node we colour in T is the root. The lemma thus follows by applying Lemma 2.44 to G, using that the number of paths m in G is less than 2^d. □

In the context of the `clear()` method, colouring corresponds to setting the sequence number stored in a node to a value at least ℓ; from Lemma 2.16, this is an irreversible process.

Claim 2.46. *If a process executes line 40 as part of a* `clear(ℓ)` *operation, and the second argument of the* `CAS()` *on $T[j]$ equals ℓ, then, $T[j] \geq \ell$ immediately after the* `CAS()` *operation.*

Proof. Suppose, for the purpose of proving a contradiction, that $T[j] < \ell$ immediately after a process p executes $T[j]$.`CAS`$(node, \ell)$ at point t in line 40. This `CAS()` fails, which implies that $T[j]$ changed from *node* after p's last execution of line 34 before t, at point $t' < t$. Because only line 40 can change $T[j]$, another process p' executes a successful `CAS()` on $T[j]$ at a point $t^* in\ (t', t)$, with a second argument $\ell' < \ell$. From Claim 2.15, a `clear(ℓ')` operation is invoked before t^*, and from (1) a `clear(ℓ')` operation responds before p's invocation, at some point $t_R < t'$. Finally, from Corollary 2.17 $T[j] = \ell'$ at point t_R, and since $T[j]$ does not exhibit ABAs due to Lemma 2.16, this contradicts that $T[j]$ changes to ℓ' at point $t^* > t' > t_R$. □

Adversary. We use the same ip-aware adversary defined in Section 2.4. In particular, recall that the adversary knows the value of $T[1]$ but not C or the local variables of each process. Therefore, it does not learn about any process's random selection of j in line 33 until the point when $T[1] \geq \ell$ for the first time.

Lemma 2.47. *For any $i \geq 1$, let op_i denote the i-th* `clear(ℓ)` *operation invoked by the ip-aware adversary if such an operation exists. Given an integer $b \geq 0$ such that the adversary invokes at least $h \cdot b$* `clear(ℓ)` *operations, the expected total number of shared memory steps that the operations in $\{op_{hb+1}, \ldots, op_{h(b+1)}\}$ take to each execute* `clear(ℓ)` *is in $O(h \log h)$.*

Lemma 2.47's proof is adapted from the Phase Three proof in Section 3.1.3 of [22]. Although we have a simpler algorithm and weaker result, our result holds in the case of the ip-aware adversary and with CAS objects, whereas Martel and Subramonian's result holds against an oblivious adversary and with read/write registers.

Proof. First, note that using Lemma 2.16, after the point t_ℓ in which $T[1] \geq \ell$ for the first time, every `clear(ℓ)` operation returns as soon as it executes line 32 or 38. It takes each process a constant number of steps to reach either of these lines, so the total number of shared memory steps from the up to h operations after t_ℓ is in $O(h)$. It suffices to prove that the total number of shared memory steps taken by the up to h operations before t_ℓ is in $O(h \log h)$.

We divide the total shared memory steps from the up to h operations before t_ℓ, which we refer to as *work*, into *blocks* of h while-loop iterations. Precisely, the first block begins when the first process invokes `clear(ℓ)`, and a block ends and the next block begins after exactly h while-loop iterations start and end since the block started, out of the up to h operations in the set being counted. We count the work of at most h `clear(ℓ)` operations, and a single iteration of the while loop takes constant work: therefore, the total amount of extra work from partial while-loop iterations (i.e., while loop iterations that begin before the block started or do not finish by the end of the block) is in $O(h)$. It follows that a single block takes $O(h)$ work from the up to h operations. The final block ends at t_ℓ, and so may include less than h completed while-loop iterations.

Consider a complete block, i.e., a block that contains exactly h complete while-loop iterations. Line 40 is executed at least h times in the block. Each of the h iterations selects an index j uniformly at random from the set $\{1, \ldots, h\}$ in line 33. Because it is a complete block, the steps in the block are taken before t_ℓ and so $T[1] < \ell$ throughout the block. Recall that the only indication of progress visible to the adversary is the value of $T[1]$ and whether a method responds in line 32 or 38. In particular, the adversary does not know the outcome of random

decisions or the contents of the array C or of $T[2\dots]$. Therefore, the adversary cannot learn any information about j until t_ℓ. Therefore, each complete while-loop iteration's selection of j is still chosen uniformly and independently at random given the adversary's knowledge. This ensures that each selection of j is independent of each prior selection of j.

From Claim 2.46, when the process executing the while loop as part of an execution of `clear(`ℓ`)` executes line 40, $T[j]$ becomes at least ℓ if both its children $T[\min\{2j, h+1\}]$ and $T[\min\{2j+1, h+1\}]$ were set to at least ℓ before the block began. Selections of nodes from incomplete while-loop iterations and from processes not in the set being counted may set additional nodes to at least ℓ, but we make the pessimistic assumption that these extra selections never set additional nodes to ℓ since setting extra nodes to ℓ can only reduce the work to set the root to ℓ. Thus, each complete block includes at least h node selections that set the node to ℓ if its children were at least ℓ when the block began. Each of the up to h selections are of an index $j \in \{1, \dots, h\}$ chosen uniformly and independently at random given the adversary's knowledge.

There are h non-leaf nodes in T (i.e., $T[1]$ through $T[h]$), and at least h node selections chosen uniformly and independently at random from complete while-loop iterations within the block: therefore, each node in $\{T[1], \dots, T[h]\}$ has a probability of at least $1 - 1/e$ of being selected within a while-loop iteration completely in the block. (Note that although each process' random selection of a node is independent, the probability of a given node being selected by any process within a given block is not independent of the probability for other nodes. In particular, the minimum number of selections implies that the probability that *no* nodes are selected is zero.)

We can thus apply Lemma 2.45 on the subtree excluding $T[h+1]$: colouring corresponds to setting a node to ℓ and a step corresponds to a complete block. Since the depth of the subtree is $\lceil \log h \rceil$, it takes at most $O(\lceil \log h \rceil / (1 - 1/e)) = O(\log h)$ complete blocks in expectation to set all nodes of $C[i][5\phi+1, \dots, \lambda]$ to ℓ. Although the final block may consist of less than h node selections and hence not be complete, we can conceptually complete the block by adding enough node selections to make a full block: these extra node selections don't have any effect on T due to Corollary 2.17, Lemma 2.16, and Claim 2.46, and a difference of h work does not affect the asymptotic upper bound. Since each block consists of $O(h)$ work, the lemma follows. □

Lemma 2.48. *Let K be a random variable counting the number of processes that the adversary schedules to execute* `clear(`ℓ`)` *for a given $\ell > 0$ in a given execution E. The total expected number of shared memory steps taken by all processes in E within* `clear(`ℓ`)` *operations is in $(h + \mathbf{E}[K]) \cdot O(\log h)$.*

Proof. From Lemma 2.47, the $(hb+1)$-th through $(h(b+1))$-th `clear(`ℓ`)` operations scheduled by the adversary take a combined expected total of $O(h \log h)$ shared memory steps when scheduled by the adversary. The total expected number of shared memory steps of the K processes is therefore at most

$$\begin{aligned}\sum_{b=0}^{\infty} \Pr[K > hb] O(h \log h) &\le O(h \log h) + O(\log h) \sum_{k=0}^{\infty} \Pr[K > k] \\ &= O(h \log h) + O(\log h) \sum_{k=0}^{\infty} \Pr[K = k] k \\ &= O(\log h) \cdot (h + \mathbf{E}[K]).\end{aligned}$$

□

2.5.2 Unlock Method. For each $a \ge 1$ we define the following (random) points in time (they are ∞ if they do not exist). The first three points are identical to the corresponding points defined in Definition 2.8, as proven in Lemma 2.9.

- t_a^{CL} is the linearization point of the a-th successful `choose&lock()` operation,
- t_a^{UL} is the linearization point of the a-th successful `unlock()` operation,

- t_a^{PL} is the pivot-lock point when the a-th successful `CAS()` operation is executed in line 19 (this line changes the value of P),
- r_a is the point when the process that executes the successful `CAS()` at t_a^{PL} executes line 18, in the same `unlock()` operation (this line reads $P.pval$ at t_a^{PL}), and
- f_a is the point when $T[1]$ (the root of the tree) is updated to $2a-1$ by a `clear`$(2a-1)$ operation.

From Lemma 2.9, we have that $t_a^{CL} < r_a < t_a^{PL} < t_a^{UL} < t_{a+1}^{CL}$. From Lemma 2.12, if $t_a^{PL} < \infty$ and $P_{t_a^{PL}}.pval \neq \perp$, then $t_a^{PL} < f_a < t_a^{UL}$. We define the following random variables:

- k_a is the number of proposals in the interval $\left(r_{a-2}, t_a^{CL}\right)$ (where $r_{a-2} = 0$ if $a-2 < 1$),
- l_a is the number of proposals in the interval (r_{a-2}, r_a),
- m_a is the number of processes that call the method `clear`$(2a-1)$ prior to point f_a,
- s_a is the total number of steps of all `clear`$(2a-1)$ operations executed by any processes prior to point f_a unless $2a-1 \bmod \lambda \geq \lambda - 4$, in which case it is zero, and
- $\varphi_a = C_{t_a^{CL}}[S_{t_a^{CL}}.i][\phi]$ if $t_a^{CL} < \infty$.

In the context of `BDCAS()`, for any given point t, the total number of invocations of the `clear()` method prior to t is bounded above by the total number of proposals prior to t.

Recall from Section 2.4 that $\mathcal{T}(t)$ denotes all the information about the past execution that is available to the adversary at point t.

Claim 2.49. *Let $a \geq 1$, and suppose that $t_a^{CL} < \infty$. Then*

$$\Pr\left[\varphi_a \neq \perp \,\middle|\, \mathcal{T}\left(t_a^{CL}\right)\right] \leq k_a/\log n.$$

Proof. From Lemma 2.9 and that each proposed value is unique, any proposal written to $C[S_{t_a^{CL}}.i][\phi]$ before point r_{a-2} is no longer present in C by point t_{a-2}^{UL}. Therefore, $\varphi_a \neq \perp$ only if some proposal writes to $C[S_{t_a^{CL}}.i][\phi]$ in the interval (r_{a-2}, t_a^{CL}). The number of proposals to side $S_{t_a^{CL}}.i$ in the interval (r_{a-2}, t_a^{CL}) is at most k_a, and they are each in the set L defined in (8).

Using Lemma 2.31 with $s^* = a-2$, each value proposed after r_a and written to $C[S_{t_a^{CL}}.i]$ has a probability of writing to $C[S_{t_a^{CL}}.i][\phi]$ equal to $\pi_\phi = 2^{-\lceil \log\log n \rceil} \leq 1/\log n$ given $\mathcal{T}\left(t_a^{CL}\right)$. (Proposals written in (r_a, t_{a-2}^{PL}) do not affect $P_{t_{a-2}^{PL}}.pval$, so the proof of Lemma 2.31 applies to them as well.) The claim then follows by using the union bound over the at most k_a proposals to side $S_{t_a^{CL}}.i$ in the interval (r_{a-2}, t_a^{CL}). □

Claim 2.50. *Let $a \geq 1$, and suppose that $t_a^{PL} < \infty$. Then*

$$\Pr\left[P_{t_a^{PL}}.pval \neq \perp \,\middle|\, \varphi_a = \perp, \mathcal{T}\left(t_a^{PL} - 1\right)\right] \leq l_a/\log n.$$

Proof. Since we assume that $\varphi_a = \perp$, $P_{t_a^{PL}}.pval \neq \perp$ only if some proposal in $\left(t_a^{CL}, r_a\right)$ writes to $C[S_{t_a^{CL}}.i][\phi]$. By Lemma 2.9, S and P do not change during the interval $\left(t_a^{CL}, t_a^{PL} - 1\right)$, and hence are not affected by the proposals to side $S_{t_a^{CL}}.i$ in $\left(t_a^{CL}, r_a\right)$. The decision of whether to invoke `clear()` in line 26 depends either on a value read from P or $S.\ell$, which does not depend on proposal locations. Thus, it also follows that $T[1]$ is not affected by proposals to side $S_{t_a^{CL}}.i$ in $\left(t_a^{CL}, r_a\right)$ by point $t_a^{PL} - 1$. Therefore, $\mathcal{T}\left(t_a^{PL} - 1\right)$ contains no information about the index β_v in $C[S_{t_a^{CL}}.i]$ of proposals v written to $C[S_{t_a^{CL}}.i]$, making $\Pr\left[\beta_v = j \,\middle|\, \mathcal{T}\left(t_a^{PL} - 1\right)\right] = \pi_\phi = 2^{-\lceil \log\log n \rceil} \leq 1/\log n$. Using the union bound over all such proposals, we have that $\Pr\left[P_{t_a^{PL}}.pval \neq \perp \,\middle|\, \varphi_a = \perp, \mathcal{T}\left(t_a^{PL} - 1\right)\right] \leq (l_a - k_a)/\log n$. □

Claim 2.51. *Let $a \geq 1$, and suppose that $t_a^{PL} < \infty$ and $P_{t_a^{PL}}.pval \neq \bot$. Then*

$$\mathbf{E}\left[s_a \,\middle|\, \mathcal{T}\left(t_a^{PL}\right)\right] \text{ in } \left(\mathbf{E}\left[m_a \,\middle|\, \mathcal{T}\left(t_a^{PL}\right)\right] + \log n\right) \cdot O(\log\log n).$$

Also, any process that invokes `clear(2a − 1)` *after f_a executes $O(1)$ steps of the method.*

Proof. This follows directly from Lemma 2.48. □

Lemma 2.52. *Let $a \geq 1$, and suppose that $t_a^{CL} < \infty$. Then*

$$\mathbf{E}\left[s_a \,\middle|\, \mathcal{T}\left(t_a^{CL}\right)\right] \in \mathbf{E}\left[l_a + m_a \,\middle|\, \mathcal{T}\left(t_a^{CL}\right)\right] \cdot O(\log\log n).$$

Proof. Suppose we fix $\mathcal{T}\left(t_a^{CL}\right)$, so that we do not need to explicitly condition our probability and expectation statements on $\mathcal{T}\left(t_a^{CL}\right)$. We have

$$\begin{aligned}
\mathbf{E}[s_a] &= \mathbf{E}\left[s_a \mid \varphi_a \neq \bot\right] \cdot \Pr[\varphi_a \neq \bot] + \mathbf{E}\left[s_a \mid \varphi_a = \bot\right] \cdot \Pr[\varphi_a = \bot], \\
\mathbf{E}\left[s_a \mid \varphi_a \neq \bot\right] &= \sum_l \mathbf{E}\left[s_a \mid \varphi_a \neq \bot, l_a = l\right] \cdot \Pr\left[l_a = l \mid \varphi_a \neq \bot\right] \\
&= \sum_l \left(\mathbf{E}\left[m_a \mid \varphi_a \neq \bot, l_a = l\right] + \log n\right) \cdot O(\log\log n) \cdot \Pr\left[l_a = l \mid \varphi \neq \bot\right] \\
&\quad \text{by Lemma 2.48} \\
&= \mathbf{E}\left[m_a \mid \varphi_a \neq \bot\right] \cdot O(\log\log n) + \log n \cdot O(\log\log n), \\
\Pr[\varphi_a \neq \bot] &\leq \min\left\{1, \frac{k_a}{\log n}\right\} \text{ by Claim 2.49}, \\
\mathbf{E}\left[s_a \mid \varphi_a = \bot\right] &= \mathbf{E}\left[s_a \,\middle|\, \varphi_a = \bot, P_{t_a^{PL}}.pval \neq \bot\right] \cdot \Pr\left[P_{t_a^{PL}}.pval \neq \bot \,\middle|\, \varphi_a = \bot\right], \\
\mathbf{E}\left[s_a \,\middle|\, \varphi_a = \bot, P_{t_a^{PL}}.pval \neq \bot\right] &\leq \mathbf{E}\left[m_a \,\middle|\, \varphi_a = \bot, P_{t_a^{PL}}.pval \neq \bot\right] \cdot O(\log\log n) + \log n \cdot O(\log\log n) \\
&\quad \text{by Lemma 2.48 as before}, \\
\Pr\left[P_{t_a^{PL}}.pval \neq \bot \,\middle|\, \varphi_a = \bot\right] &\leq \min\left\{1, \frac{\mathbf{E}\left[l_a \mid \varphi_a = \bot\right]}{\log n}\right\} \text{ by Claim 2.50}.
\end{aligned}$$

Combining the above, we get

$$\begin{aligned}
\mathbf{E}[s_a] &= \mathbf{E}\left[s_a \mid \varphi_a \neq \bot\right] \cdot \Pr[\varphi_a \neq \bot] + \mathbf{E}\left[s_a \mid \varphi_a = \bot\right] \cdot \Pr[\varphi_a = \bot] \\
&\leq \mathbf{E}\left[m_a \mid \varphi_a \neq \bot\right] \cdot O(\log\log n) \Pr[\varphi_a \neq \bot] + k_a \cdot O(\log\log n) \\
&\quad + \mathbf{E}\left[s_a \,\middle|\, \varphi_a = \bot, P_{t_a^{PL}}.pval \neq \bot\right] \cdot \Pr\left[P_{t_a^{PL}}.pval \neq \bot \,\middle|\, \varphi_a = \bot\right] \cdot \Pr[\varphi_a = \bot] \\
&\leq \mathbf{E}[m_a] \cdot O(\log\log n) + k_a \cdot O(\log\log n) \\
&\quad + \mathbf{E}\left[m_a \,\middle|\, \varphi_a = \bot, P_{t_a^{PL}}.pval \neq \bot\right] \cdot O(\log\log n) \cdot \Pr\left[P_{t_a^{PL}}.pval \neq \bot \,\middle|\, \varphi_a = \bot\right] \cdot \Pr[\varphi_a = \bot] \\
&\quad + \mathbf{E}\left[l_a \mid \varphi_a = \bot\right] \cdot O(\log\log n) \cdot \Pr[\varphi_a = \bot] \\
&\leq \mathbf{E}[m_a] \cdot O(\log\log n) + k_a \cdot O(\log\log n) + \mathbf{E}[m_a] \cdot O(\log\log n) + \mathbf{E}[l_a] \cdot O(\log\log n) \\
&= O((\mathbf{E}[m_a] + \mathbf{E}[l_a]) \cdot \log\log n).
\end{aligned}$$

□

Theorem 2.53. *Consider a random execution E of finite expected length, in which an ip-aware adversary schedules calls to the methods of an RC object R. Let K be a random variable counting the total number of invocations of* `propose()` *and* `unlock()` *operations in E. The expected total steps from* `propose()` *and* `unlock()` *operations in E is in $O(\mathbf{E}[K] \cdot \log\log n)$.*

Proof. The `propose()` method takes a single shared memory step, the number of steps in the `unlock()` method outside of steps taken within `clear()` is in $O(\log\log n)$, and `clear(`a`)` takes $O(1)$ steps after f_a. Therefore, it suffices to show that the total expected number of steps of `clear()` in E before each `clear(`a`)`'s respective point f_a is in $O(K \cdot \log\log n)$.

We can apply the weaker, unconditional version of Lemma 2.52 for each $a \in [1, \infty)$. Each invocation of `unlock()` adds at most 1 to m_a for some $a \in [1, \infty)$. Each `propose()` operation linearizes in the intervals (r_{a-2}, r_a) and (r_{a-1}, r_{a+1}) for some $a \in [1, \infty)$, so it adds 1 to l_a for at most two values of $a \in [1, \infty)$. Thus, $K \geq \sum_{a=1}^{\infty} (l_a + m_a/2)$ and

$$\sum_{a=1}^{\infty} \mathbf{E}[s_a] \in \sum_{a=1}^{\infty} \mathbf{E}[l_a + m_a] \cdot O(\log\log n) \subseteq \mathbf{E}[K] \cdot O(\log\log n).$$

Finally, we add the steps from `clear(2a-1)` operations, when $2a-1 \bmod \lambda \geq \lambda-4$. Let ℓ_E be a random variable which is equal to the value of $S.\ell$ at the end of execution E. There are at most $2\lfloor (\ell_E + 4)/\lambda \rfloor \in O(\ell_E/\log n)$ values of $2a-1$ in $\{0, \ldots, \ell_E\}$ such that $2a-1 \bmod \lambda \geq \lambda - 4$, in which cases the steps of `clear(`$2a-1$`)` are not counted in s_a. Let s'_a be a random variable counting the number of steps all `clear(`$2a-1$`)` operations executed by any processes when $2a-1 \bmod \lambda \geq \lambda-4$, and is zero otherwise. From Lemma 2.48, $\mathbf{E}[s'_a] = (h + \mathbf{E}[m_a]) \cdot O(\log h) = (\log n + \mathbf{E}[m_a]) \cdot O(\log\log n)$ when $2a-1 \bmod \lambda \geq \lambda-4$. We have

$$\sum_{a=1}^{\infty} \mathbf{E}[s'_a] \in \left(\log n \frac{\mathbf{E}[\ell_E]}{\log n} + \mathbf{E}[K]\right) \cdot O(\log\log n) = \mathbf{E}[K] \cdot O(\log\log n),$$

where the last equality follows from the fact that $K \geq \lfloor \ell_E/2 \rfloor$, using Lemma 2.5. □

We note that the `clear()` method, and hence the `unlock()` method, does not have a deterministic upper bound on its number of steps. This can be easily addressed by alternating the while-loop with for-loop steps similar to lines 22–25 for each $j \in \{5\phi + 1, \ldots, \lambda\}$. Such a change would make the faster RC object's deterministic worst-case complexity $O(\log n)$, which is the same as the original RC object [9].

Method `solo(`V'`)`

```
41 while R.read() ∉ V' do
42     Choose some v ∈ V' that has not been proposed yet
43     R.propose(v)
44     R.choose&lock()
45     u ← R.read()
46     R.unlock(u)
```

Fig. 6. The solo execution of RC methods used in Lemma 2.54

2.5.3 Solo Execution.

Lemma 2.54. *Given an execution E on a RC object R and an infinite set $V' \subseteq V$, suppose that after point t a single process p starts to run solo, according to the following procedure. First, p completes all its pending steps until it has no more pending step. Then, p executes the* `solo(`V'`)` *method in Figure 6 until it responds at point t'. Then,*

$$\mathbf{E}\left[t' \mid E_t\right] \le t + O(\log^2 n \log\log n).$$

Proof. We first find an upper bound on the expected number of steps to finish p's pending method call at point t. In the case of the `propose()`, `choose&lock()`, and `read()` methods, the number of steps is deterministic and in $O(\log\log n)$. The number of steps of the `unlock()` method outside `clear()` is also deterministic and in $O(\log\log n)$. In the case of the `clear()` method, from Lemma 2.47 the expected number of steps of `clear()` in a solo execution is in $O(h \log h) = O(\log n \log\log n)$. Thus, the expected number of steps until op completes its pending method call at point t is at most $O(\log n \log\log n)$. Using the same analysis, it also follows that the expected number of steps for each while-loop iteration is at most $O(\log n \log\log n)$. Therefore, it suffices to show that the `solo()` method takes at most $O(\log n)$ expected while-loop iterations to prove the lemma.

Consider the first while-loop iteration of the `solo()` method. If R is unlocked when p invokes `solo()`, then, by the sequential specification, the `choose&lock()` is successful and `unlock(`u`)` successfully unlocks R. If R is locked when p invokes `solo()`, then by the sequential specification the `choose&lock()` is unsuccessful but `unlock(`u`)` still successfully unlocks R. Thus, by the end of the first while-loop iteration, R is unlocked. Moreover, every while-loop iteration after the first has successful `choose&lock()` and `unlock(`u`)` operations. Since the while loop terminates once the interpreted agreement-value is in V', and p is running solo, it follows that on the last while-loop iteration, the `choose&lock()` chooses some value in V' as the new interpreted agreement-value.

From line 26 and that $S.\ell$ increments with each successful `choose&lock()` and `unlock()` operation, it follows that after at most $\lambda/2 \le \log n$ while-loop iterations, the `clear()` method is executed solo by p for both sides of C. Using Claim 2.21, C does not contain any values proposed before t after p finishes the two while-loop iterations that execute `clear()` for both sides of C; call the point of the end of the second such while-loop iteration t_c. After t_c, C contains only $\perp$ and values from V', so `solo()` terminates as soon as the interpreted agreement-value is not $\perp$.

We now show that after an expected $O(1)$ while-loop iterations after t_c, the interpreted agreement-value is a value from V'. First, denote by t_ϕ the first point when a `propose()` operation executed in line 43 chooses a value $\beta \le \phi$. Each `propose()` operation writes to $C[\alpha][\beta]$ with $\beta \le \phi$ with probability $1 - \phi_\pi = 1 - 2^{-\lceil \log\log n \rceil} \ge 1 - 1/\log n$. Therefore, t_ϕ occurs after an expected $O(1/(1 - 1/\log n)) = O(1)$ while-loop iterations after t_c.

With probability $1/2$, the proposal at t_ϕ chooses $\alpha = S_{t_\phi}.i$. Since p is running solo, this proposal is in C throughout the `choose&lock()` operation immediately after t_ϕ. We use a similar argument to the proof of Lemma 2.35, with one simplification: we already know that there is a value in $C[S_{t_\phi}.i]$, which is the first step of that proof. Thus, the new interpreted agreement-value once the (successful) `choose&lock()` linearizes is not $\perp$. With the remaining $1/2$ probability, the proposal at t_ϕ chooses $\alpha = 1 - S_{t_\phi}.i$. Since p runs solo and values in $C[\alpha]$ are cleared only by `unlock()` operations invoked when $S.i = \alpha$, it follows that the proposal stays in $C[\alpha]$ for the remainder of the while-loop iteration and into the next iteration. When p's successful `unlock()` operation at the end of t_ϕ's while-loop iterations linearizes, it flips the value of $S.i$ in line 28. In the next iteration, with $S.i = \alpha$, the `choose&lock()` in line 44 chooses a non-$\perp$ value, again using a similar argument to the proof of Lemma 2.35. ◻

3 BIPARTITE DCAS

A BDCAS object stores an array $B[0..m-1]$ of values from $D \cup \{\perp\}$, where D is some set and $\perp \notin D$. The initial value of each array entry is $\perp$. The set of addresses $[m] = \{0, 1, \ldots, m-1\}$ is partitioned into two sets, M_0 and M_1. The object supports the operation `BDCAS(`$\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$`)`, where $a_i \in M_i$, $old_i \in D \cup \{\perp\}$,

and $new_i \in D$, for $i \in \{0,1\}$. If $B[a_i] = old_i$ for each $i \in \{0,1\}$, then the `BDCAS()` operation changes the value of $B[a_i]$ to new_i, for both $i \in \{0,1\}$. Otherwise, the object's value remains unaffected by the `BDCAS()` operation. The operation does not return anything.[1] The object also supports operation `read`(a), for $a \in [m]$, which returns the value of $B[a]$.

Our BDCAS implementation has the following *irreflexivity requirement*.

Definition 3.1 (Irreflexivity Requirement)**.** For any $a \in [m]$, an execution on a BDCAS object induces a binary relation $\lhd_a$ on $D \cup \{\bot\}$, where $x \lhd_a y$ if and only if there is a `BDCAS()` call using the argument triple $\langle a, x, y\rangle$. Let $\prec_a$ denote the transitive closure of $\lhd_a$. We require that for each $a \in [m]$, the relation $\lhd_a$ is acyclic, or equivalently, the relation $\prec_a$ is irreflexive, i.e., there is no $x \in D \cup \{\bot\}$ with $x \prec_a x$.

The irreflexivity requirement can easily be achieved by adding sequence numbers, provided that no `BDCAS()` calls with arguments of the form $\langle a, x, x\rangle$ are allowed.

3.1 Implementation

In Figure 7, we present a randomized strongly linearizable implementation of a BDCAS object B.

Our DCAS implementations store the values of their array entries, and also information for each `BDCAS()` operation, to `Task` objects. A `Task` object contains the fields: $stat \in \{\bot, \mathsf{True}, \mathsf{False}\}$, $add_i \in M_i$, $old_i \in D \cup \{\bot\}$, and $new_i \in D$, for $i \in \{0,1\}$. New `Task` objects are created by `new Task`$(add_0, old_0, new_0, add_1, new_1, old_1)$ operation, which creates a task initialized with the listed values and $stat = \bot$, and returns a reference to that object. All objects created like that are distinct. We will use '*task*' and '*task reference*' as shorts for '`Task` object' and 'reference to a `Task` object', respectively. Moreover, we will often drop the distinction between a task reference and the task itself, when there is no danger of confusion or when the distinction is not important.

The implementation uses an array $A[0..m-1]$, where each entry stores a task reference. In particular, if $A[a] = \tau$ and $a \in M_i$, then $\tau.add_i = a$. Initially, for each $i \in \{0,1\}$ and $a \in M_i$, $A[a]$ stores a reference to the *initial task* λ_a, where $\lambda_a.stat = \mathsf{True}$, $\lambda_a.add_i = a$, and $\lambda_a.old_i = \lambda_a.new_i = \bot$; the values of the remaining fields of λ_a can be arbitrary.

In addition to A, we use an array L that consists, for each $a \in M_0$, of an RC object $L[a]$. The proposed values at each RC object are task references.

For each $a \in [m]$, if $A[a] = \tau$ and $a \in M_i$, then we define the *interpreted value* of $B[a]$ to be $\tau.new_i$ if $\tau.stat = \mathsf{True}$, and $\tau.old_i$ otherwise. Note that if $A[a_0] = A[a_1] = \tau$, where $a_i \in M_i$ for $i \in \{0,1\}$, then the status field $\tau.stat$ will allow us to *simultaneously* change the interpreted value of $B[a_0]$ and $B[a_1]$, from $\tau.old_0$ to $\tau.new_0$ and from $\tau.old_1$ to $\tau.new_1$, respectively, by changing $\tau.stat$'s value from $\bot$ to True.

For each $i \in \{0,1\}$, we call the portion of array A with entries $A[a]$, $a \in M_i$, the *i-side* of A. We do the same for B.

3.2 High Level Idea

We describe first a lock-free *deterministic* BDCAS implementation, and then explain how to achieve low expected amortized complexity by using randomized RC objects.

Recall that each array entry $A[a]$, where $a \in M_i$ and $i \in \{0,1\}$, stores a task τ with $\tau.add_i = a$, and the status field $\tau.stat$ indicates whether the interpreted value of $B[a]$ is $\tau.new_i$ or $\tau.old_i$.

To perform operation `read`(a), the calling process simply reads the task τ stored in $A[a]$, then the status field of τ, and returns $\tau.new_i$ if $\tau.stat = \mathsf{True}$ or $\tau.old_i$ otherwise.

Suppose now that process p wants to perform a `BDCAS`$(\langle a_0, v_0, v'_0\rangle, \langle a_1, v_1, v'_1\rangle)$ operation. For that, p checks if the interpreted values of $B[a_0]$ and $B[a_1]$ match the expected values v_0 and v_1, respectively, by performing

[1] It is not difficult to add return values indicating success or failure, but this would further complicate our code. In our general DCAS algorithm, `DCAS()` operations do provide return values, and the algorithm does not rely on return values of `BDCAS()` operations.

Shared Data:

- $A[j]$, for $j \in [m]$, is a CAS object storing a task reference, and initially stores a reference to λ_j.
- $L[j]$, for $j \in M_0$, is an RC object for task references

Method BDCAS($\langle a_0, old_0, new_0 \rangle$, $\langle a_1, old_1, new_1 \rangle$)

```
47 while True do
48     γ0 ← finish(a0)
49     if read(a0)≠ old0 or read(a1)≠ old1 then return
50     γ ← new Task(a0, old0, new0, a1, old1, new1)
51     L[a0].propose(γ)
52     L[a0].choose&lock()
53     γ′ ← L[a0].read()
54     if A[a0] = γ0 then
55         if γ′ ≠ ⊥ and γ′.old0 = old0 then
56             A[a0].CAS(γ0, γ′)
57         finish(a0)
58         L[a0].unlock(γ′)
```

Method finish(a_0)

```
59 γ0 ← A[a0]
60 a1 ← γ0.add1
61 γ1 ← A[a1]
62 γ1.stat.CAS(⊥, True)
63 if γ0.old1 = γ1.new1 then
64     A[a1].CAS(γ1, γ0)
65     γ1 ← A[a1]
66     γ1.stat.CAS(⊥, True)
67 γ0.stat.CAS(⊥, False)
68 return γ0
```

Method read(a)

```
69 let i ∈ {0, 1} be such that a ∈ Mi
70 γ ← A[a]
71 if i = 1 then γ.stat.CAS(⊥, True)                // Help the task finish
72 if γ.stat = True then return γ.newi
73 return γ.oldi
```

Fig. 7. BDCAS implementation, taken almost verbatim from [9] with minor modifications

read() operations as described above. If at least one of the values does not match, then the BDCAS() can return immediately (it fails). Hence, assume that both values match.

Then p creates a task τ whose field values correspond to p's BDCAS() arguments (i.e., $\tau.add_i = a_i$, $\tau.old_i = v_i$, and $\tau.new_i = v_i'$ for each $i \in \{0, 1\}$) and $\tau.stat = \perp$. The goal of p is now to place its task τ in the array entries $A[a_0]$ and $A[a_1]$. This must happen without at first changing the interpreted values of $B[a_0]$ or $B[a_1]$. Once τ is put into both array entries, p can change $\tau.stat$ from $\perp$ to True.

We say a task is *finalized*, if its status is either True or False (i.e., it is not ⊥). Our algorithm ensures that once a task is finalized, its status does not change anymore. To place a task into $A[a_i]$, $i \in \{0, 1\}$, process p reads the task τ' stored in that array entry *before* it performs its initial check whether the BDCAS() call's expected values v_0 and v_1 match. It then performs a helping mechanism (which we describe later) that ensures task τ' is finalized. Thus, the only way the interpreted value of $B[a_i]$ can change afterwards is if task τ' gets replaced by a different task in $A[a_i]$. Now, to place its task τ into $A[a_i]$, process p can simply perform an $A[a_i]$.CAS(τ', τ) operation. If that CAS() succeeds, then τ has been put into $A[a_i]$ without changing the interpreted value of $B[a_i]$ (which remains $v_i = \tau.old_i$).

If p manages to place its task τ into both array entries, $A[a_0]$ and $A[a_1]$, it can simply change $\tau.stat$ to True, and both interpreted values of $B[a_i]$, $i \in \{0, 1\}$, change simultaneously from $v_i = \tau.old_i$ to $v_i' = \tau.new_i$. Hence, p's BDCAS() linearizes (and is successful).

But p may not manage to put its tasks in one of the array entries, if the corresponding CAS() operation on $A[a_i]$ fails, because some other task τ^* has already been put in there. The standard lock-free approach would now be to help τ^* make progress. This can lead to a long chain of helping, and requires care that no cyclic helping is encountered.

In the bipartite case, however, things are easier: Each process first tries to put its task into the 0-side array entry of A, before it tries to put it also into the 1-side entry. This way, if any task is successfully put into a 1-side array entry, the BDCAS() operation that created the task is bound to succeed. Thus, if process p fails to put its task into $A[a_1]$, because that array entry was taken by some other task τ^*, then some other BDCAS() operation (which created τ^*), with the same 1-side entry $A[a_1]$, will succeed once τ^* is finalized, and thus the interpreted value of $B[a_1]$ will change from v_1 to a different value. As a result, p's BDCAS() operation can terminate, and fail, after helping to finalize τ^* (which means just changing $\tau^*.stat$ to True). This way long chains of helping and cyclic helping are avoided.

A process p may still not make progress, if it fails to put its task into the 0-side entry $A[a_0]$. If that happens, it has to help the task τ^* that caused p's attempt to fail, but again, since τ^* has already been put into the 0-side, only the 1-side of τ^* remains to deal with. However, putting τ^* into the 1-side may fail, in which case none of the interpreted values of B at the addresses a_0 and a_1 that p is interested in may have changed. In that case, p's only option is to start over. This may lead to high step complexity, in the case of high contention on the 0-side entry $A[a_0]$ (high contention on the 1-side entry $A[a_1]$ does not cause high step complexity).

To deal with that, we employ a mechanism to choose a task τ^* at random among concurrent tasks that use the same 0-side address. To this end, we use for each 0-side address $a_0 \in M_0$ an RC object $L[a_0]$. Consider the processes that are concurrently performing BDCAS() operations with the same 0-side address a_0. Instead of directly trying to put their tasks into $A[a_0]$, they propose them on $L[a_0]$, and choose a random task τ^* among all the proposed ones. Then each process helps τ^*, by first putting it into $A[a_0]$, and then trying to put it into $A[\tau^*.add_1]$. As discussed earlier, if τ^* is successfully put into its desired 1-side array entry, the interpreted value of $B[a_0]$ changes eventually. Hence, all processes helping τ^* can finish their BDCAS() operation (the one that created task τ^* succeeds, and all the others fail). But if τ^* cannot be successfully put into $A[\tau^*.add_1]$, then all processes helping τ^* have to start over.

To give an intuition why that achieves low expected amortized complexity, assume there are k processes that concurrently perform BDCAS() operations with the same 0-side address a_0. Let $a_{1,1}, \ldots, a_{1,k}$ be the 1-side addresses of these BDCAS() operations, and suppose for simplicity they are all distinct. All k processes will help the same random task τ^*, by trying to put the task into the 1-side address $a_{1,j} = \tau^*.add_1$. Suppose that ℓ of the 1-side array entries, say $A[a_{1,1}], \ldots, A[a_{1,\ell}]$, change before the first attempt is made to put τ^* into $A[a_{1,j}]$. Since j is chosen at random, the attempt to put τ^* into $A[a_{1,j}]$ fails with probability ℓ/k (assuming, for the ease of this discussion, a uniform distribution for j). Out of the k processes, just ℓ make progrees (by failing)

if $j \leq \ell$, and all k make progress otherwise. Thus the expected number of processes that make progress is $\ell \cdot \ell/k + k \cdot (k-\ell)/k = k - \ell(k-\ell)/k \geq 3k/4$.

3.3 Detailed Algorithm Description

3.3.1 Read Method. In method `read`(a), the calling process p determines the interpreted value of $B[a]$ in a straightforward manner. First, p reads the current task γ from $A[a]$, in line 70. If a is a 1-side address, then the `BDCAS()` operation that created γ is bound to succeed, so p changes $\gamma.stat$ to True if it is still $\perp$, in line 71. (This is not necessary for linearizability, but for the desired randomized step complexity: A task that is put into its 1-side array entry may prevent other processes from making progress, until these processes observe that the interpreted value of that entry has changed.) After that, in lines 72 and 73, p simply determines and returns the interpreted value of $B[a]$ based on the current status of γ. We will show that if the status of γ changes at some point, then $A[a]$ contains γ at that point. Hence, the `read()` operation can linearize either at the first point after its invocation when $\gamma.stat \neq \perp$, or when p reads $\perp$ from $\gamma.stat$ in line 72.

3.3.2 Finish Method. The helper method `finish`(a_0), where $a_0 \in M_0$, "finishes" the task stored in array entry $A[a_0]$. The method guarantees that, if process p calls `finish`(a_0) when task γ_0 is stored in $A[a_0]$ (more precisely, p reads γ_0 from $A[a_0]$ in line 59), then by the time the call returns, either γ_0 has been put into $A[\gamma_0.add_1]$ and $\gamma_0.stat$ = True, or γ_0 was not put into $A[\gamma_0.add_1]$ because in the meantime the interpreted value of $B[\gamma_0.add_1]$ changed (from value $\gamma_0.old_1$ to a different value) and $\gamma_0.stat$ = False. The `finish()` call returns the task γ_0, in line 68.

In lines 60–62, process p reads γ_1 from $A[a_1]$, where $a_1 = \gamma_0.add_1$, and tries to change $\gamma_1.stat$ to True using operation $\gamma_1.stat$.CAS($\perp$, True). It is then guaranteed that $\gamma_1.stat$ = True after p's `CAS()` operation, and the interpreted value of $B[a_1]$ changed to $\gamma_1.new_1$ at the point when $\gamma_1.stat$ became True.

Next, p checks in line 63, if $\gamma_0.old_1 = \gamma_1.new_1$. If this is not the case, then either some other process has already put γ_0 into $A[a_1]$ and changed its status, or the `BDCAS()` operation that created γ_0 is bound to fail. The latter is the case because the interpreted value of $B[a_1]$ changed during the `BDCAS()` call that created γ_0 from its expected value, $\gamma_0.old_1$, to a different value, which by the irreflexivity requirement will never again be $\gamma_0.old_1$. Hence, in line 67 process p executes $\gamma_0.stat$.CAS($\perp$, False), which changes the status of γ_0 to False in that case (but fails if γ_0 was put into $A[a_1]$).

Now assume that $\gamma_0.old_1 = \gamma_1.new_1$, and p evaluates the if-condition in line 63 to True. Then p tries to put γ_0 in the desired 1-side entry $A[a_1]$, using operation $A[a_1]$.CAS(γ_1, γ_0) in line 64. If the `CAS()` is successful, it does not change the interpreted value of $B[a_1]$, because $\gamma_0.stat = \perp$ and $\gamma_1.stat$ = True at this point, and so the interpreted value remains $\gamma_1.new_1 = \gamma_0.old_1$. To make sure the status of γ_0 changes to True if γ_0 is successfully put into $A[a_1]$, p reads task γ_1' from $A[a_1]$ and performs $\gamma_1'.stat$.CAS($\perp$, True), in lines 65–66. After that, in line 67, p also executes $\gamma_0.stat$.CAS($\perp$, False). This `CAS()` can only succeed if neither p nor any other process managed to put γ_0 into $A[a_1]$.

3.3.3 BDCAS Method. If process p calls method `BDCAS`($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$), then p repeats the while-loop in lines 47–58, until it observes in line 49 that one of the interpreted values of $B[a_0]$ and $B[a_1]$ does not match the expected value old_0 or old_1, respectively.

First, in line 48, p finishes the task γ_0 currently stored in $A[a_0]$. Then, it checks if the interpreted values of $B[a_0]$ and $B[a_1]$ match the values old_0 and old_1, respectively, and if not the `BDCAS()` operation returns. Since we don't require a `BDCAS()` operation to return a value that indicates whether it was successful, it is always correct to return after the above check fails. If the check failed because some of the interpreted values changed from their expected values at point t, and t is after the invocation point of `BDCAS()`, then the linearization point of the `BDCAS()` can be immediately after t in case the `BDCAS()` fails, or at point t if it succeeds.

In line 50, p creates a new task γ whose status is $\bot$, and whose other fields match the arguments of the BDCAS() operation. Then, in lines 51 and 52, p proposes task γ on the RC object $L[a_0]$, and calls $L[a_0]$.choose&lock() to allow that a random task is chosen on $L[a_0]$ (but recall that $L[a_0]$ may also end up with agreement-value $\bot$). Next, in line 53, p reads the agreement-value γ' of $L[a_0]$.

In line 54, p checks if the task stored in $A[a_0]$ is still the same task γ_0 that p finished earlier. If not, p just starts another iteration of the while-loop. Now suppose that $A[a_0]$ is still γ_0 when p executes line 54. In line 55, p checks to make sure that $\gamma' \neq \bot$ (i.e., γ' is a task) and $\gamma'.old_0 = old_0$, before trying to put γ' into the 0-side using $A[a_0]$.CAS(γ_0, γ'), in line 56. The check $\gamma'.old_0 = old_0$ is required for the following reason: The task γ' that p read from $L[a_0]$ may be completely unrelated to p's expected value old_0, because γ' may have been proposed on $L[a_0]$ a long time before it was chosen on $L[a_0]$. But if $\gamma'.old_0 = old_0$ and γ_0 is still stored in $A[a_0]$ (and thus the CAS() in line 56 may succeed), then the interpreted value of $B[a_0]$ is equal to $\gamma'.old_0$. Hence, if p's CAS() in line 56 succeeds, it does not change that interpreted value.

After that, p finishes the task now stored in $A[a_0]$ by calling finish(a_0) in line 57, and then it calls $L[a_0]$.unlock(γ') in line 58, unlocking $L[a_0]$ if it hasn't already been unlocked by some other process. The finish(a_0) call in line 57 ensures that every task γ' that is successfully put into $A[a_0]$ is finished before any process calls $L[a_0]$.unlock(γ'), i.e., either γ' is put into $A[a_1]$ and its status is True, or it is not put into $A[a_1]$ and its status is False.

3.4 BDCAS Correctness Statements

In this section we repeat several claims (without proofs) from Section 7 of [10] used in Section 3.5 with some minor revisions. Each of the statements uses the corresponding numbering, i.e., Claim 3.1 corresponds to Claim 7.1 of [10].

Claim 3.1. *Let $i \in \{0, 1\}$ and $a \in M_i$ and let t be a point at which $A[a] = \tau$.*

- *(a) τ is a task reference and $\tau.add_i = a$.*
- *(b) If $\tau \neq \lambda_a$, then τ is returned before t from a* newTask() *call in line 50, and prior to t $A[a]$ changes to τ with a successful* CAS() *operation executed in line 56 if $i = 0$, and in line 64 if $i = 1$.*

Claim 3.3. *Let $a \in M_1$ and let $\tau \neq \lambda_a$ be a task reference. If $A[a] = \tau$ at point t_1, then there is a point $t_0 \leq t_1$ at which $A[\tau.add_0] = \tau$.*

Claim 3.5. *For any task τ, $\tau.stat \in \{\bot, \text{False}, \text{True}\}$, and if $\tau.stat \neq \bot$ at point t, then $\tau.stat$ does not change throughout $[t, \infty)$.*

Claim 3.6. *Let $a \in [m]$ and suppose at point t the value of $A[a]$ changes from τ to $\tau' \neq \tau$. Then there is a value $v \in \{\text{True}, \text{False}\}$ such that $\tau.stat = v$ throughout $[t, \infty)$.*

Lemma 3.7. *Let τ be a task and suppose at some point t the value of $\tau.stat$ changes to $v \in \{\text{True}, \text{False}\}$. If $v = \text{False}$, then $A_t[\tau.add_0] = \tau$, and if $v = \text{True}$ then $A_t[\tau.add_0] = A_t[\tau.add_1] = \tau$.*

Claim 3.8. *Let $a \in M_1$, and τ, τ' two distinct task references, so that at some point t the value of $A[a]$ changes from τ to τ'. Then $\tau.new_1 = \tau'.old_1 \prec_a \tau'.new_1$.*

Claim 3.9. *Let $a \in M_1$, τ_1, τ_2 two task references and $t_1 < t_2$ two points in time, such that $A_{t_j}[a] = \tau_j$ for each $j \in \{1, 2\}$.*

- *(a) If $\tau_1 \neq \tau_2$ then $\tau_1.new_1 \prec_a \tau_2.new_1$.*
- *(b) If $\tau_1 = \tau_2$, then $A_t[a] = \tau_1$ for all $t \in [t_1, t_2]$.*

Lemma 3.11. *Let $a \in M_1$. If $A[a] = \tau$ at any point in the execution, then $\tau.stat \neq$ False throughout the execution.*

Claim 3.13. *Let $a \in M_1$, and suppose at some point the interpreted value of $B[a]$ changes from b to $b' \neq b$. Then $b \prec_a b'$.*

Lemma 3.15. *Let $a \in [m]$. Each complete* `read(a)` *operation op returns the interpreted value of $B[a]$ at $lin(op)$.*

Lemma 3.16. *Let $a \in M_0$. If $A[a] = \tau$ at point t and $A[a] \neq \tau$ at point $t' > t$, then $A[a] \neq \tau$ throughout $[t', \infty)$.*

Claim 3.17. *Suppose a* `finish()` *call returns task τ at point t. Then there is a value $v \in \{\text{False}, \text{True}\}$ such that $\tau.stat = v$ throughout $[t, \infty)$.*

Lemma 3.18. *Let $a \in M_0$. Suppose at point t the value of $A[a]$ changes from τ to $\tau' \neq \tau$. Then at that point*

(a) $\tau'.stat = \bot$, and

(b) either $\tau.stat = \text{True}$ and $\tau'.old_0 = \tau.new_0$, or $\tau.stat = \text{False}$ and $\tau'.old_0 = \tau.old_0$.

In particular, the interpreted value of $B[a]$ does not change at point t.

Corollary 3.19. *Fix $a \in M_0$, and let $b_1, \ldots, b_k$ be the interpreted values of $B[a]$ obtained in this order (i.e., $b_{i+1} \neq b_i$ for $i \in \{1, \ldots, k-1\}$). Then $b_i \prec_a b_j$ for all $1 \leq i < j < k$.*

Claim 3.20. *Let τ be a task, and suppose at point t the value of $\tau.stat$ changes to True. Then at that point the interpreted value of $B[\tau.add_i]$, $i \in \{0, 1\}$, changes from $\tau.old_i$ to $\tau.new_i$, where $\tau.old_i \prec_{\tau.add_i} \tau.new_i$.*

Claim 3.23. *If op is a successful* `BDCAS`$(\langle a_0, v_0, v_0' \rangle, \langle a_1, v_1, v_1' \rangle)$ *operation, then $t_{stat}(op) = lin(op)$.*

Lemma 3.24. *For any address $a \in [m]$ and any point t during the execution, the following is true:*

(a) If at point t the interpreted value of $B[a]$ changes from v to $v' \neq v$, then there is a successful `BDCAS()` *operation op such that $lin(op) = t$, and one of the argument triples used in the invocation of op is $\langle a, v, v' \rangle$.*

(b) Let op be a successful `BDCAS`$(\langle a_0, v_0, v_0' \rangle, \langle a_1, v_1, v_1' \rangle)$ *operation. Then for each $i \in \{0, 1\}$ at point $lin(op)$ the interpreted value of $B[a_i]$ changes from v_i to v_i', and $lin(op)$ is between the invocation and response of op.*

3.5 Step Complexity Preliminaries

We next establish an upper bound on the expected amortized step complexity of `BDCAS()` operations. We consider an execution on a BDCAS object B scheduled by the ip-aware adversary described in Definition 2.24.

Theorem 3.25. *Let E be a random execution of finite expected length, in which an ip-aware adversary schedules calls to the methods of a BDCAS object B. For any $x \in M_0$, if N_x denotes the number of* `BDCAS()` *operations involving address x that are invoked in E, then the expected total number of steps of those operations is in $O(\log \log n) \cdot \mathbf{E}[N_x]$.*

The remainder of this section is devoted to the proof of this theorem.

Since we consider a single BDCAS object B, we omit prefix B when we refer to methods or variables of B, e.g., we write $L[a]$ instead of $L[a]$. Also, for simplicity we do not distinguish between a task reference and the task object itself, and use '*task*' to refer to both.

3.5.1 Doomed and Successful Tasks. We recall the definition of *interpreted values*, and introduce the related notion of *imminent interpreted values*.

Definition 3.26 (Interpreted Values and Imminent Interpreted Values)**.** Let $i \in \{0, 1\}$ and $a \in M_i$.

- The *interpreted value* of $B[a]$, denoted $B[a].val$, is equal to $A[a].new_i$ if $A[a].stat = \text{True}$, and $A[a].old_i$ otherwise.
- The *imminent interpreted value* of $B[a]$, denoted $B[a].val'$, is equal to the interpreted value that $B[a]$ would have if we set to True the status of all tasks $A[a_1]$, $a_1 \in M_1$. Formally, $B[a].val' = A[a].new_i$ if $A[a].stat = \text{True}$ or $A[A[a].add_1] = A[a]$, and $B[a].val' = A[a].old_i$ otherwise.

Next we define the intuitive notions of doomed and successful tasks.

Definition 3.27 (Doomed Tasks and Successful Tasks)**.** A process *creates* task τ when a `newTask()` operation in line 50 returns τ. Suppose a process creates task τ at point t_τ. For any $t \geq t_\tau$,

- τ is *successful* at point t if $A_{t'}[\tau.add_1] = \tau$ for some $t' \leq t$.
- τ is *doomed* at point t if it is not successful at t, and $B_t[\tau.add_i].val' \neq \tau.old_i$ for some $i \in \{0, 1\}$.

Each initial task λ_a, $a \in [m]$, is successful and not doomed at any point of the execution. For $t \geq 0$, we denote by D_t and S_t the set of tasks that are respectively doomed or successful at t.

It is immediate from the definition that D_t and S_t are disjoint. We show in Lemma 3.42 that they are also monotone increasing.

3.5.2 Strengthening the Adversary: BDCAS Traces. Similarly to the analysis of the RC object, we assume that the adversary has some extra information on the past execution, and formally describe this information in terms of the *trace* for BDCAS object B.

Definition 3.28 (BDCAS Trace)**.** Let E be an execution on BDCAS object B, and let $x \in M_0$. Trace $\mathcal{T}_{B,x}(E)$ contains the following information for each step $t \in \{1, 2, \ldots\}$ of E:

(i) The process p that executes step t.
(ii) The number of the line in the program code containing the shared memory operation executed at step t.
(iii) If step t was the first step of an implemented method call of B (i.e., `BDCAS()`, `finish()`, or `read()`), or the first step of an implemented method call of $L[j]$ for $j \in M_0$ (i.e., `propose()`, `choose&lock()`, `unlock()` , `read()`, or `clear()`), the name *and arguments* (if any) of that call.
(iv) Whether or not step t was the final step of an implemented method call.
(v) The following values:
  - $A_t[a]$, for each $a \in [m] \setminus \{x\}$;
  - $L_t[a]$, for each $a \in M_0 \setminus \{x\}$ (i.e., the complete state of object $L[a]$);
  - Whether or not $A_t[x] = A_{t-1}[x]$ (but not the actual value of $A[x]$);
  - $L[x].S.\ell_t$, $L[x].P_t$, and $L[x].T[1]$ (but not $L_{t-1}[x].val$).
  - If step t was the linearization point of an $L[x]$.`propose`(v) call, the value of α_v.

Lemma 3.29. *From trace $\mathcal{T}_{B,x}(E)$, one can infer the interpreted (and imminent interpreted) value of $B[a]$ for every $a \in [n]$, at all points $t \leq |E|$, the set S_t of successful tasks, and the fields of all doomed tasks (the tasks in D_t).*

Proof. If $a \neq x$, then from (v) and Definition 3.26 the interpreted value of $B[a]$ is directly available from $A[a]$ which is in the trace.

Next, consider $B[a].val'$ (the imminent interpreted value of $B[a]$) for $a \neq x$. Since $x \notin M_1$, the trace includes the tasks stored in $A[a]$ for each $a \in M_1$. If $a \in M_1$, then $A[a].add_1 = a$ at all points, so by Definition 3.26 $B[a].val' = A[a].new_1$ at all points, which is in the trace. If $a \in M_0$, since the trace includes both $A[a]$ (including $A[a].add_1$) and $A[A[a].add_1]$, $B[a].val'$ can be derived from the trace.

Now consider $B[x].val$. From Lemma 3.18, $B[x].val$ does not change when the task stored in $A[x]$ changes; it therefore follows from Claim 3.5 and Definition 3.26 that $B[x].val$ can only change when $A[x].stat$ changes from $\perp$ to True. Suppose that at point t, $B[x].val$ changes, and hence that $A[x].stat$ changes from $\perp$ to True. By Lemma 3.7, there exists an $a_1 \in M_1$ such that $a_1 = A_t[x].add_1$ and $A_t[x] = A_t[a_1]$. Since the trace includes $A[a_1]$ and in particular includes the fact that $A[a_1].stat$ changes from $\perp$ to True at point t and that $A_t[a_1].add_0 = x$, the value of $B_t[x].val = A_t[x].new_0 = A_t[a_1].new_0$ can be derived from the trace. Since $B[x].val$ begins as $\perp$ and is derivable from the trace every time it changes, it follows that it is derivable from the trace at all points.

We next prove the case for $B[x].val'$. Suppose that $B[x].val'$ changes at point t. In addition, suppose for the purpose of proving a contradiction that $A[x]$ changes at point t. By Lemma 3.18, $A_t[x].stat = \perp$, and by

Lemma 3.16 and Claim 3.3, $A_t[A_t[x].add_1] \neq A_t[x]$. Therefore, from Definition 3.26, $B_t[x].val' = A_t[x].old_0 = B_t[x].val$. Using Lemma 3.18 again, there are two possibilities. First, it could be that $A_{t-1}[x].stat = \mathtt{True}$ and $A_{t-1}[x].new_0 = A_t[x].old_0$; by Definition 3.26, $B_{t-1}[x].val' = A_{t-1}[x].new_0 = A_t[x].old_0$, which contradicts that $B[x].val'$ changes at point t'. Second, it could be that $A_{t-1}[x].stat = \mathtt{False}$ and $A_{t-1}[x].old_0 = A_t[x].old_0$; by Definition 3.26 and Lemma 3.11 again, $A_{t-1}[A_{t-1}[x].add_1] \neq A_{t-1}[x]$ and so $B_{t-1}[x].val' = A_{t-1}[x].old_0$, also contradicting that $B[x].val'$ changes at point t. It thus follows that $A[x]$ does not change at point t, so either $A[x].stat$ changes from $\bot$ to $\mathtt{True}$ at point t or $A[A_t[x].add_1]$ changes to $A_t[x]$ at point t. In the first case, $B[x].val$ changes at point t and is equal to $B_t[x].val'$, so by the previous paragraph the new value of $B[x].val'$ is visible in the trace. In the second case, the analysis is similar to the previous paragraph; specifically, the trace at point t includes the value of $A_t[A_t[x].add_1] = A_t[x]$, and in particular the values $A_t[x].add_0 = x$ and $A_t[x].new_0 = B_t[x].val'$.

Because task creation in line 50 creates a task whose fields only depend on the calling operation's arguments, and because the trace includes instruction pointers, the trace includes the fields of each task at the point they are created. From Definition 3.27, tasks τ are successful starting at the point t when $A_t[\tau.add_1] = \tau$, which is directly in the trace. Thus, the set of successful tasks is derivable from the trace. Doomed tasks are defined based on successful tasks and imminent interpreted values, all of which are derivable from the trace. However, their references may not be stored anywhere in A, so only their fields are guaranteed to be derivable from the trace. ◻

Unlike the trace for RC object $L[x]$ (Definition 2.25), the BDCAS trace does not explicitly describe the agreement value $L[x].val$. However, the value can sometimes be inferred indirectly, e.g., when a process executes line 58 the trace contains the argument of the `unlock()` call, by point (iii).

3.5.3 Extending E to Executions E' and E''. We define two additional executions E' and E'' such that E' is an extension of the execution E described in the statement of Theorem 3.25 and has finite expected length, while E'' is an infinite extension of E'.

We extend E into execution E' by scheduling all processes that have a pending `BDCAS()` operation at the end of E to run solo (one after the other) until their operations are completed. E' has finite expected length because E has finite expected length, and because Lemma 3.59 implies that just a finite expected number of additional steps are needed to obtain E'. Hence, no additional `BDCAS()` operations are invoked in E', and no `BDCAS()` operation is pending at the end of E'.

We further extend E' into an infinite execution E'', such that E'' contains an infinite sequence of `BDCAS()` operations involving address x. Precisely, after the last step of E', a *single* process is scheduled to run solo, and executes an infinite sequence of `BDCAS()` calls such that each call successfully changes the value of the same pair (x, y) of addresses, where $y \in M_1$. It follows from Lemma 3.59 that each `BDCAS()` call takes a finite number of steps in expectation and so E'' contains an infinite number of successful $L[x]$.`choose&lock()` calls.

Finally, we assume that if $t \geq |E|$ then $\mathcal{T}_{B,t}(E'')$ indicates which one was the last step of E, and if $t \geq |E'|$ then $\mathcal{T}_{B,t}(E'')$ indicates also which one was the last step of E'.

3.5.4 Other Notation. For $k \geq 1$, let t_k denote the linearization point of the k-th successful method call $L[x]$.`choose&lock()` in E'', and let s_k be the linearization point of the k-th successful $L[x]$.`unlock()` call in E''. Also let $t_0 = s_0 = 0$. From Lemma 2.5, it follows that $t_k < s_k < t_{k+1}$, for all $k \geq 1$.

For a task τ, let r_τ denote the point when call $L[x]$.`propose(`τ`)` linearizes in E'', or $r_\tau = \infty$ if no such point exists. For $k \geq 0$, let Q_k and P_k be the following sets of tasks,

$$Q_k = \{\tau \colon \tau.add_0 = x,\ t_k < r_\tau < t_{k+2}\}, \quad P_k = \{\tau \colon \tau.add_0 = x,\ s_k < r_\tau < s_{k+1}\};$$

and let $p_k = |P_k|$. From Definition 2.26, we have that Q_k is the set of recent proposals on RC object $L[x]$ in execution $E''_{t_{k+2}}$.

For $k \geq 1$, let $\tau_k = L_{t_k}[x].val$ be the agreement value of $L[x]$ at point t_k in E''.
In the following, we use $\mathcal{T}_t$ as a shorthand for trace $\mathcal{T}_{B,x}(E''_t)$.

3.6 Distribution of Agreement-Value τ_k

We apply Theorems 2.27 and 2.28 to bound the distribution of τ_k in E''. From Lemma 3.56, we have that $O(n)$ method calls of object $L[x]$ are invoked between points s_k and s_{k+1} in E'', for any $k \geq 0$, thus requirement (R) is satisfied. Applying Theorem 2.27 we obtain:

Corollary 3.30. *Let $k \geq 0$. There is an integer random variable $g_k = g_k(\mathcal{T}_{s_{k+1}})$ with $0 \leq g_k \leq p_k$ and*

$$\mathbf{E}[g_k \mid E''_{s_k}] = (1/2) \cdot \mathbf{E}[p_k \mid E''_{s_k}], \tag{14}$$

such that the following hold.[2] *If $\tau \in Q_k$ then*

$$\Pr[\tau_{k+2} = \tau \mid \mathcal{T}_{t_{k+2}}] \leq 20/\max\{g_k, 20\}$$

and

$$\Pr[\tau_{k+2} = \perp \mid \mathcal{T}_{t_{k+2}}] \leq (1/\log n)^{g_k}.$$

Proof. It suffices to show that $\Pr[\tau_{k+2} = \tau \mid \mathcal{T}_{t_{k+2}}] \leq 20/\max\{g_k, 20\}$ from Theorem 2.27's result that $\Pr[\tau_{k+2} = \tau \mid \mathcal{T}_{t_{k+2}}] \leq 11/\max\{g_k, 11\} + 3/n$. Suppose that $g_k > 20$, otherwise the result is immediate. From Lemma 3.56, it follows that $g_k \leq p_k \leq 3n$, and so $n \geq g_k/3$. Thus,

$$\Pr[\tau_{k+2} = \tau \mid \mathcal{T}_{t_{k+2}}] \leq 11/g_k + 3/n \leq 11/g_k + 9/g_k = 20/g_k. \qquad \square$$

Similarly, Theorem 2.28 gives:

Corollary 3.31. *Let $k \geq 0$ and let integer j be such that $0 \leq j \cdot \lambda \leq k < (j+1)\lambda$. There is an event $\mathcal{B}_k = \mathcal{B}_k(E''_{s_k})$ such that*

$$\Pr[\mathcal{B}_k \mid E''_{s_{j\lambda}}] \leq 1/n^2, \tag{15}$$

and

$$\Pr[(\tau_{k+2} \notin Q_k \cup \{\perp\}) \wedge \bar{\mathcal{B}}_k \mid \mathcal{T}_{t_{k+2}}] \leq 6/\log^2 n.$$

The above statements compute the conditional distribution of τ_k given $\mathcal{T}_{t_k}$. Using this distribution, denoted $\pi(\cdot)$, Lemma 3.32 below computes for $t > t_k$ the conditional distribution of τ_k given $\mathcal{T}_t$ and $\tau_k \in Q_{k-2} \setminus (D_{t-1} \cup S_t)$, i.e., $r_{\tau_k} > t_{k-2}$ and task τ_k is not doomed before round t nor is successful after round t. The lemma implies if we know that $\tau_k \in Q_{k-2} \setminus (D_{t-1} \cup S_t)$, then knowing the complete trace $\mathcal{T}_t$ does not provide more information about τ_k than just knowing $\mathcal{T}_{t_k}$ and D_{t-1}.

Lemma 3.32. *Let $k \geq 2$. For any task τ and set of tasks T, let*

$$\pi(\tau) = \Pr[\tau_k = \tau \mid \mathcal{T}_{t_k}], \quad \pi(T) = \sum_{\tau \in T} \pi(\tau).$$

If $t > t_k$ and $\tau \in Q_{k-2} \setminus D_{t-1}$ then

$$\Pr[\tau_k = \tau \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D_{t-1} \cup S_t)] = \frac{\pi(\tau)}{\pi(Q_{k-2} \setminus D_{t-1})}.$$

[2] It is still true that $g_k = g_k(\mathcal{T}_{s_{k+1}})$, even though $\mathcal{T}_{s_{k+1}}$ does not specify the agreement value of $L[x]$, because g_k depends just on the random choices α_v of the proposals between s_k and s_{k+1}, and this information is contained in $\mathcal{T}_{s_{k+1}} = \mathcal{T}_{B,x}(E''_{s_{k+1}})$.

Proof. Suppose we fix trace $\mathcal{T}_{t_k}$. We prove by induction on $t \geq t_k$ that if $\tau \in Q_{k-2} \setminus D'_{t-1}$ then

$$\Pr[\tau_k = \tau \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D'_{t-1} \cup S_t)] = \pi(\tau)/\pi(Q_k \setminus D'_{t-1}), \tag{16}$$

where

$$D'_t = \begin{cases} D_t & \text{if } t \geq t_k \\ \emptyset & \text{if } t < t_k. \end{cases}$$

For the base case $t = t_k$, we have that event $\tau_k \in Q_{k-2} \setminus (D'_{t-1} \cup S_t)$ is the same as $\tau_k \in Q_{k-2}$, because $D'_{t_k-1} = \emptyset$ and from Definition 3.27 and Lemma 3.50 it follows $\tau_k \notin S_{t_k}$. Thus, for $t = t_k$, the left side of (16) equals

$$\Pr[\tau_k = \tau \mid \mathcal{T}_{\tau_k}, \tau_k \in Q_{k-2}].$$

And if $\tau \in Q_{k-2}$, the above probability equals $\pi(\tau)/\pi(Q_k) = \pi(\tau)/\pi(Q_k \setminus D'_{t_k-1})$. This proves the base case of the induction.

For the induction step, we fix $t \geq t_k$, and we show that if $\tau \in Q_{k-2} \setminus D_t$ then

$$\Pr[\tau_k = \tau \mid \mathcal{T}_{t+1}, \tau_k \in Q_{k-2} \setminus (D_t \cup S_{t+1})] = \pi(\tau)/\pi(Q_k \setminus D_t).$$

If if $\tau \in Q_{k-2} \setminus D_t$ then

$$\begin{aligned} \Pr[\tau_k = \tau \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D_t \cup S_t)] &= \frac{\Pr[\tau_k = \tau \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D'_{t-1} \cup S_t)]}{\Pr[\tau_k \in Q_{k-2} \setminus D_t \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D'_{t-1} \cup S_t)]} \\ &= \frac{\pi(\tau)/\pi(Q_k \setminus D'_{t-1})}{\pi(\tau_k \in Q_{k-2} \setminus D_t)/\pi(Q_k \setminus D'_{t-1})} \\ &= \frac{\pi(\tau)}{\pi(\tau_k \in Q_{k-2} \setminus D_t)}, \end{aligned}$$

where in the second line we applied induction hypothesis (16). We show next that if $\tau \in Q_{k-2} \setminus D_t$,

$$\Pr[\tau_k = \tau \mid \mathcal{T}_{t+1}, \tau_k \in Q_{k-2} \setminus (D_t \cup S_{t+1})] = \Pr[\tau_k = \tau \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D_t \cup S_t)]. \tag{17}$$

Combining that with the previous equation completes the proof of induction step. Thus, it remains to show (17).

To prove (17) we must argue that the additional information about step $t+1$ contained in $\mathcal{T}_{t+1}$, when $\tau_k \notin S_{t+1}$, does not change the distribution of τ_k. Let p denote the process that executes step $t+1$. If p did not have a pending next operation before step $t+1$, let mc denote the method call of B that p invokes in step $t+1$ (`BDCAS()` or `read()`) along with its arguments ($mc = \perp$ otherwise). Let $\mathcal{T}_t^+ = \mathcal{T}_t \cup \{(p, mc)\}$. We can replace $\mathcal{T}_t$ on the right side on (17) by $\mathcal{T}_t^+$, because revealing p, mc does not reveal any new information about τ_k, since $\mathcal{T}_t$ already contains all information that the adversary knows of at the end of step t.

Suppose that we fix $\mathcal{T}_t^+$, and that $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$. Next we consider all possibilities for step $t+1$.

Case 1: Step $t+1$ is a step of call `read(a)`. From $\mathcal{T}_t^+$, we can infer if step $t+1$ is the first step of call `read(a)`: Process p executes the first step of `read(a)` if indicated in $mc \neq \perp$, or if in its last step before step $t+1$, p just finished the call in line 48 or finished the first `read()` call in line 49 (and the `BDCAS()` call did not return at that step).

If $a \neq x$ then we can infer from $\mathcal{T}_t^+$ exactly which operation process p executes at step $t+1$ (including its outcome), as well as the return value of the call if a return command is executed. This is also true if $a = x$, even though $\mathcal{T}_t^+$ does not indicate the value of $A[x]$: It suffices that $\mathcal{T}_t^+$ contains the points at which the value of $A[x]$ changes, to infer the value of $A[x].stat$ and the value of $A[x].old_0$, and also to infer the value of $A[x].new_0$ when $A[x].stat = \text{True}$.

Therefore, $\mathcal{T}_{t+1}$ does not contain any information that cannot be inferred from $\mathcal{T}_t^+$.

Case 2: Step $t+1$ is a step of call `finish(a)`. From $\mathcal{T}_t^+$, we can infer if step $t+1$ is the first step of call `finish(a)`. There are several possibilities for the previous step for p, but only two involve if statements. First, p invokes `finish(a)` call in line 48 if in its previous step p executed line 54 and the if-condition was false. However, the value of the if-condition can be inferred from $\mathcal{T}_t$ (this is true also when $a_0 = x$, since $\mathcal{T}_t$ contains the points at which the value of $A[x]$ changes). Second, p invokes `finish(a)` call in line 57 if in its previous step p evaluated to false one of the two if-conditions in line 55. However, we will argue in Case 3 that when $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$, trace $\mathcal{T}_t$ contains the value of γ' if $\gamma' \neq \tau_k$, and if $\gamma' = \tau_k$ then then we can infer from $\mathcal{T}_t$ the value of $\gamma'.old_0$.

We can assume that $a \neq x$, since otherwise we can infer from $\mathcal{T}_t^+$ exactly which operation process p executes at step $t+1$ (including its outcome). Suppose also that p does not executes an operation from lines 59–61 at step $t+1$, since they are just read operations.

First, suppose that the task that p reads on $A[x]$ in line 59 of the call is $\tau \neq \tau_k$. Then $A[x] = \tau$ at some point before t (precisely, at the point when p reads $A[x]$). Also $L_t[x].val = \tau_k$, otherwise Lemmas 3.42 and 3.51 would contradict that $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$. It follows that a successful `unlock(τ)` call was invoked before t_k, and thus the value of τ can be inferred from $\mathcal{T}_t$. Therefore, as in case $a \neq x$, we can infer from $\mathcal{T}_t^+$ exactly which operation process p executes at step $t+1$.

Next we assume that p reads task τ_k on $A[x]$ in line 59. From Lemma 3.44 and assumption $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$, it follows that

(i) if p executes line 62 at step $t+1$, the `CAS()` operation fails,
(ii) if p executes line 64 at $t+1$, the `CAS()` operation is successful, and
(iii) p does not execute the `CAS()` operation in line 67 at $t+1$.

From (i), if p executes line 62, then nothing happens. (Note that if the `CAS()` were successful, it would reveal $\tau_k.add_1$!) From (ii), if p executes line 64 at $t+1$, then $\tau_k \in S_{t+1}$. Hence, this case is not relevant to the proof of (17), because the probability on the left side is conditional on $\tau_k \notin S_{t+1}$. Moreover, p does not execute lines 65 and 66 at $t+1$, because p must have previously executed line 64. Finally, if p executes an operation in line 63, then nothing happens.

Therefore, $\mathcal{T}_{t+1}$ does not contain any information that cannot be inferred from $\mathcal{T}_t^+$.

Case 3: Step $t+1$ is a step of call `BDCAS()`. We have already considered the case where step $t+1$ is a step of a method `finish()` or `read()` invoked in one of lines 48, 49 and 57. Also, if p creates a new task at step $t+1$, in line 50, then this information can be inferred from $\mathcal{T}_t^+$. So, we assume that p executes an operation in one of the remaining lines, lines 51–56 and 58, at step $t+1$.

Suppose first that the `DCAS()` operation is not applied to address x. Then the exact operation that p executes at step $t+1$ (and its outcome) can be inferred from $\mathcal{T}_t^+$, in all cases except for two which involve a randomized decision: when p executes line 2 of the `propose()` call invoked in line 51; and when p executes line 34 during an `unlock()` call invoked in line 58. In both cases, however, knowing the random values involved in the operations does not provide any additional information about τ_k.

For the remainder of the proof, we assume that the `DCAS()` operation is applied to address x. If p executes the `propose()` operation in line 51 at step $t+1$, then, as before, knowing the randomness involved in the operations does not reveal any information about τ_k (recall that $\mathcal{T}_{t+1}$ contains random value α). If p executes a step of `choose&lock()` in line 52, then $\mathcal{T}_{t+1}$ provides no new information as B does not change, because from Lemma 3.51 and assumption $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$, it follows that $L_{t+1}[x].val = L_t[x].val = \tau_k$. Moreover, we can infer from $\mathcal{T}_t^+$ whether $t+1$ is the last step of call `choose&lock()`. If p executes the RC `read()` operation in line 53 at step $t+1$, then no new information is revealed. The same is true if p executes the if operation in line 54, and moreover the value of the if-condition can be inferred from H_t^+. It remains to consider the case where p executes an operation in one of lines 55, 56 and 58 in step $t+1$.

First we assume that the value that p read in line 53 during the same while-loop iteration is $\tau \neq \tau_k$. As noted above, from Lemma 3.51 and assumption $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$, it follows that $L_{t+1}[x].val = L_t[x].val = \tau_k$. Thus, $L_y[x].val = \tau$ for some $y < t$, which implies that a successful $L[x]$.unlock(τ) call was invoked before t. And since $\mathcal{T}_t$ contains the arguments of all method calls invoked, we conclude that $\mathcal{T}_t$ contains the value of τ.

Suppose that p executes the operation in line 55 at step $t+1$ (only the second of the two if-conditions involves a shared memory operation). This happens if $\tau \neq \bot$, and in its previous step p evaluated to true the if-condition in line 54. These can be inferred from $\mathcal{T}_t^+$, as well as the value of the second condition in line 55, evaluated at step $t+1$. Hence, $\mathcal{T}_{t+1}$ does not contain any additional information.

Next suppose that p executes the CAS() operation in line 56 at step $t+1$. This happens if in its previous step, p evaluated to true the if-conditions in line 54 (which can be inferred from $\mathcal{T}_t^+$). The outcome of the operation can also be inferred from $\mathcal{T}_t^+$.

Last suppose that p executes a step of operation unlock(τ) in line 58 at step $t+1$. Recall that the value of τ is contained in $\mathcal{T}_t$, and $L_{t+1}[x].val = \tau_k \neq \tau$. It follows from Claim 2.19 that step $t+1$ does not change the values of any of $L[x].S$, $L[x].P$, or $L[x].T[1]$.Hence, $\mathcal{T}_{t+1}$ does not contain more information than $\mathcal{T}_t^+$.

Finally, we need to consider the case where p executes an operation in one of lines 55, 56 and 58 in step $t+1$, and p previously read value τ_k in line 53 in the same while-loop iteration. From assumption $\tau_k \in Q_{k-2} \setminus (D_t \cup S_t)$ it follows that $\tau_k \neq \bot$ and $\tau_k.old_0 = B_t[x].val'$, where $B_t[x].val'$ can be inferred from $\mathcal{T}_t$. If p executes line 55 at step $t+1$ (which is possible since $\tau_k \neq \bot$), we can then infer from H_t^+ the value of the second if-condition. If p executes the CAS() operation in line 56 at step $t+1$, then the outcome, success or failure, of the CAS operation can also be inferred from $\mathcal{T}_t^+$. Finally, it is not possible that p executes a step of call unlock(τ_k) in line 58, because otherwise Lemma 3.60 would contradict that $\tau_k \notin D_t \cup S_t$.

This concludes the proof of (17), and the proof of Lemma 3.32. ◻

3.7 k-Fated Tasks

We lower bound the expected number of tasks $\tau \in Q_{k-2}$ that are doomed or successful by the time that task τ_k itself is also doomed or successful. We call those tasks k-fated, and give their formal definition below.

Definition 3.33 (k-Fated Tasks)**.** For $k \geq 2$, let ρ_k be the following point

$$\rho_k = \begin{cases} \min\{t\colon \tau_k \in D_t \cup S_t,\ t \geq t_k\}, & \text{if } \tau_k \in Q_{k-2} \\ t_k, & \text{otherwise.} \end{cases}$$

We define the set F_k of *k-fated* tasks as

$$F_k = Q_{k-2} \cap (D_{\rho_k} \cup S_{\rho_k}),$$

and let $f_k = |F_k|$.

From Lemma 3.45, it follows that point ρ_k, $k \geq 2$, always exists. Also, from Lemma 3.51, it follows that for all $k \geq 2$,

$$t_k \leq \rho_k < t_{k+1}. \tag{18}$$

To show the main result we will need the next two lemmas. The first one is based on Lemma 3.32.

Lemma 3.34. *Let $k \geq 2$, and for any $t \geq 0$ let*

$$Z_t = Q_{k-2} \cap (D_t \cup S_t).$$

If $\tau \in Q_{k-2}$ and $t \geq t_k$ then

$$\Pr[\tau_k = \tau \mid \mathcal{T}_{t_k}, \tau_k \in Q_{k-2},\ t \leq \rho_k,\ Z_{t_k} \ldots Z_t] = \Pr[\tau_k = \tau \mid \mathcal{T}_{t_k}, \tau_k \in Q_{k-2},\ t \leq \rho_k,\ Z_{t_k} \ldots Z_{t-1}].$$

Proof. Suppose that we fix trace $\mathcal{T}_{t_k}$. First we argue that no task in Q_{k-2} is successful at point ρ_k except possibly for τ_{k-1} and τ_k, i.e.,

$$(Q_{k-2} \setminus \{\tau_{k-1}, \tau_k\}) \cap S_{\rho_k} = \emptyset.$$

Suppose that $\tau \in Q_{k-2} \cap S_{\rho_k}$. Then Lemma 3.50 implies $L_y[x].val = \tau$ at some point $y \leq \rho_k$. From Lemma 3.51 and the definition of ρ_k, it follows that $L_t[x].val$ remains the same for all $t_k \leq t \leq \rho_k$. Thus, $y \leq t_k$. Moreover, $y \geq t_{k-1}$ because $r_\tau > t_{k-2}$, hence the earliest point that τ can be written to $L[x].val$ is t_{k-1}. Since τ_{k-1} and τ_k are the only values of $L_y[x].val$ for $t_{k-1} \leq y \leq t_k$, it follows that $\tau \in \{\tau_{k-1}, \tau_k\}$.

Next we argue that

$$\tau_k \in S_{\rho_k} \implies Q_{k-2} \setminus \{\tau_{k-1}, \tau_k\} \subseteq D_{\rho_k}.$$

Suppose that $\tau_k \in S_{\rho_k}$. Then Lemma 3.50 implies $\rho_k > t_k$. At point ρ_k, a process p changes the value of $A[\tau_k.add_1]$ to τ_k, by executing a successful `CAS()` operation in line 64. This operation changes also the imminent interpreted value $B[x].val'$. As we saw above, no task $\tau \in Q_{k-2} \setminus \{\tau_{k-1}, \tau_{k-2}\}$ gets written to $L[x].val$ before ρ_k, and thus neither does it get written to array A before ρ_k. Hence, the change in the imminent interpreted value $B[x].val'$ at point ρ_k implies $\tau \in D_{\rho_k}$, for all $\tau \in Q_{k-2} \setminus \{\tau_{k-1}, \tau_{k-2}\}$.

Let

$$A = \{\tau_{k-1}\} \cap Q_{k-2} \cap S_{t_k},$$

i.e., $A = \{\tau_{k-1}\}$ if $\tau \in Q_{k-2} \cap S_{t_k}$, and $A = \emptyset$ otherwise. The value of τ_{k-1} can be inferred from $\mathcal{T}_{t_k}$, because between points t_{k-1} and t_k a successful `unlock(`τ_{k-t}`)` call is invoked, and $\mathcal{T}_{t_k}$ contains the arguments of all invoked method calls. Moreover, we have

$$\tau_{k-1} \in Q_{k-2} \implies \tau_{k-1} \in D_{t_k} \cap S_{t_k},$$

because if $\tau_{k-1} \in Q_{k-2}$ then $\tau_{k-1} \in D_{\rho_{k-1}} \cap S_{\rho_{k-1}}$, and $\rho_{k-1} < t_k$ by (18). Thus set A can be inferred from $\mathcal{T}_{t_k}$.

Combining the above, and the definition of $Z_t = Q_{k-2} \cap (D_t \cup S_t)$, we obtain that

$$Z_t = \begin{cases} (Q_{k-2} \cap D_t) \cup A, & \text{if } t_k \leq t < \rho_k, \text{ or if } t = \rho_k \text{ and } \tau_k \notin S_{\rho_k} \\ Q_{k-2}, & \text{if } t = \rho_k \text{ and } \tau_k \in S_{\rho_k}. \end{cases}$$

Let $t \geq t_k$, and suppose that $t \leq \rho_k$. Let p be the process that executes step t. We distinguish three cases, depending on the following event:

> $\mathcal{E}$ is the event that process p just finished line 63 of call `finish(`x`)` in its last step before step t, and moreover p read $A[x] = \tau_k$ in line 59 of the same call.

Case 1: event $\mathcal{E}$ holds. In this case, $t = \rho_k > t_k$ and $\tau_k \in S_t$. The reason is that in step t process p executes either line 64 or line 67, since it just completed line 63 in its previous step. But, from Lemma 3.44, if p executed a failed `CAS()` in line 64, or executed line 67, it would imply $\rho_k < t$. Thus, the only possibility left is that p executes a successful `CAS()` in line 64, and thus $t = \rho_k$ and $\tau_k \in S_t$.

Then $Z_t = Q_{k-2}$, as we saw above. Therefore, the value of Z_t is completely determined in this case (as Q_{k-2} can be inferred from $\mathcal{T}_{t_k}$). It follows that for any $\tau \in Q_{k-2}$,

$$\Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t \leq \rho_k,\, \mathcal{E},\, Z_{t_k}, \ldots, Z_t] = \Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t \leq \rho_k,\, \mathcal{E},\, Z_{t_k}, \ldots, Z_{t-1}].$$

Case 2: event $\bar{\mathcal{E}}$ holds and $t > t_k$. In this case we have that $\tau_k \notin S_t$. Then, from Lemma 3.32, if $\tau \in Q_{k-2} \setminus D_{t-1}$,

$$\Pr[\tau_k = \tau \mid \mathcal{T}_t, \tau_k \in Q_{k-2} \setminus (D_{t-1} \cup S_t)] = \pi(\tau)/\pi(Q_{k-2} \setminus D_{t-1}).$$

For the left side we have that event $\tau_k \in Q_{k-2} \wedge t \leq \rho_k \wedge \bar{\mathcal{E}}$ implies $\tau_k \in Q_{k-2} \setminus (D_{t-1} \cup S_t)$, and moreover $D_{t_k}, \ldots, D_t$ and thus $Z_{t_k}, \ldots, Z_t$ can be inferred from $\mathcal{T}_t$. For the right side we have that $\pi(Q_{k-2} \setminus D_{t-1}) = \pi(Q_{k-2} \setminus Z_{t-1})$. Therefore, the above equation implies that if $\tau \in Q_{k-2} \setminus Z_{t-1}$,

$$\Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t \leq \rho_k,\, \bar{\mathcal{E}},\, Z_{t_k} \ldots Z_t] = \pi(\tau)/\pi(Q_{k-2} \setminus Z_{t-1}).$$

(Thus, the probability is zero if $\tau \notin Q_{k-2} \setminus Z_{t-1}$.) Since the right side is independent of Z_t, it follows

$$\Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t \le \rho_k,\, \bar{\mathcal{E}},\, Z_{t_k} \dots Z_t] = \Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t \le \rho_k,\, \bar{\mathcal{E}},\, Z_{t_k} \dots Z_{t-1}].$$

Combining Cases 2 and 3 proves the lemma for the case where $t > t_k$.

Case 3: $t = t_k$. For every $\tau \in Q_{k-2}$, we have

$$\Pr[\tau_k = \tau \mid \tau \in Q_{k-2}] = \pi(\tau)/\pi(Q_{k-2}).$$

As in Case 2, D_{t_k} and thus Z_{t_k} can be inferred from $\mathcal{T}_{t_k}$, and the right side does not depend on Z_{t_k}. Therefore, the lemma holds in this case, as well. □

The next lemma analyzes a very simple game.

Lemma 3.35 (Guessing Game). *Consider the following game. An element g^* of a finite set G is sampled at random according to a distribution λ, such that $\max_{g \in G} \lambda(g) \le \epsilon < 1$. The player, who knows G and λ, tries to guess g^* in a sequence of attempts. In each attempt i, the player proposes a set $G_i \subseteq G$. If $g^* \in G_i$ the game ends and the player's score is $\sum_{j \le i} |G_j|$. Then, for any player strategy, the expected score is at least $(2\epsilon)^{-1}$.*

Proof. Clearly, proposing more than one elements in an attempt, or proposing an element twice can only increase the player's score. Hence, we can assume that the player just decides an order $g_1, g_2, \dots, g_{|G|}$ in which the elements are proposed, one element per attempt. The expected score is then

$$\sum_{1 \le i \le |G|} i \cdot \lambda(g_i) \ge \sum_{1 \le i \le \lfloor 1/\epsilon \rfloor} i \cdot 1/\lfloor 1/\epsilon \rfloor,$$

where the inequality holds because $\lambda(g_i) \le \epsilon \le 1/\lfloor 1/\epsilon \rfloor$, and thus we can obtain a lower bound of the sum by assigning to each of the first $\lfloor 1/\epsilon \rfloor$ elements probability $1/\lfloor 1/\epsilon \rfloor$, and to the remaining ones probability zero. The desired lower bound then follows from

$$\sum_{1 \le i \le \lfloor 1/\epsilon \rfloor} i \cdot 1/\lfloor 1/\epsilon \rfloor = \frac{\lfloor 1/\epsilon \rfloor \cdot (1 + \lfloor 1/\epsilon \rfloor)}{2} \cdot 1/\lfloor 1/\epsilon \rfloor \ge \frac{1/\epsilon}{2}.$$ □

We are now ready to prove the main lemma of this section, which provides a lower bound on the expected value of f_k given $\mathcal{T}_{t_k}$.

Lemma 3.36. *For $k \ge 2$ and $n \ge 157$,*

$$\mathbf{E}[f_k \mid \mathcal{T}_{t_k}] \ge \frac{1}{40} \cdot (1 - 4\Pr[\mathcal{B}_{k-2} \mid \mathcal{T}_{t_k}]) \cdot g_{k-2}.$$

Proof. Suppose that we fix $\mathcal{T}_{t_k}$ and that $g_{k-2} > 0$, otherwise the lemma holds trivially. As in Lemma 3.32, we denote by $\pi(\cdot)$ the probability distribution of τ_k. We have

$$\mathbf{E}[f_k] \ge \mathbf{E}[f_k \mid \tau_k \in Q_{k-2}] \cdot \Pr[\tau_k \in Q_{k-2}] \ge \mathbf{E}[f_k \mid \tau_k \in Q_{k-2}] \cdot \pi(Q_{k-2}). \tag{19}$$

Next we lower bound $\mathbf{E}[f_k \mid \tau_k \in Q_{k-2}]$, by expressing f_k as the score of the game described in Lemma 3.35, using Lemma 3.34. Let $Z_t = Q_{k-2} \cap (D_t \cup S_t)$, for $t \ge 0$, as in Lemma 3.34. For each $t > t_k$, let

$$G_t = Z_t \setminus Z_{t-1},$$

and let $G_{t_k} = Z_{t_k}$. Then Lemma 3.34 gives that, for any $\tau \in Q_{k-2}$ and $t \ge t_k$,

$$\Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t_k \le \rho_k,\, G_{t_k} \dots G_t] = \Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2},\, t_k \le \rho_k,\, G_{t_k} \dots G_{t-1}].$$

Therefore, given $\tau_k \in Q_{k-2}$, $t \leq \rho_k$, $G_{t_k} \dots G_{t-1}$, we have that G_t is independent of τ_k. It follows that the sequence $G_{t_k}, G_{t_k+1}, \dots$ corresponds to a randomized player strategy for the guessing game of Lemma 3.35, where $G = Q_{k-2}$, $g^* = \tau_k$, and for each $\tau \in Q_{k-2}$,

$$\lambda(\tau) = \Pr[\tau_k = \tau \mid \tau_k \in Q_{k-2}] = \pi(\tau)/\pi(Q_k).$$

Then Lemma 3.35 gives

$$\mathbf{E}\left[\sum_{t_k \leq t \leq \rho_k} |G_t| \,\middle|\, \tau_k \in Q_{k-2}\right] \geq \left(2 \max_{\tau \in Q_{k-2}} \lambda(\tau)\right)^{-1} = \left(2 \max_{\tau \in Q_{k-2}} \pi(\tau)/\pi(Q_k)\right)^{-1} \geq \pi(Q_k) \cdot g_{k-2}/20,$$

where for the last inequality we used Corollary 3.30 to obtain that $\pi(\tau) \leq 20/g_{k-2}$ (recall we have assumed $g_{k-2} > 0$). And since $F_k = Q_{k-2} \cap (D_{\rho_k} \cup S_{\rho_k}) = Z_{\rho_k} = \bigcup_{t_k \leq t \leq \rho_k} G_t$, we have $f_k = |F_k| = \sum_{t_k \leq t \leq \rho_k} |G_t|$, and the inequality above yields

$$\mathbf{E}[f_k \mid \tau_k \in Q_{k-2}] \geq \pi(Q_{k-2}) \cdot g_{k-2}/20.$$

Substituting this to (19) gives

$$\mathbf{E}[f_k] \geq (\pi(Q_k))^2 \cdot g_{k-2}/20.$$

Finally,

$$\begin{aligned}
(\pi(Q_k))^2 &= (1 - \Pr[\tau_k \notin Q_k])^2 \geq 1 - 2\Pr[\tau_k \notin Q_k] \\
&\geq 1 - 2(\Pr[\tau_k = \bot] + \Pr[(\tau_k \notin Q_{k-2} \cup \{\bot\}) \wedge \bar{\mathcal{B}}_k] + \Pr[\mathcal{B}_k]) \\
&\geq 1 - 2((1/\log n)^{g_{k-2}} + 6/\log^2 n + \Pr[\mathcal{B}_k]) \\
&\geq 1/2 - 2\Pr[\mathcal{B}_k],
\end{aligned}$$

where in the second-last line we used Corollaries 3.30 and 3.31, and the last line holds for $n \geq 157$. Substituting this to the previous inequality gives

$$\mathbf{E}[f_k] \geq (1/2 - 2\Pr[\mathcal{B}_k]) \cdot g_{k-2}/20,$$

which concludes the proof of Lemma 3.36. ◻

3.8 Lower Bounding the Sum of k-Fated Tasks

In this section and the next one, we establish respectively a lower bound and an upper bound on the sum of f_k in execution E'. In particular, in this section we lower bound the expectation of that sum in terms of the expected number of `propose()` operations on object $L[x]$. We introduce some notation first.

Definition 3.37 (C_x and Π_x)**.** By C_x we denote the number of successful $L[x]$.`choose&lock()` calls in E',

$$C_x = \max\{k \colon t_k \leq |E'|\}.$$

By Π_x we denote the number of $L[x]$.`propose()` operations in E',

$$\Pi_x = |\{\tau \colon \tau.add_0 = x,\ r_\tau \leq |E'|\}|.$$

Lemma 3.38. *For all $n \geq 531$,* [3]

$$\mathbf{E}\left[\sum_{2 \leq k \leq C_x+2} f_k\right] \geq \frac{1}{160} \cdot \mathbf{E}[\Pi_x].$$

[3] We sum for k up to $C_x + 2$ rather than up to C_x for a technical reason, but we will see later that the difference is at most a constant.

Proof. We have

$$\mathbf{E}\left[\sum_{2\le k\le C_x+2} f_k\right] = \sum_{k\ge 2}\mathbf{E}[f_k\cdot \mathbb{1}_{k\le C_x+2}].$$

Recall that if $t \ge |E'|$ then $\mathcal{T}_t$ indicates which one was the last step of E'. Hence, we can infer from $\mathcal{T}_{t_k}$ if $t_k \le |E'|$, and thus if $k \le C_x$. In fact, we can infer from $\mathcal{T}_{t_k}$ if $k \le C_x + j$ for any given $j \ge 0$. We will use this observation several times below.

For $k \ge 2$,

$$\begin{aligned}\mathbf{E}[f_k\cdot \mathbb{1}_{k\le C_x+2}\mid \mathcal{T}_{t_k}] &= \mathbb{1}_{k\le C_x+2}\cdot \mathbf{E}[f_k\mid \mathcal{T}_{t_k}]\\ &\ge \mathbb{1}_{k\le C_x+2}\cdot \frac{1}{40}\cdot(1-4\Pr[\mathcal{B}_{k-2}\mid \mathcal{T}_{t_k}])\cdot g_{k-2}\\ &= \frac{1}{40}\cdot \mathbb{1}_{k\le C_x+2}\cdot g_{k-2} - \frac{1}{10}\cdot \Pr[k\le C_x+2\wedge \mathcal{B}_{k-2}\mid \mathcal{T}_{t_k}]\cdot g_{k-2},\end{aligned} \tag{20}$$

where the first equation holds because we can infer from $\mathcal{T}_{t_k}$ if $k \le C_x + 2$, and the second equation follows from Lemma 3.36.

For the first term in the last line of (20), we have

$$\begin{aligned}\mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot g_{k-2}\mid \mathcal{T}_{s_{k-2}}] &= \mathbb{1}_{k\le C_x+2}\cdot \mathbf{E}[g_{k-2}\mid \mathcal{T}_{s_{k-2}}]\\ &= \mathbb{1}_{k\le C_x+2}\cdot \mathbf{E}[p_{k-2}\mid \mathcal{T}_{s_{k-2}}]/2\\ &= \mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot p_{k-2}\mid \mathcal{T}_{s_{k-2}}]/2,\end{aligned}$$

where the first equation holds because we can infer if $k \le C_x + 2$ from $\mathcal{T}_{t_{k-2}}$ and thus also from $\mathcal{T}_{s_{k-2}}$,[4] and the second equation holds because of equation (14) in Corollary 3.30. It follows

$$\mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot g_{k-2}] = \mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot p_{k-2}]/2. \tag{21}$$

For the second term in the last line of (20), we have

$$\Pr[k\le C_x+2\wedge \mathcal{B}_{k-2}\mid \mathcal{T}_{t_k}]\cdot g_{k-2} \le \Pr[k\le C_x+2+\lambda\wedge \mathcal{B}_{k-2}\mid \mathcal{T}_{t_k}]\cdot 3n,$$

because $g_{k-2} \le p_{k-2} \le 3n$, where the second inequality follows from Lemma 3.56. Let $j \ge 0$ be such that $0 \le j\cdot\lambda \le k-2 < (j+1)\lambda$. Then

$$\Pr[k\le C_x+2+\lambda\wedge \mathcal{B}_{k-2}\mid \mathcal{T}_{s_{j\lambda}}] = \mathbb{1}_{k\le C_x+2+\lambda}\cdot \Pr[\mathcal{B}_{k-2}\mid \mathcal{T}_{s_{j\lambda}}] \le \mathbb{1}_{k\le C_x+2+\lambda}\cdot n^{-2}.$$

where the last inequality follows from equation (15) in Corollary 3.31. Combining the last two equations gives

$$\mathbf{E}[\Pr[k\le C_x+2\wedge \mathcal{B}_{k-2}\mid \mathcal{T}_{t_k}]\cdot g_{k-2}] \le \frac{3}{n}\cdot \Pr[k\le C_x+2+\lambda]. \tag{22}$$

From (20)–(22), we obtain

$$\begin{aligned}\mathbf{E}[f_k\cdot \mathbb{1}_{k\le C_x+2}] &\ge \frac{1}{40}\cdot \mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot g_{k-2}] - \frac{1}{10}\cdot \Pr[k\le C_x+2\wedge \mathcal{B}_{k-2}]\cdot g_{k-2}\\ &\ge \frac{1}{80}\cdot \mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot p_{k-2}] - \frac{3}{10n}\cdot \Pr[k\le C_x+2+\lambda].\end{aligned}$$

[4]However we may not be able to infer if $k \le C_x$, and this is the reason for choosing the sum range to be up to $C_x + 2$ rather than up to C_x.

Substituting that to the very first equation of the proof, gives

$$\begin{aligned}\mathbf{E}\left[\sum_{2\le k\le C_x+2} f_k\right] &\ge \frac{1}{80}\cdot\sum_{k\ge 2}\mathbf{E}[\mathbb{1}_{k\le C_x+2}\cdot p_{k-2}] - \frac{3}{10n}\cdot\sum_{k\ge 2}\Pr[k\le C_x+2+\lambda]\\ &= \frac{1}{80}\cdot\mathbf{E}\left[\sum_{2\le k\le C_x+2} p_{k-2}\right] - \frac{3}{10n}\cdot\mathbf{E}[C_x+\lambda+1]\\ &= \frac{1}{80}\cdot\mathbf{E}\left[\sum_{0\le k\le C_x} p_k\right] - \frac{3}{10n}\cdot(\mathbf{E}[C_x]+\lambda+1).\end{aligned}$$

Finally, since

$$\sum_{0\le k\le C_x} p_k \ge \Pi_x,$$

and

$$C_x \le \Pi_x,$$

we conclude that

$$\mathbf{E}\left[\sum_{2\le k\le C_x+2} f_k\right] \ge \frac{1}{80}\cdot\mathbf{E}[\Pi_x] - \frac{3}{10n}\cdot(\mathbf{E}[\Pi_x]+\lambda+1) \ge \frac{1}{80}\cdot\mathbf{E}[\Pi_x] - \frac{3}{10n}\cdot(\mathbf{E}[\Pi_x]+\log n+1) \ge \frac{1}{160}(\mathbf{E}[\Pi_x]+\log n+1),$$

for all $n \ge 531$. □

3.9 Upper Bounding the Sum of k-Fated Tasks

We compute now an upper bound on the sum of f_k, in terms of the number N_x of BDCAS() operations on address x invoked in E. Note that N_x is also the number of BDCAS() operations on address x invoked in E'.

Lemma 3.39. *There is a constant $c'' > 0$ such that*

$$\sum_{2\le k\le C_x+2} f_k \le c''\cdot N_x.$$

Proof. Suppose that process p invokes a BDCAS() operation involving address x. Let Γ be the set of tasks that p creates during that operation, and for $k \ge 0$ let

$$\Gamma_k = \Gamma\cap Q_k = \{\tau\colon \tau\in\Gamma,\ t_k < r_\tau < t_{k+2}\}.$$

From Lemma 3.55, it follows that for all $k \ge 0$,

$$|\Gamma_k| \le c_1,$$

for some constant c_1. We also argue that the following holds for some constant c_2, for any $k \ge 0$,

$$\Gamma\cap F_{k+2}\neq\emptyset \implies |\{\tau\colon \tau\in\Gamma,\ r_\tau\ge t_{k+3}\}| \le c_2.$$

Indeed if $\tau\in\Gamma\cap F_{k+2}$ then $\tau\in D_{\rho_{k+2}}\cap S_{\rho_{k+2}}$, and lemma Lemma 3.54 implies that $|\{\tau\colon \tau\in\Gamma,\ r_\tau\ge\rho_{k+2}\}| \le c_2$. Moreover, from (18), $\rho_{k+2} < t_{k+3}$.

We will now use the above observations to bound $\sum_{0\le k\le C_k}|\Gamma\cap F_{k+2}|$, which is precisely the contribution of the BDCAS() operation to the sum $\sum_{2\le k\le C_x+2} f_k$ we are interested in. Suppose that $\sum_{0\le k\le C_k}|\Gamma\cap F_{k+2}|\neq 0$, and

let $k^* = \min\{k : F_{k+2} \cap \Gamma \neq \emptyset\}$. Then

$$\sum_{0\le k\le C_x} |\Gamma \cap F_{k+2}| \le \sum_{k\ge k^*} |\Gamma \cap F_{k+2}| \le \sum_{k\ge k^*} |\Gamma \cap Q_k|$$
$$= \sum_{k\ge k^*} |\Gamma_k| = \sum_{k^*\le k\le k^*+2} |\Gamma_k| + \sum_{k\ge k^*+3} |\Gamma_k| \le 3c_1 + 2c_2.$$

This is a bound on the contribution of a single `BDCAS()` operation. The total number of `BDCAS()` operation on x invoked before step t_{C_x+2} is at most $N_x + 2$, because N_x such operations are executed in E', and at most two are invoked before t_{C_x+2} by the process that runs solo in the extension E'' of E'. It follows

$$\sum_{2\le k\le C_x+2} f_k \le (3c_1 + 2c_2) \cdot (N_x + 2).$$ □

3.10 Completing the Proof of Theorem 3.25

Combining Lemmas 3.38 and 3.39, we obtain that the expected number of proposals on $L[x]$ in E' is at most linear in the expected number of `BDCAS()` operations on address x,

$$\mathbf{E}[\Pi_x] \le (c''/c') \cdot \mathbf{E}[N_x].$$

The final step is to bound the number of steps in terms of Π_x. Let T_x denote the number of steps of `BDCAS()` operations on address x in E'. Then T_x is an upper bound on the number of steps of `BDCAS()` operations on x in E. Since all method calls are completed in E', it is immediate from the algorithm that the number of steps of the `BDCAS()` operations on x in E', excluding the steps of methods of $L[x]$ invoked by `BDCAS()`, is at most $c_1 \cdot \Pi_x$, for a constant c_1. The number of $L[x]$.`choose&lock()` calls and the number of of $L[x]$.`read()` calls are exactly the same as the number Π_x of $L[x]$.`propose()` calls, and each `choose&lock()` call involves at most $c_2 \log\log n$ steps, for some constant c_2, while each `propose()` and `read()` call consists of one step. Finally, the number of $L[x]$.`unlock()` calls is at most Π_x, and Corollary 2.13 gives that the expected number of their steps is at most $c_3 \log\log n \cdot \mathbf{E}[\Pi_t]$. It follows

$$\mathbf{E}[T_x] \le c_1 \cdot \mathbf{E}[\Pi_x] + c_2 \log\log n \cdot \mathbf{E}[\Pi_x] + 2 \cdot \mathbf{E}[\Pi_x] + c_3 \log\log n \cdot \mathbf{E}[\Pi_x] = c \log\log n \cdot \mathbf{E}[\Pi_x],$$

for a constant c. Combining this with the previous inequality yields

$$\mathbf{E}[T_x] \le (c \cdot c''/c') \cdot \log\log n \cdot \mathbf{E}[N_x],$$

and concludes the proof of Theorem 3.25.

3.11 Auxiliary Lemmas

We show that the imminent interpreted value of $B[a]$, if different from the interpreted value of $B[a]$, is the next interpreted value that $B[a]$ takes.

Claim 3.40. *If $B_t[a].val \neq B_t[a].val'$, then for the earliest point $t' > t$ at which $B_{t'}[a].val \neq B_t[a].val$, $B_{t'}[a].val = B_t[a].val'$. Also, the task stored in $A[a]$ is not replaced in the interval $(t, t']$.*

Proof. If $B_t[a].val \neq B_t[a].val'$, then by Definition 3.26, $A_t[a].stat \neq$ True, $A_t[A_t[a].add_1] = A_t[a]$, $B_t[a].val = A_t[a].old_i$, and $B_t[a].val' = A_t[a].new_i$, as this is the only configuration in which interpreted values and imminent interpreted values differ. From Lemma 3.11 and that $A_t[A_t[a].add_1] = A_t[a]$, it follows that $A_t[a].stat = \bot$. From Claim 3.6 and that each step consists of only one shared memory operation, $A_t[a].stat$ changes to True before $A[a]$ is replaced with a different task. By definition of interpreted value, the point when $A_t[a].stat$ changes to True is t', and $B_{t'}[a].val = A_t[a].new_i = B_t[a].val'$. □

Corollary 3.41. *For any $t \ge 0$, either $B_t[a].val = B_t[a].val'$ or $B_t[a].val \prec_a B_t[a].val'$.*

PROOF. Suppose that $B_t[a].val \neq B_t[a].val'$. As in Claim 3.40, we have that by Definition 3.26, $A_t[a].stat = \bot$, $A_t[A_t[a].add_1] = A_t[a]$, $B_t[a].val = A_t[a].old_i$, and $B_t[a].val' = A_t[a].new_i$. Since $A_t[a].old_i \neq A_t[a].new_i$, the task stored in $A_t[a]$ is not an initial task. Thus, $A_t[a].old_i$ and $A_t[a].new_i$ are the parameters old_i and new_i of some `BDCAS()` operation which created the task stored in $A_t[a]$ in line 50 before t. Since $old_i \prec_a \mathbf{new}_i$ from the irreflexivity requirement, the corollary follows. □

LEMMA 3.42. *For any $t \geq 0$, $D_t \subseteq D_{t+1}$ and $S_t \subseteq S_{t+1}$.*

PROOF. Suppose that $\tau \in D_t$, and we will show that $\tau \in D_{t+1}$. Since $\tau \in D_t$, from Definition 3.27, $A_{t'}[\tau.add_1] \neq \tau$ for all $t' \leq t$, and, for some $i \in \{0, 1\}$, $B_t[\tau.add_i].val' \neq \tau.old_i$. It suffices to show that $A_{t+1}[\tau.add_1] \neq \tau$ and $B_{t+1}[\tau.add_i].val' \neq \tau.old_i$.

The process p_τ which creates τ does not return in line 49, and therefore there exists a point $t_i < t_\tau \leq t$ in which p_τ's `read`($\tau.add_i$) method returns $\tau.old_i$. By Lemma 3.15, $B_{t_i}[\tau.add_i].val = \tau.old_i$.

Since $B_t[\tau.add_i].val' \neq \tau.old_i$, from Claim 3.40 if $B[\tau.add_i].val$ changes after t then there exists a point $t'' \geq t$ such that $B_{t''}[\tau.add_i].val = B_t[\tau.add_i].val' \neq \tau.old_i$. From Claim 3.13 and Corollaries 3.19 and 3.41, $\tau.old_i = B_{t_i}[\tau.add_i].val \prec_{\tau.add_i} B_{t^*}[\tau.add_i].val$ for all $t^* \geq t''$. Therefore, $B_{t+1}[\tau.add_i].val' \neq \tau.old_i$.

Now suppose for the purpose of proving a contradiction that $A_{t+1}[\tau.add_1] = \tau$. By Claim 3.3, there is a point no later than $t + 1$ at which $A[\tau.add_0] = \tau$; let t_0 be the earliest such point. Since at most one shared memory operation takes place per point in time and $A[\tau.add_1] \neq A[\tau.add_0]$ changes at point $t+1$, $t_0 \leq t$. From Lemma 3.11, $\tau.stat \neq$ False throughout the execution. From Claim 3.5 and Lemma 3.7 and that τ is not an initial task, $\tau.stat$ does not change to True until after point $t+1$, and hence $\tau_t.stat = \bot$. Using Claim 3.6, we also have that $A_t[\tau.add_0] = \tau$. Since $A_t[A_t[\tau.add_0].add_1] = A_t[\tau.add_1] \neq \tau$, from Definition 3.26 $B_t[\tau.add_0].val' = \tau.old_0$, so it follows that $i = 1$ and $B_t[\tau.add_1].val' \neq \tau.old_1$; again from Definition 3.26, we have that $B_t[\tau.add_1].val' = A_t[\tau.add_1].new_1 \neq \tau.old_1$. This contradicts Claim 3.8.

Now suppose that $\tau \in S_t$: then, there exists some $t' \leq t$ such that $A_{t'}[\tau.add_1] = \tau$. Since $t' \leq t < t + 1$, it also follows that $\tau \in S_{t+1}$. □

CLAIM 3.43. *Suppose process p creates task τ (in line 50) at some point t_τ. Let $t_{read} < t_\tau$ be the linearization point of p's `read`(a_1) operation in line 49 during the same iteration of the while-loop. Further, let $t > t_\tau$.*

(a) *If the value of $A[\tau.add_1]$ changes during $[t_{read}, t]$, then $\tau \in S_t \cup D_t$.*

(b) *If there is a task τ^* with $\tau^*.add_1 = \tau.add_1$, such that $\tau^*.stat$ changes from $\bot$ to True in $[t_{read}, t]$, then $\tau \in S_{t'} \cup D_{t'}$ for some $t' < t$.*

PROOF. Let $a = \tau.add_1$. Consider the while-loop iteration in which p creates task τ in line 50 (at point t_τ). Then p's `read`(a) in line 49 of the same while-loop iteration returns $\tau.old_1$, as otherwise p would complete its `BDCAS()` call in that line. Hence, by Lemma 3.15, $B_{t_{read}}[a].val = \tau.old_1$.

Let κ be the task stored in $A[a]$ at point t_{read}. Then by the definition of the interpreted value of $B[a]$,

$$\tau.old_1 = \begin{cases} \kappa.new_1 & \text{if } \kappa.stat = \text{True at point } t_{read}, \textit{and} \\ \kappa.old_1 & \text{otherwise.} \end{cases} \tag{23}$$

To prove (a), we assume that

$$A_z[a] \neq \tau \text{ for all } z \leq t, \tag{24}$$

because otherwise $\tau \in S_t$, and (a) follows immediately. Now, suppose that the value of $A[\tau.add_1]$ changes during $[t_{read}, t]$. Let $\tau' = A_t[a]$. By Claim 3.9 (b), $\kappa \neq \tau'$. Then by Claim 3.9 (a) $\kappa.new_1 \prec_a \tau'.new_1$. Since either $\kappa.old_1 \prec_a \kappa.new_1$ or $\kappa.old_1 = \kappa.new_1$ (the latter is the case if $\kappa = \lambda_a$), we obtain from (23) and transitivity of $\prec_a$ that $\tau.old_1 \prec_a \tau'.new_1$. Hence, by irreflexivity $\tau.old_1 \neq \tau'.new_1$. By (24) and since $A_t[a_1] = \tau'$, it follows that $\tau \in D_t$ since $B_t[a_1].val' = \tau'.new_1 \neq \tau.old_1$.

To prove (b), assume that there is a task τ^* with $\tau^*.add_1 = \tau.add_1$, such that $\tau^*.stat$ changes from $\bot$ to True at point $t^* \in [t_{read}, t]$. By Lemma 3.7, $A_{t^*}[a] = \tau^*$. Since at most one shared memory operation takes place at point t, it follows that $A_{t^*-1}[a] = \tau^*$.

We assume that $\tau^* \neq \tau$, because otherwise $\tau \in S_{t^*-1}$, in which case (b) follows immediately. Also, if the value of $A[\tau.add_1]$ changes during $[t_{read}, t-1]$, then by (a) $\tau \in S_{t-1} \cup D_{t-1}$. So, suppose that the value of $A[\tau.add_1]$ does not change during $[t_{read}, t-1]$. Hence, $A[a] = \tau^* = \kappa$ throughout $[t_{read}, t-1]$.

Again using that at most one shared memory operation takes place at point t, it follows that $\kappa.stat$ changes to True after t_{read}. Then by Claim 3.5 $\kappa.stat \neq$ True at point t_{read}, so by (23), $\tau.old_1 = \kappa.old_1$. Moreover, then κ is not an initial task, and so $\kappa.old_1 \neq \kappa.new_1$, and so $\tau.old_1 \neq \kappa.new_1$. Since $t_\tau \in [t_{read}, t-1]$, $A_{t_\tau}[a] = \kappa$, and it follows that $B_{t_\tau}[a].val' = \kappa.new_1 \neq old_1$. Hence, $\tau \in S_{t_\tau} \cup D_{t_\tau}$ which completes the proof of (b). ◻

Lemma 3.44. *Suppose that at point $t_{p@59}$, process p reads $A[a] = \tau$ in line 59 of operation* `finish(a)`*, and at some point $t > t_{p@59}$ during the same* `finish()` *call one of the following happens:*

(a) p executes a successful `CAS()` *operation in line 62,*
(b) p executes a failed `CAS()` *operation in line 64, or*
(c) p executes line 67.

Then $\tau \in D_{t'} \cup S_{t'}$ for some $t' < t$.

Proof. If $\tau = \lambda_a$, then $\tau \in S_{t'}$ for all $t' \geq 0$. Hence, assume that $\tau \neq \lambda_a$. By Claim 3.1 (b), task τ is created in line 50 at some point $t_\tau < t_{p@59}$.

First assume that (a) is true, i.e., at point t process p performs a successful $\tau_1.stat$.`CAS(⊥, True)` in line 62, where τ_1 is the task that p reads from $A[\tau.add_1]$ in line 61. Then $\tau_1.add_1 = \tau.add_1$, and so by Claim 3.43 (b) $\tau \in S_{t'} \cup D_{t'}$ for some $t' < t$.

Now assume that (b) is true. Then at point t process p executes a failed $A[\tau.a_1]$.`CAS(`τ_1, τ`)` in line 64, where τ_1 is the task p previously read from $A[\tau.a_1]$ in line 61 at some point $t_{61} < t$. Since the CAS fails, the value of $A[\tau.add_1]$ changes at some point $t' \in [t_{61}, t)$ from τ_1 to a different value. Thus, by Claim 3.43 (a) $\tau \in S_{t'} \cup D_{t'}$.

Finally, assume that (c) is true. Then at point t process p executes a $\tau.stat$.`CAS(⊥, False)` in line 67. If prior to that p executes a failed CAS in line 64 during the same `finish()` call, then the claim follows from part (b). If p executes a successful CAS in line 64 at point $t_{p@64}$, then as a result of that $A_{t_{p@64}}[\tau.add_1] = \tau$, and so $\tau \in S'_t$ for $t' = t_{p@64} < t$.

Hence, assume that p does not execute line 64. Then p evaluates the if-condition in line 63 to False. Hence, $\tau_1.new_1 \neq \tau.old_1$, where τ_1 is the task stored in $A[\tau.add_1]$ when p reads that register in line 61 at point $t_{p@61}$. Thus, for $t' = t_{p@61} < t$, if $A[\tau.add_1] \neq \tau$ throughout $[0, t']$ then $\tau \in D_{t'}$ because $B_{t'}[\tau.add_1].val' = \tau_1.new_1 \neq \tau.old_1$, and otherwise $\tau \in S_{t'}$. ◻

Lemma 3.45. *If a process p creates task τ during a* `BDCAS(`$\langle a_0, v_0, v'_0 \rangle, \langle a_1, v_1, v'_1 \rangle$`)` *operation, and the operation returns before point t, then $\tau \in D_t \cup S_t$.*

Proof. Since p's `BDCAS(`$\langle a_0, v_0, v'_0 \rangle, \langle a_1, v_1, v'_1 \rangle$`)` operation returns, there is an index i such that p's last `read(`a_i`)` operation in line 49 returns a value different from v_i. Let t^* be the linearization point of that `read(`a_i`)` operation. Then by Lemma 3.15, $B_{t^*}[a_i].val \neq v_i$.

Now consider the iteration of the while-loop in which p creates task τ. Then $\tau.old_i = v_i$ and $\tau.add_i = a_i$ due to the arguments used in the `newTask()` call in line 50. Let $t_{read} < t^*$ be the point when p's `read(`a_i`)` in line 49 of that iteration linearizes. Since p's `BDCAS()` call does not return in that while-loop iteration $B_{t_{read}}[a_i].val = v_i$. Hence,

$$\tau.old_i = B_{t_{read}}[\tau.add_i].val \neq B_{t^*}[\tau.add_i].val. \tag{25}$$

First assume that $i = 1$. By (25) and Lemma 3.24 a successful BDCAS() operation op that uses an argument triple $\langle \tau.add_1, u_1, u_1' \rangle$ for some values u_1, u_1', linearizes at some point $lin(op) \in (t_{read}, t^*]$. By the definition of linearization points and by Claim 3.23, the process that executes op, creates a task κ whose status changes to True at point $lin(op)$. Then $\kappa.add_1 = \tau.add_1$. Since $lin(op) \in (t_{read}, t^*] \subseteq (t_{read}, t]$ it follows from Claim 3.43 (b) that $\tau \in S_t \cup D_t$.

Now assume that $i = 0$. If $A_{t'}[\tau.add_0] \neq \tau$ for all $t' \leq t^*$, then $\tau \in D_{t^*} \subseteq D_t$, since either $B_{t^*}[\tau.add_0].val' = B_{t^*}[\tau.add_0].val \neq \tau.old_0$, or $\tau.old_0 = B_{t_{read}}[\tau.add_i].val \prec_{\tau.add_0} B_{t^*}[\tau.add_0].val \prec_{\tau.add_0} B_{t^*}[\tau.add_0].val'$ by Corollaries 3.19 and 3.41, and by irreflexivity $\tau.old_0 \neq B_{t^*}[\tau.add_0].val'$. Hence, assume

$$\exists t' \leq t : A_{t'}[\tau.add_0] = \tau. \tag{26}$$

First consider the case that $A_{t^*}[\tau.add_0] = \tau$. Then by the definition of interpreted value, $B_{t^*}[\tau.add_0].val \in \{\tau.old_0, \tau.new_0\}$. Thus, by (25), $B_{t^*}[\tau.add_0].val = \tau.new_0$. Then by the definition of interpreted value, $\tau.stat$ = True at point t^*. Since $\tau.stat$ = False when τ is being created, which is before t^*, it follows from Claim 3.43 (b) that $\tau \in S_{t^*} \cup D_{t^*} \subseteq S_t \cup D_t$.

Finally, consider the case that $A_{t^*}[\tau.add_0] \neq \tau$. By (26), before point t^* the value of $A[\tau.add_0]$ changes from τ to a different value, say τ'. This can only happen when some process executes a successful $A[\tau.add_0]$.CAS(τ, τ') in line 56. Then that process previously obtained τ from a `finish()` call in line 48, and hence, executed line 67 after reading τ from A in line 59. Then it follows from Lemma 3.44 (c) that $\tau \in S_t \cup D_t$. □

Claim 3.46. *Let $a \in M_0$, and t a point in time such that $\tau = L_t[a_0]$ is a task. Then at all points $t' \geq t$, $B_{t'}[a_0].val = \tau.old_0$ or $\tau.old_0 \prec_a B_{t'}[a_0].val$.*

Proof. Since $\tau = L_t[a]$ is a task, some process p calls propose(τ) on $L[a]$ in line 51 prior to t. (According to the RC specification, initially $L[a] = \bot$.) Due to the if-statement in line 49 and the arguments of p's newTask() call in line 50, p's preceding read(a) in line 49 returns $\tau.old_0$. Hence, by Lemma 3.15, there is a point $t^* < t$ (namely the linearization point of p's read(a)), such that $B_{t^*}[a].val = \tau.old_0$. Thus, it follows immediately from Corollary 3.19 that $B_{t'}[a_0].val = \tau.old_0$ or $\tau.old_0 \prec_a B_{t'}[a_0].val$ for any $t' \in [t^*, \infty) \supseteq [t, \infty)$. □

Claim 3.47. *Let τ be a task, $a = \tau.add_0$, and t a point in time such that $\tau.old_0 \prec_a B_t[a]$. Then either $A[a] = \tau$ at point t or $A[a] \neq \tau$ throughout $[t, \infty)$.*

Proof. For the purpose of contradiction, assume that $A_t[a] \neq \tau$ but there is a point $t' > t$ at which the value of $A_t[a]$ changes to τ. Then by Lemma 3.18 (b) $\tau.stat = \bot$ at point t', and thus $B_{t'}[a].val = \tau.old_0$. Thus, we have $B_{t'}[a].val \prec_a B_t[a].val$, which contradicts Corollary 3.19, since $\prec_a$ is transitive and irreflexive. □

Claim 3.48. *Suppose process p calls* BDCAS$(\langle a_0, v_0, v_0' \rangle, \langle a_1, v_1, v_1' \rangle)$, *and in one while-loop iteration p first reads a task τ from $L[a_0]$ in line 53 at point t, and then it evaluates the if-condition in line 55 to* False. *Then $\tau.old_0 \prec_{a_0} B_t[a_0].val$.*

Proof. Let τ_0 be the value that p's `finish`(a_0) call in line 48 returns at point $t_{p@48}$. Then by Claim 3.17

$$\exists s \in \{\text{False}, \text{True}\} : \tau_0.stat = s \text{ throughout } [t_{p@48}, \infty). \tag{27}$$

Since in line 54 process p also reads τ_0 from $A[a_0]$ at point $t_{p@54}$ (because it proceeds to line 55), it follows from Lemma 3.16 that

$$A[a_0] = \tau_0 \text{ throughout } [t_{p@48}, t_{p@54}]. \tag{28}$$

Thus, by (27) p's read(a_0) in line 49 returns $\tau_0.old_0$ if s = False and $\tau_0.new_0$ if s = True. Since p's BDCAS$(\langle a_0, v_0, v_0' \rangle, \langle a_1, v_1, v_1' \rangle)$ call does not return in line 49, the read(a_0) operation returns v_0. Hence, by (27) and (28)

$$B[a_0].val = v_0 \text{ throughout } [t_{p@48}, t_{p@54}]. \tag{29}$$

Now consider the task τ that p reads from $L[a_0]$ in line 53 at point $t_{p@53} \in [t_{p@48}, t_{p@54}]$. Since p evaluates the if-condition in line 55 to False, $\tau.old_0 \neq v_0$. Thus, by (29) and Claim 3.46, $\tau.old_0 \prec_{a_0} B_t[a_0].val$. ◻

Claim 3.49. *Let $a \in M_0$.*

(a) If at point t the value of $A[a]$ changes from τ_0 to $\tau \neq \tau_0$, then $L_t[a] = \tau$.
(b) Suppose at point t some process executes a successful $L[a]$.unlock(τ) operation. Then
  (b1) either at point t it holds $A[a] = \tau$ and $\tau.stat \neq \bot$, or $A[a] \neq \tau$ throughout the entire execution, and
  (b2) if $\tau \neq \bot$ then $\tau \in S_{t'} \cup D_{t'}$ for some $t' < t$.

Proof. Let $T_0 = 0$ be the beginning of the execution E and for each integer $\ell \geq 1$ let T_ℓ be the point of the ℓ-th shared memory operation in E. We will show by induction on ℓ that the claim is true provided $t \in [0, T_\ell)$. This is trivially true for $\ell = 0$.

Hence, assume we have proved that the claim is true for $t \in [0, T_\ell)$. Let $t \in [T_\ell, T_{\ell+1})$. If $t \in (T_\ell, T_{\ell+1})$, then no shared memory operation occurs at point t, and (a) and (b) are trivially true. Hence, assume $t = T_\ell$.

Part (a): Suppose at point $t = T_\ell$ the value of $A[a]$ changes from τ_0 to $\tau \neq \tau_0$. Then some process p executes a successful $A[a]$.CAS(τ_0, τ) operation in line 56 at point t. Prior to that, at point $t_{p@53} < t$ process p reads τ from $L[a]$ in line 53. Thus, $L_{t_{p@53}}[a] = \tau$, so there must be a successful choose&lock() call that linearizes before $t_{p@53} < t$ and which decides τ. (By the if-condition in line 55, $\tau \neq \bot$, so τ is not the initial value of $L[a]$, which is $\bot$ according to the RC specification.)

For the purpose of proving a contradiction assume $L_t[a] \neq \tau$. Then some successful unlock(τ) call linearizes at a point $t_u < t = T_\ell$. By the assumption that the claim is true for all $t < T_\ell$, we conclude from statement (b1) that either $A[a] = \tau$ at point t_u or $A[a] \neq \tau$ throughout the entire execution. Since at point $t > t_u$ the value of $A[a]$ changes from τ_0 to τ, the latter cannot be the case, so $A_{t_u}[a] = \tau$. But then $A_{t_u}[a] = A_t[a] = \tau$, and so by Lemma 3.16 $A[a] = \tau$ throughout $[t_u, t)$. This contradicts that at point t the value of $A[a]$ changes from $\tau_0 \neq \tau$ to τ. Thus, (a) is true for $t = T_\ell$ and thus also for $t \in [0, T_{\ell+1})$.

Part (b): Suppose that at point $t = T_\ell$ some process p executes a successful $L[a]$.unlock(τ) operation. If $\tau = \bot$, then by Claim 3.1 $A[a] \neq \tau$ throughout the entire execution, so (b) is true. Hence, assume $\tau \neq \bot$. Let $t_{p@53}$ be the point when p reads $L[a]$ in line 53 prior to its successful $L[a]$.unlock(τ). Then $L[a] = \tau$ at that point. Since p's unlock(τ) at point t is successful, and (from the sequential specification of RC) $L[a]$ is ABA-free,

$$L[a] = \tau \text{ throughout } [t_{p@53}, t]. \tag{30}$$

Now let t_f be the point when p reads $A[a]$ in line 59 during the finish() call p executes in line 48, and let $t_{p@54}$ be the point when it reads $A[a]$ again in line 54. Since p evaluates the if-statement in line 54 to True, it reads the same value τ_0 from $A[a]$ at points t_f and $t_{p@54}$. Then by Lemma 3.16, $A[a] = \tau_0$ throughout the entire interval $[t_f, t_{p@54}]$, and in particular

$$A_{t_{p@53}} = \tau_0. \tag{31}$$

Since we assume that the claim is true for all $t < T_\ell$ and already proved that (a) is true for $t = T_\ell$, it follows from part (a) of the claim and (30) that if the value of $A[a]$ changes in the interval $[t_{p@53}, t]$, then it changes to τ. Thus,

$$A[a] \in \{\tau_0, \tau\} \text{ throughout } [t_f, t]. \tag{32}$$

First assume that p evaluates the if-condition in line 55 to True. Then at some point $t_{p@56}$ process p executes $A[a]$.CAS(τ_0, τ) in line 56. Hence, by (32) $A[a] = \tau$ throughout $[t_{p@56}, t]$. Moreover, during its finish(a) call in line 57 process p reads τ from $A_t[a]$ in line 59, and executes $\tau.stat$.CAS($\bot$, False) in line 67. Hence, $\tau.stat \neq \bot$ at point t by Claim 3.5, and since $A[a] = \tau$ at point t (b1) is true. Moreover, (b2) follows from Lemma 3.44 (c).

Now assume that p evaluates the if-condition in line 55 to False. Then by Claim 3.48, $\tau.old_0 \prec_a B_{t_{p@53}}[a].val$. Hence, by (31) and Claim 3.47, $A[a] \neq \tau$ throughout $[t_{p@53}, \infty)$. We will now show that

$$A[a] \neq \tau \text{ throughout } [0, t_{p@53}].$$

Then clearly (b1) is true. Moreover, $p \in D_{t_{p@53}}$ by Corollary 3.41, and so (b2) is true.

For the purpose of proving a contradiction, assume there is a point in $[0, t_{p@53})$ at which $A[a] = \tau$. Since $L_{t_{p@53}}[a] = \tau$, some process proposes τ in line 51, so τ is not the initial task λ_a. Hence, at some point $t' < t_{p@53} < t = T_\ell$ the value of $A[a]$ changes to τ. Then by part (a) of the claim (which we proved true for all $t < T_\ell$), we have $L_{t'}[a] = \tau \neq \perp$. By (30) and since $L[a]$ is ABA-free,

$$L[a] = \tau \text{ throughout } [t', t]. \tag{33}$$

Since $A_{t'}[a] = \tau$, it follows from (32) that at some point $t'' \in [t', t_f) \subseteq [t', t]$ the value of $A[a]$ changes to τ_0 from a different value. By part (a) of the claim $L[a] = \tau_0 \neq \tau$ at point t'', which contradicts (33). ◻

Lemma 3.50. *Let $a \in [m]$, t a point in time, and $\tau \neq \lambda_a$ a task such that $A_t[a] = \tau$.*

(a) If $a \in M_0$, then there is a point $t' < t$ such that $L_{t'}[a] = \tau$.
(b) If $a \in M_1$, then there is a point $t' < t$ such that $A_{t'}[\tau.add_0] = \tau$.

Proof. Part (a) follows immediately from Claim 3.49 (a) and part (b) from Claim 3.3. ◻

Lemma 3.51. *Let $a \in M_0$, let τ be a task, and let $t_1 < t_2$ be two points in time such that $L_{t_1}[a] = \tau \neq L_{t_2}[a]$. Then there exists a point $t < t_2$ such that $\tau \in D_t \cup S_t$.*

Proof. Since τ is a task, $\tau \neq \perp$. As $L_{t_1}[a] = \tau \neq L_{t_2}[a]$, at some point $t_{p@58} \in [t_1, t_2]$ some process p executes a successful $L[a]$.`unlock`(τ) operation in line 58. Hence, by Claim 3.49 (b) there exists $t < t_{p@58} \leq t_2$ such that $\tau \in D_t \cup S_t$. ◻

Claim 3.52. *Let τ be a task that is created at t_τ, and suppose that, for some point $t \geq t_\tau$, there is an index $i \in \{0, 1\}$ such that $B_t[\tau.add_i] \neq \tau.old_i$. Then $B_{t'}[\tau.add_i].val \neq \tau.old_i$ for all $t' \geq t$.*

Proof. Let p be the process creating τ at point t_τ, i.e., p's `newTask()` in line 50 returns τ at that point. Then before that, in line 49 process p executes a `read`$(\tau.add_i)$ operation for each $i \in \{0, 1\}$. Since p does not complete its `BDCAS()` call in this line, that `read`$(\tau.add_i)$ operation returns $\tau.old_i$. Hence, by Lemma 3.15, the interpreted value of $B[\tau.add_i]$ is $\tau.old_i$ at the linearization point of that `read`$(\tau.add_i)$, and thus at some point before t_τ.

Assume that $B_t[\tau.add_i].val \neq \tau.old_i$ for some $i \in \{0, 1\}$. Let $a = \tau.add_i$. By Claim 3.13 and Corollary 3.19, $\tau.old_i \prec_a B_t[\tau.add_i].val$. Now the claim follows by irreflexivity and transitivity of $\prec_a$. ◻

Claim 3.53. *Let $t_1 < t_2$. Suppose that process p calls `BDCAS`$(\langle a_0, v_0, v_0' \rangle, \langle a_1, v_1, v_1' \rangle)$, and during that call it creates a task τ, such that $\tau \in D_{t_1} \cup S_{t_1}$, and during $[t_1, t_2]$ process p completes a `finish`(a_0) call in line 48. Then there is an index $i \in \{0, 1\}$ such if p calls `read`(a_i) in line 49 after point t_2, then that `read()` operation returns a value different from v_i.*

Proof. Since τ is not an initial task, process p creates τ at some point $t_\tau \leq t_1$, and for each $i \in \{0, 1\}$

$$\tau.old_1 = v_1 \neq v_1' = \tau.new_1. \tag{34}$$

Moreover, since p's `BDCAS()` does not complete in its execution of line 49, it follows that for each $i \in \{0, 1\}$, there is a point $t_{p@49,i} < t_\tau$ in which $B[a_i].val = v_i$.

First assume that either $\tau \in S_{t_1}$, or $\tau \in D_{t_1}$ because there is a task τ' (in case $\tau \in S_{t_1}$ we have $\tau = \tau'$) such that $A_{t_1}[a_1] = \tau'$, and $B[a_1].val' = \tau'.new_1 \neq \tau.old_1 = v_1$. Consider the first process p' to execute line 62, 66,

or 71 after τ' is first placed into $A[a_1]$, and call this point t^*. From Lemma 3.7 and that $\tau'.stat$ is initialized to $\bot$, $B[a_1]$ changes from $\tau'.old_1$ to $\tau'.new_1$ at point t^*, so $t^* > t_{p@49,i}$. Consider p's `read`(a_1) call after point t_2. At some point $t_{p@70}$ process p reads some task τ'' from $A[a_1]$ in line 70, and then at point $t_{p@71}$ it performs $\tau''.stat$.CAS($\bot$, True) in line 71. Thus, $t_{p@71} \geq t^*$. Since $B[a_1].val$ was v_1 at point $t_{p@49,i}$ and $\tau'.new_1$ at point $t^* \leq t_{p@71}$, the claim follows from Claim 3.13.

Now assume that $\tau \in D_{t_1}$ because $B_{t_1}[a_0].val' \neq \tau.old_0 = v_0$.

Consider the case where $B_{t_1}[a_0].val' = B_{t_1}[a_0].val \neq \tau.old_0$; since $t_\tau \leq t_1$, it follows from Claim 3.52 that $B_{t'}[a_0].val \neq \tau.old_0$ for all $t' > t_1$, and from Claim 3.40 that $B_{t'}[a_0].val' \neq \tau.old_0$ for all $t' > t_1$. (Note that Definition 3.26 and Corollary 3.19 implies that if $B_{t'}[a_0].val \neq B_{t'}[a_0].val'$, then $B_{t'}[a_0].val \prec_{t'} B_{t'}[a_0].val'$). Hence, by Lemma 3.15, the `read`(a_0) that p calls at point t_2 returns a value different from $\tau.old_0 = v_0$.

The remaining case is when $B_{t_1}[a_0].val' \neq \tau.old_0$ and $B_{t_1}[a_0].val' \neq B_{t_1}[a_0].val$. In this case, there is a non-initial task τ' such that $A_{t_1}[\tau'.add_1] = \tau'$, $\tau'.add_0 = \tau.add_0 = a_0$, and $\tau'.old_0 = \tau.old_0 = v_0$. By Claim 3.3,

$$\text{there is a point } t_0 \leq t_1 \text{ at which } A[\tau'.add_1] = A[a_0] = \tau'. \tag{35}$$

By the claim's assumptions, p completes a `finish`(a_0) call in $[t_1, t_2]$. Let $t_f \geq t_1$ be the point when the `finish`(a_0) call returns. If the `finish()` call returns $\tau'' \neq \tau'$, then p reads τ'' from $A[a_0]$ in line 59 of that `finish()` call. Hence, by Lemma 3.18 (b) and Claim 3.5 $\tau'.stat \neq \bot$ at point t_f. If, on the other hand, the `finish()` call returns τ', then by Claim 3.17 $\tau'.stat \neq \bot$ at point t_f. In either case, by Lemma 3.11 $\tau'.stat =$ True when p's `finish()` call terminates at point t_f. Since by Claim 3.5 $\tau'.stat$ does not change once it is True, and by Lemma 3.7 it changes to True while $A[a_0] = \tau'$, by (35) there is a point $t^* \in [t_0, t_f]$ at which $A[a_0] = \tau'$ and $\tau'.stat =$ True. Hence, $v_0 = \tau'.old_0 \prec_{a_0} \tau'.new_0 = B_{t^*}[a_0].val$ (because τ' is a non-initial task). Thus, by Claim 3.13 and transitivity and irreflexivity of $\prec_a$, $B_{t'}[a_0].val \neq v_0$ for all $t' > t^*$. Hence, by Lemma 3.15, the `read`(a_0) that p calls at point t_2 returns a value different from v_0. □

Lemma 3.54. *If process p proposes task τ during a* `BDCAS()` *operation and $\tau \in D_t \cup S_t$, then p executes line 51 at most once after t and before the* `BDCAS()` *returns.*

Proof. Suppose for the purpose of proving a contradiction that p executes line 51 twice after t and before its `BDCAS()` call returns, at points t_1 and t_2. From the while-loop of `BDCAS()`, it follows that p executes lines 48 and 49 in (t_1, t_2). Since $\tau \in D_t \cup S_t \subseteq D_{t_1} \cup S_{t_1}$ (by Lemma 3.42), using Claim 3.53 it follows that for some $i \in \{0, 1\}$, p's `read`(a_i) operation in line 49 returns a value different from its argument old_i. Thus, p returns in line 49, contradicting that it executes line 51 at point t_2 before its `BDCAS()` call returns. □

Lemma 3.55. *Let $a \in M_0$ and let $t_0 < t_1 < t_2 < t_3$ be four points in time, such that at each point t_i, $i \in \{0, \ldots, 3\}$, process p executes an $L[a]$*.`choose&lock()` *call in line 52. Then some successful $L[a]$*.`unlock()` *call and some successful $L[a]$*.`choose&lock()` *call linearize in $(t_0, t_3]$.*

Proof. Suppose the claim is not true, i.e., either no successful $L[a]$.`choose&lock()` linearizes in $(t_0, t_3]$ or no successful $L[a]$.`unlock()` call linearizes in (t_0, t_3). Since p calls `choose&lock()` at point t_0, $L[a]$ is locked at t_0. If any successful $L[a]$.`unlock()` call linearizes in (t_0, t_3), then a successful $L[a]$.`choose&lock()` linearizes no later than p's next `choose&lock()` call in the interval, contradicting both cases. Therefore, no successful $L[a]$.`unlock()` call linearizes in (t_0, t_3), and $L[a]$ is locked throughout $(t_0, t_3]$, so there is a value τ such that $L[a] = \tau$ throughout $(t_0, t_3]$. By Claim 3.49, if the value of $A[a]$ changes in the interval $(t_1, t_3]$, then it changes from $\tau' \neq \tau$ to τ. In particular, the value of $A[a]$ changes at most once in the interval $(t_0, t_3]$. Process p executes at least two complete iterations of the while-loop in the interval $(t_0, t_3]$, and thus throughout at least one complete iteration $A[a]$ remains unchanged. Hence, let τ_0 be a task such that $A[a] = \tau_0$ throughout one of the two iterations. Then p's `finish()` call in line 48 returns τ_0 because p reads that value from $A[a]$ in line 59. Moreover, p reads τ_0

from $A[a]$ in line 54, and so the if-statement in that line evaluates to True. In line 53 process p reads τ from $L[a]$, and so in line 58 p calls $L[a]$.unlock(τ). Clearly, this unlock() call is successful, which is a contradiction. □

LEMMA 3.56. *Let $a \in M_0$ and let $s_1 < s_2$ be two points in which some successful $L[a]$.unlock() calls linearize. Then the number of method calls to $L[a]$ in the interval (s_1, s_2) is no more than $3n$ for each of $L[a]$.propose(), $L[a]$.choose&lock(), $L[a]$.unlock(), and no more than $4n$ for $L[a]$.read().*

PROOF. It suffices to show that if any process executes four invocations of $L[a]$.propose(), $L[a]$.choose&lock(), or $L[a]$.unlock() in a given interval (or five invocations of $L[a]$.read()), then some successful $L[a]$.unlock() operation linearizes in that interval.

The case of $L[a]$.choose&lock() follows immediately from Lemma 3.55. Let o be one of operations $L[a]$.propose() or $L[a]$.unlock(), and suppose for the purpose of proving a contradiction that a process p executes o at points o_0, o_1, o_2, o_3 but no successful $L[a]$.*unlock* operation linearizes in the interval $(o_0, o_3]$. Because p can only execute o at points lines 51 and 58, respectively, it follows that p executes line 52 at points t_0, t_1, t_2 such that $o_0 < t_0 < o_1 < t_1 < o_2 < t_2 < o_3$. Using similar arguments to Lemma 3.55, $L[a]$ is locked throughout $[t_0, o_3]$, and p executes at least two complete iterations of the while-loop in the interval $[t_0, o_3]$. Following the proof of Lemma 3.55, p executes a successful $L[a]$.unlock() operation by the end of the two complete while-loop iterations, which is a contradiction. The case of $L[a]$.read() is nearly identical, except that we need five $L[a]$.read() operations to guarantee that p executes at least two complete while-loop iterations while $L[a]$ is locked. This is because t_0 is in a different while-loop iteration than o_0 for this case. □

CLAIM 3.57. *Let $[t_1, t_2]$ be a time interval during which only process p takes steps, and only executes a single BDCAS($\langle a_0, v_0, v'_0 \rangle$, $\langle a_1, v_1, v'_1 \rangle$) call. If p completes k full iterations of the while-loop during $[t_1, t_2]$, then in each complete iteration it executes line 58, and the $L[a_0]$.unlock() call in that line is successful. Moreover, p's choose&lock() calls in line 52 during iterations $2, \ldots, k$ are successful.*

PROOF. Consider any complete iteration of the while-loop during interval $[t_1, t_2]$. Let τ be the task that p's finish() call in line 48 returns. Then p reads τ from $A[a_0]$ in line 59 during the corresponding finish() call. Since $A[a_0]$ does not change until p reads $A[a]$ again in line 54, the if-condition in that line evaluates to True. Hence, in each of the k iterations of the while-loop, process p executes lines 55–58. In particular, when p calls $L[a_0]$.unlock(τ') in line 58, it uses the argument τ' that it read from $L[a_0]$ in line 53. Hence, each such unlock() call succeeds.

Thus, whenever p calls choose&lock() in line 52 during one of iterations $2, \ldots, k$, $L[a_0]$ is unlocked. Hence, each such choose&lock() call is successful. □

CLAIM 3.58. *Suppose that after point t only process p takes steps, and only executes a single BDCAS($\langle a_0, v_0, v'_0 \rangle$, $\langle a_1, v_1, v'_1 \rangle$) call, until the first point $t^* > t$ such that $L_{t^*}[v_0]$.val $= \tau$, where τ is one of the tasks p proposes in line 51. The expected number of shared memory steps in $[t, t^*]$ is at most $O(\log^2 n \log\log n)$.*

PROOF. For $i \geq 1$, let t_i be the point when p starts its i-th complete while-loop iteration.

By Claim 3.57, during each of the iterations process p executes a successful $L[a_0]$.unlock() call in line 58. Then the RC methods invoked by p in each while-loop iteration correspond to the algorithm in Figure 6, where $R = L[a_0]$. Precisely, lines 42–46 correspond to lines 50–53 and 58, respectively, and the while-loop condition in line 41 is false until just after t^*. The claim follows by applying Lemma 2.54. □

LEMMA 3.59 (PROBABILISTIC OBSTRUCTION FREEDOM). *If process p invokes some method call BDCAS($\langle a_0, v_0, v'_0 \rangle$, $\langle a_1, v_1, v'_1 \rangle$), and at some point during the operation, p starts to run solo, then the operation returns after at most $O(\log^2 n \log\log n)$ shared memory steps in expectation.*

PROOF. By applying Claim 3.58 twice, there is a point t and a point $t' > t$ such that

- t' is at most $O(\log^2 n \log\log n)$ expected steps after p starts running solo, and
- $L_t[add_0] = \tau$ and $L_{t'}[a_0] = \tau'$ for two distinct tasks τ and τ' proposed by p.

Thus, by Lemma 3.51 $\tau \in D_{t'} \cup S_{t'}$. Hence, after point t process p completes at most two more iterations of the while-loop according to Lemma 3.54, which by Lemma 2.47 adds at most $O(\log n \log\log n)$ steps in expectation. ◻

Lemma 3.60. *Let $a \in M_0$, let τ be a task, and suppose at point t some process is poised to call $L[a]$.*`unlock`*(τ) in line 58. Then $\tau \in D_{t'} \cup S_{t'}$ for some $t' < t$.*

Proof. Consider the execution prefix that ends at point t. If $L_t[a] = \tau$ and $L[a]$ is locked at point t, then we can let p run solo until its $L[a]$.`unlock`(a) completes. Hence, by Claim 3.49 (b2) $\tau \in D_{t'} \cup S_{t'}$. Then this is obviously also true for any other execution that has the same prefix up to point t.

Now suppose that either $L_t[a] \neq \tau$ or $L[a]$ is unlocked. Since p reads τ from $L[a]$ in line 53 prior to t, there is a point $t^* < t$ at which a successful $L[a]$.`unlock`(τ) call linearizes. For that case we have already proved that there exists $t' < t^* < t$ such that $\tau \in D_{t'} \cup S_{t'}$. ◻

4 CONCLUSION

We presented a new and more efficient implementation of the repeated choice object, originally introduced by Giakkoupis, Giv, and Woelfel [9]. Using this improved building block in the DCAS algorithm of the same paper improves the algorithm's expected amortized step complexity from $O(\log n)$ to $O(\log\log n)$. The improved complexity of the RC object is obtained by adding a binary search before the linear search, and distributing the work for clearing parts of an array over multiple processes when needed, and otherwise only partially clearing the array. While this sounds simple, it turned out to be quite challenging. Both the new RC algorithm and its analysis are highly involved. As the randomized properties of the new RC object are different from the previous one, we also provided a new analysis of the BDCAS building block used in the DCAS algorithm.

We believe that the RC object may have other applications. A natural open question is whether a similar primitive with constant step complexity exists.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. Support is gratefully acknowledged from the Agence Nationale de la Recherche (ANR) under project ByBloS (ANR-20-CE25-0002), the Natural Sciences and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN/2019-04852, the Canada Graduate Scholarships-Doctoral program (CGSD-578918-2023/198), and the Canada Research Chairs program, as well as the University of Calgary's Eyes High Doctoral Recruitment Scholarship program.

REFERENCES

[1] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. 1997. Disentangling multi-object operations. In *Proceedings of the 16th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 111–120. https://doi.org/10.1145/259380.259431

[2] Ole Agesen, David Detlefs, Christine Flood, Alexander Garthwaite, Paul Martin, Mark Moir, Nir Shavit, and Guy Steele. 2002. DCAS-based concurrent deques. *Theory of Computing Systems* 35 (June 2002), 349–386. https://doi.org/10.1007/s00224-002-1058-2

[3] Zahra Aghazadeh and Philipp Woelfel. 2016. Upper bounds for boundless tagging with bounded objects. In *Proceedings of the 30th International Symposium on Distributed Computing (DISC)*. 442–457. https://doi.org/10.1007/978-3-662-53426-7_32

[4] James H. Anderson and Mark Moir. 1995. Universal constructions for multi-object operations. In *Proceedings of the 14th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 184–193. https://doi.org/10.1145/224964.224985

[5] Hagit Attiya and Eyal Dagan. 2001. Improved implementations of binary universal operations. *Journal of the ACM* 48, 5 (Sept. 2001), 1013–1037. https://doi.org/10.1145/502102.502105

[6] Hagit Attiya and Eshcar Hillel. 2011. Highly concurrent multi-word synchronization. *Theoretical Computer Science* 412, 12-14 (2011), 1243–1262. https://doi.org/10.1016/j.tcs.2010.12.049

[7] Oksana Denysyuk and Philipp Woelfel. 2016. Are Shared Objects Composable under an Oblivious Adversary?. In *Proceedings of the 35th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 335–344. https://doi.org/10.1145/2933057.2933115

[8] Steven Feldman, Pierre LaBorde, and Damian Dechev. 2015. A wait-free multi-word compare-and-swap operation. *International Journal of Parallel Programming* 43, 4 (2015), 572–596. https://doi.org/10.1007/s10766-014-0308-7

[9] George Giakkoupis, Mehrdad Jafari Giv, and Philipp Woelfel. 2021. Efficient Randomized DCAS. In *Proceedings of the 53rd Annual ACM Symposium on Theory of Computing (STOC)*. 1–64. https://doi.org/10.1145/3406325.3451133

[10] George Giakkoupis, Mehrdad Jafari Giv, and Philipp Woelfel. 2021. *Efficient randomized DCAS.* Research Report. Inria. https://hal.inria.fr/hal-03195692

[11] Wojciech Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*. 373–382. https://doi.org/10.1145/1993636.1993687

[12] Michael Greenwald. 2002. Two-handed emulation: How to build non-blocking implementation of complex data-structures using DCAS. In *Proceedings of the 21st SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 260–269. https://doi.org/10.1145/571825.571874

[13] Michael Greenwald and David R. Cheriton. 1996. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 123–136. https://doi.org/10.1145/238721.238767

[14] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient multi-word compare and swap. In *Proceedings of the 34th International Symposium on Distributed Computing (DISC)*. 4:1–4:19. https://doi.org/10.4230/LIPIcs.DISC.2020.4

[15] Phuong Hoai Ha and Philippas Tsigas. 2004. Reactive multi-word synchronization for multiprocessors. *The Journal of Instruction-Level Parallelism* 6 (2004), 25 pages. http://www.jilp.org/vol6/v6paper3.pdf

[16] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A practical multi-word compare-and-swap operation. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC)*. 265–279. https://doi.org/10.1007/3-540-36108-1_18

[17] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems* 12, 3 (1990), 463–492.

[18] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the 13th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 151–160. https://doi.org/10.1145/197917.198079

[19] Yujie Liu and Michael F. Spear. 2012. A lock-free, array-based priority queue. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. 323–324. https://doi.org/10.1145/2145816.2145876

[20] Michael Luby. 1988. *On the parallel complexity of symmetric connection networks.* Technical Report 214/88. Department of Computer Science, University of Toronto.

[21] Victor Luchangco, Mark Moir, and Nir Shavit. 2003. Nonblocking k-compare-single-swap. In *25th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. 314–323. https://doi.org/10.1145/777412.777468

[22] Charles Martel and Ramesh Subramonian. 1994. On the Complexity of Certified Write-All Algorithms. *Journal of Algorithms* 16, 3 (1994), 361–387. https://doi.org/10.1006/jagm.1994.1017

[23] Henry Massalin and Calton Pu. 1992. A lock-free multiprocessor OS kernel. *ACM SIGOPS Operating Systems Review* 26, 2 (1992), 108. https://doi.org/10.1145/142111.993246

[24] Mark Moir. 1997. Transparent support for wait-free transactions. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG)*. 305–319. https://doi.org/10.1007/BFb0030692

[25] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *Proceedings of the 14th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*. 204–213. https://doi.org/10.1145/224964.224987

[26] Håkan Sundell. 2011. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *International Journal of Parallel Programming* 39, 6 (2011), 694–716. https://doi.org/10.1007/s10766-011-0167-4