On improving complexity of linearizable and wait-free implementations of concurrent objects by relaxing their specifications.

Adnane Khattabi Riffi

To cite this version:

Adnane Khattabi Riffi. On improving complexity of linearizable and wait-free implementations of concurrent objects by relaxing their specifications.. Computational Complexity [cs.CC]. Université de Bordeaux, 2023. English. NNT : 2023BORD0101 . tel-04147099

HAL Id: tel-04147099
https://theses.hal.science/tel-04147099

Submitted on 30 Jun 2023

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.

THÈSE PRÉSENTÉE

POUR OBTENIR LE GRADE DE

DOCTEUR
DE L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE MATHÉMATIQUES ET

INFORMATIQUE

Par **Adnane KHATTABI**

Amélioration de Complexité d'Implémentations Linéarisables et Wait-free d'Objets Concurrents en Relaxant leurs Spécifications

Sous la direction de : **Colette JOHNEN** et **Alessia MILANI**

Soutenue le 31 mars 2023

Membres du jury :

Mme. Emmanuelle ANCEAUME	Directrice de Recherche	IRISA	Rapporteure
M. Achour MOSTEFAOUI	Professeur	Université de Nantes	Rapporteur
M. Nicolas HANUSSE	Directeur de Recherche	Université de Bordeaux	Président
Mme. Janna BURMAN	Maître de conférences	Université Paris-Saclay	Examinatrice
Mme. Alessia MILANI	Professeure	Aix-Marseille Université	Directrice
Mme. Colette JOHNEN	Professeure	Université de Bordeaux	Directrice

THESIS PRESENTED

TO OBTAIN THE DEGREE OF

DOCTOR

FROM BORDEAUX UNIVERSITY

DOCTORAL SCHOOL OF MATHEMATICS AND

COMPUTER SCIENCE

Adnane Khattabi

On Improving the Complexity of Linearizable and Wait-free Implementations of Concurrent Objects by Relaxing their Specifications

Supervisors: **Colette Johnen** and **Alessia Milani**

Defended the 31st of March 2023

Members of the jury:

Emmanuelle ANCEAUME	Research Director	IRISA	Reporter
Achour MOSTEFAOUI	Professor	Nantes University	Reporter
Nicolas HANUSSE	Research Director	Bordeaux University	President
Janna BURMAN	Associate Professor	Paris-Saclay University	Examinator
Alessia MILANI	Professor	Aix-Marseille University	Supervisor
Colette JOHNEN	Professor	Bordeaux University	Supervisor

Amélioration de Complexité d'Implémentations Linéarisables et Wait-free d'Objets Concurrents en Relaxant leurs Spécifications

Résumé : Dans un contexte distribué, les différents problèmes de synchronicité entre processus sont modélisés à l'aide d'objets partagés. Lorsqu'un nouvel objet partagé est implémenté, on s'appuie souvent sur des objets de base préexistants. En cherchant à maximiser l'efficacité de ces implémentations, un nouveau domaine de recherche a émergé ces dernières années, centré sur le compromis possible entre la précision d'une implémentation et sa complexité.

Nous étudions dans cette thèse la définition d'objets partagés relaxés où les opérations ont le droit à une certaine marge d'erreur, et comment cela peut améliorer la complexité de leurs implémentations. Nous considérons le cas d'objets partagés connus : counter, max register, et FIFO queue.

Tout d'abord, nous étudions la possibilité d'améliorer la complexité des implémentations relaxées du counter et max register par rapport à leurs implémentations exactes. Dans le modèle de mémoire partagée classique, nous étudions dans quelle mesure permettre aux implémentations linéarisables et wait-free de ces objets de retourner des valeurs approximatives, plutôt que des valeurs précises, peut améliorer leur complexité.

Nous considérons le k-multiplicatif max register et le k-multiplicatif counter, où les opérations de lecture sont autorisées à se tromper d'un facteur multiplicatif de k. Nous présentons une implémentation du k-multiplicatif counter wait-free linéarisable pour $k \geq n$ avec une complexité de pas amortie constante où n est le nombre de processus. Nous montrons également qu'en bornant l'exécution, nous sommes capables d'implémenter le counter k-multiplicatif pour $k \geq \sqrt{n}$ d'une manière linéarisable wait-free avec une complexité de pas dans le pire des cas de $O(min(\log(\log(m+1)), n))$ où m représente la limite du nombre d'opérations CounterIncrement lors d'une exécution. Les deux implémentations offrent une amélioration exponentielle de la complexité de leurs équivalents exacts dans l'état de l'art.

Ensuite, nous montrons que la relaxation de la sémantique du max register en autorisant l'imprécision d'un facteur multiplicatif constant produit une amélioration exponentielle de la complexité de pas dans le pire des cas pour la variante bornée, et de la complexité de pas amortie pour la variante non bornée.

Afin de mesurer les limites de ces relaxations, nous étudions les bornes inférieures de la complexité du counter et max register k-multiplicatif. Nous obtenons le résultat que lorsque le paramètre d'approximation k ne dépend pas du nombre de processus, assouplir la sémantique du counter en autorisant l'imprécision d'un facteur multiplicatif ne peut asymptotiquement réduire la complexité des pas amortis des compteurs non bornés de plus d'un facteur logarithmique. Nous prouvons également que notre max register k-multiplicatif borné est optimal.

En ce qui concerne la FIFO queue, la conception d'implémentations efficaces wait-free est complexe malgré son utilisation dans de nombreuses applications distribuées. La plupart des implémentations des FIFO queue dans la littérature s'appuient sur des contraintes de concurrence : tous les processus ne sont pas autorisés à exécuter des opérations de Enqueue et de Dequeue.

Dans cette thèse, nous étudions la possibilité d'implémenter la FIFO queue d'une façon wait-free avec une complexité logarithmique dans le pire des cas sans contraintes de concurrence. Par conséquent, nous présentons une implémentation qui prend en charge n enqueuers et k dequeuers où la complexité dans le pire des cas d'une opération Enqueue est en $O(\log n)$ et où la complexité de l'opération Dequeue dépend du niveau de concurrence et est $O(k \log n)$ dans le pire des cas.

Nous nous appuyons ensuite sur l'assouplissement de la sémantique de la FIFO queue pour montrer que le fait d'autoriser des opérations Dequeue concurrentes à retourner le même élément engendre une implémentation avec une complexité en $O(\log n)$ dans le pire des cas

pour les opérations Enqueue et Dequeue.
Mots-clés : objets concurrents, algorithmes distribués, calcul distribué, complexité, tolérance aux pannes

---

On Improving the Complexity of Linearizable and Wait-free Implementations of Concurrent Objects by Relaxing their Specifications

Abstract: In a distributed context, the different problems of synchronicity between processes are modeled using shared objects. When a new shared object is implemented, it relies on base objects consisting of preexisting implementations, as building blocks. In seeking to maximize the efficiency of these implementations, a new research field has emerged in recent years, with a focus on the possible trade-off between the accuracy of an implementation and its complexity.

We investigate in this thesis how defining relaxed shared objects where the operations are allowed a certain margin of error can result in improved theoretical complexity results. We consider the case study of well-known shared objects, namely: the counter, max register, and FIFO queue.

First, we study the possible improvement in step complexity of the relaxed implementation of the counter and max register objects compared to their exact implementations. In the classical shared memory model, we investigate the extent to which allowing wait-free linearizable implementations of these objects to return approximate values, rather than accurate ones, may improve their step complexity.

We consider the k-multiplicative-accurate max register and the k-multiplicative-accurate counter, where read operations are allowed to err by a multiplicative factor of k. We give a wait-free linearizable k-multiplicative-accurate counter implementation for $k \geq n$ with constant amortized step complexity where n is the number of processes. We also show that by bounding the execution, we are able to implement the k-multiplicative-accurate counter for $k \geq \sqrt{n}$ in a wait-free linearizable manner and with a worst-case step complexity of $O(min(\log(\log(m+1)), n))$ where m represents the bound on the number of CounterIncrement operations during an execution. Both implementations offer an exponential improvement on the complexities of their exact counterparts in the state of the art.

Then, we show that relaxing the semantics of max registers by allowing inaccuracy of even a constant multiplicative factor yields an exponential improvement in the worst-case step complexity of the bounded variant and in the amortized step complexity of the unbounded one.

For the sake of gauging the limitations of these relaxations, we study the lower bounds of the complexity of the k-multiplicative-accurate counter and max register in both their bounded and unbounded variations. We obtain the result that when the approximation parameter k does not depend on the number of processes, relaxing counter semantics by allowing inaccuracy of a multiplicative factor cannot asymptotically reduce the amortized step complexity of unbounded counters by more than a logarithmic factor. We also prove that our bounded k-multiplicative-accurate max register is optimal and matches the lower bound.

When it comes to the FIFO queue, designing efficient wait-free implementations remains a challenge despite its usage in many distributed applications. Most of the FIFO queue implementations in the literature rely on concurrency constraints: not all processes are allowed to execute either/or Enqueue and Dequeue operations.

In this thesis, we investigate whether it is possible to implement a logarithmic worst-case step complexity wait-free implementation that does not suffer from concurrency constraints.

Therefore, we present a wait-free FIFO queue implementation that supports n enqueuers and k dequeuers where the worst-case step complexity of an Enqueue operation is in $O(\log n)$ and where the complexity of the Dequeue operation depends on the level of concurrency during the execution and is $O(k \log n)$ in the worst-case scenario.

We then rely on the relaxation of the FIFO queue semantics to show that allowing concurrent Dequeue operations to retrieve the same element results in an implementation with $O(\log n)$ worst-case step complexity for both the Enqueue and Dequeue operations.
Keywords: concurrent shared objects, distributed algorithms, distributed computing, complexity, fault tolerance

---

Laboratoire Bordelais de Recherche en Informatique (LaBRI)
UMR 5800, Université de Bordeaux, 33405 Talence, France.

Acknowledgements

I would like to thank the many people in my life that have supported me throughout the different stages of my thesis and have made it possible for me to reach the other side of the tunnel.

Starting with my supervisors Alessia Milani and Colette Johnen who have been patient and supportive with their guidance even when I found it difficult to look ahead. I also would like to thank the collaborators I had the chance to work with in different capacities namely, Corentin Travers, Danny Hendler, and Jennifer Welch who have helped shape this thesis alongside all the colleagues at the LaBRI who were more than happy to discuss theory or even just hear me rant. Especially, I would like to name Nicolas Hanusse and Pascal Weil, the members of my monitoring committee, and thank them for their encouragement and objective assessment of my thesis throughout the years. I also would like to mention the people I have shared an office with and came to consider friends Yanis, Colleen, Amaury, and many others.

I owe deep gratitude to my close friends whose presence in my life does more than I could ever quantify. Thank you, Marwa, Ghita for the decade-old friendship and your unwavering love and support.

Korlan, who I met the first month of my thesis, and have gone through the entire journey together and I now consider one of my closest friends.

My personal trainer and friend Marc, for inspiring me to live life for myself and teaching me to disconnect and destress during some of the most difficult periods of my thesis.

And the friends that have made an impact on different periods of my life but we have since grown apart, I will always hold love for you.

Finally, to my mom and dad for being an example to me of hard work and dedication no matter what cards you have been dealt in life. I love you and hope you are proud.

`To seeing things through`.

Contents

1 Introduction **9**
- 1.1 Overview . . . 9
- 1.2 Model and Preliminaries . . . 12
  - 1.2.1 Model . . . 12
  - 1.2.2 Termination Conditions . . . 15
  - 1.2.3 Consistency Conditions . . . 16
  - 1.2.4 Sequential Specification Relaxations . . . 19
- 1.3 Related Work and Contributions . . . 20
  - 1.3.1 Counter and Max Register . . . 20
  - 1.3.2 FIFO Queue . . . 22
- 1.4 Organization . . . 25

2 K-multiplicative-accurate Counter and Max Register **26**
- 2.1 Introduction . . . 27
- 2.2 Unbounded k-multiplicative-accurate Counter . . . 29
  - 2.2.1 Algorithm Description . . . 29
  - 2.2.2 Wait-freedom and Technical Lemmas . . . 33
  - 2.2.3 Linearizability . . . 35
  - 2.2.4 Complexity Analysis . . . 38
- 2.3 Bounded k-multiplicative-accurate Counter . . . 41
  - 2.3.1 Algorithm Description . . . 41
  - 2.3.2 Linearizability . . . 42
  - 2.3.3 Complexity Analysis . . . 45
- 2.4 Bounded k-multiplicative-accurate Max Register . . . 45
- 2.5 Unbounded k-multiplicative-accurate Max Register . . . 47
  - 2.5.1 Algorithm Description . . . 47
  - 2.5.2 Linearizability and Wait-freedom . . . 48
- 2.6 Worst-case Step Complexity Lower bound for k-multiplicative-accurate m-bounded Max Register and Counter . . . 52
- 2.7 Amortized Step Complexity Lower bound for k-multiplicative-accurate Counter . . . 54
- 2.8 Discussion . . . 59

3 Efficient Queue Implementations **60**
- 3.1 Introduction . . . 61
- 3.2 Wait-Free Linearizable Queue . . . 62
  - 3.2.1 Inspiration . . . 62

3.2.2 Algorithm Overview 64
3.2.3 Algorithm Pseudocode 65
3.2.4 Proof 69
3.3 Set-Linearizable Wait-free Queue Algorithm with Multiplicity 80
3.3.1 Algorithm Pseudocode and Description 80
3.3.2 Algorithm Properties 82
3.3.3 Set-linearizability 84
3.3.4 FIFO Queue Specification 88
3.3.5 Step Complexity 89
3.4 Discussion 89

4 Conclusion **91**

Chapter 1

Introduction

1.1 Overview

Following the natural evolution of modern hardware architectures into multi-core systems, the issues of synchronicity between different processes became more prevalent. For instance, ensuring the data stored remains consistent for a multi-process system is much more challenging than the case of a single-process environment.

Take, for example, the problem of assigning valid memory addresses for different applications. While the solution for a single-process system consists of simply retrieving the last attributed address and then assigning the next available slot, the problem becomes complex when it requires coordination between multiple processes.

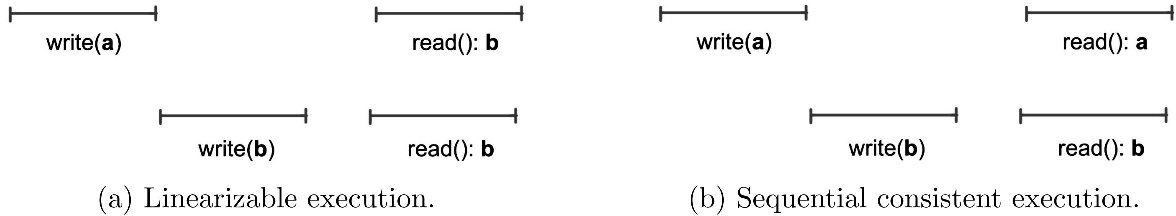
Many fundamental multi-process coordination problems (akin to memory address assignment) can be expressed as *counting problems* [6]. By considering an *abstract data type*, like the *shared counter* in the case of counting problems, it is possible to resolve the synchronicity problems for multiple processes by implementing the data type. This formalism shifts the problems at hand from low-level and architecture-specific to high-level abstract questions.

A *shared object* is an instance of an abstract data type represented by a domain of possible value and a set of *operations* that provide the correct behavior of the object when the operations are invoked in a sequential setting. This definition is what we call the *sequential specification* of the object.

An implementation of a shared object offers the algorithms describing the steps executed by each process when applying an operation on the object being implemented.

Distributed algorithms that implement new shared objects rely on preexisting implementations of common shared objects denoted *base objects* as building blocks because they already solve many of the synchronization problems when considering a distributed execution environment. Relying on base objects also benefits from the *composable* or *local* property of linearizability and wait-freedom; meaning that if all the base objects used in the implementation of a new shared object are linearizable and wait-free, the implemented object is also linearizable and wait-free.

We say that a process takes a *step* during the execution of an operation of an implemented shared object when it executes an operation on a base object. Any computations that are executed locally by the process do not count in the total cost of the algorithm that implements the shared object. An *execution* is the sequence of steps executed by the processes as they follow the algorithms implementing a shared object.



(a) Linearizable execution. (b) Sequential consistent execution.

Figure 1.1: Different execution scenarios for a shared register.

Measuring the correctness of an implementation depends on how closely matched the behavior of the operations executed in a concurrent setting, is to the sequential specification of the object. *Consistency conditions* formalize this distance and can vary by how strictly they relate concurrent executions to the sequential specification. The most common of these conditions is *linearizability* introduced by Herlihy and Wing [26] such that, roughly speaking, an implementation is linearizable if any execution where operations are executed concurrently is equivalent to a sequential execution where each operation appears to take effect instantly at an instance during the execution of the operation and behave according to the sequential specification of the object.

It has been shown that linearizable implementations are often more costly than implementations with more lenient consistency conditions. For instance, this is the case for *sequential consistency* which requires that the operations appear to take place in an order that reflects the order of operations for each individual process as opposed to linearizability which requires a total order of all operations (Figure 1.1). Different results substantiate the claim by showcasing the cost difference between linearizable and sequential consistent implementations of different shared objects from read/write register, FIFO queue, and stack (Attiya and Welch [11]) to snapshot (Petrin et al. [36]).

The implementation of a shared object is also subject to *termination conditions* which ensure a certain degree of progress during an execution. To ensure the operations have some guarantee of ending, these conditions are of varying degrees depending on whether the guarantee of progress is only system-wide or if it is process-specific. We consider the strongest of the termination conditions denoted *wait-freedom* [25] which requires that every operation ends after executing a finite number of steps.

In this thesis, we focus on *complexity analysis* to measure the efficiency of an implementation of a shared object and how it compares to the state of the art. For a given implementation of a shared object, many variations of the complexity of the operations can be calculated: from space complexity to step complexity and from the worst-case scenario to an overall average. In our analysis, We consider the *worst-case step complexity* and the *amortized step complexity*. The worst-case step complexity is defined as the worst-case (over all possible executions) total number of steps taken by an operation. The amortized step complexity is defined as the worst-case average number of steps performed by operations. It measures the performance of an implementation as a whole rather than the performance of individual operations. More precisely, given a finite execution E, an operation Op *appears* in E if it is invoked in E. We denote by $Nsteps(op, E)$ the number of steps performed by op in E and by $Ops(E)$ the set of operations that appear in E. The amortized step complexity of an implementation A

is then:

$$AmtSteps(A) = \max_{E} \frac{\sum_{op \in Ops(E)} Nsteps(op, E)}{|Ops(E)|}$$

A large portion of the research around shared objects centers around the goal of improving the efficiency of the implementations and reaching the best possible complexities. It is also of high interest to prove *lower bounds* or the limits to how low the complexity of an implementation can get under a specific computational model. For many common shared objects (e.g. *Counter*, *Max Register*, *FIFO queue*, etc.), the aim is to find implementations that match the complexity lower bounds.

For instance, a well-known result by Jayanti, Tan and Toueg [29] proved a linear lower bound in the number of processes n on the worst-case step complexity of a large class of shared objects that includes the counter object. An implementation of a wait-free counter with optimal worst-case step complexity can be constructed easily by using a *wait-free atomic snapshot*: Each process has a component in the snapshot object, and to increment the counter, a process simply increments its component. To read the counter's value, the process invokes `Scan` to obtain an atomic view of the snapshot, and returns the sum of all components in the view it obtains. Since wait-free atomic snapshot can be implemented, using reads and writes only, with worst-case step complexity linear in n, e.g. [9], so can counters.

To further optimize beyond this point, different strategies have been conceived to bypass the limitations of a lower bound on shared objects. For instance, by considering bounded executions where the number of operations permitted is restricted to a number m of calls; sub-linear implementations of the counter object have been obtained [5].

More generally, a *bounded* shared object is a variation of a regular object but with a restriction on the number of operations in an execution of the object.

Recently, however, there has been a surge of interest in the relaxation of the sequential specification of different shared objects in order to obtain more efficient implementations. The intuition for these relaxations comes from the disconnect between the strict sequential specification of shared objects and the applicative needs in practical settings.

In many cases, applications can function normally even in the case of some anomalies in data. For instance, in the context of Big Data applications, many popular data platforms including BigQuery [13], Oracle [37], and SQL Server [4] support an approximate form of counting because the real-life applications can tolerate a margin of error and run more efficiently using approximation.

The goal is to be able to implement a shared object in a more efficient manner by allowing operations applied to the object to err to a certain degree defined by the relaxation. This thesis investigates different relaxations of widely used shared objects. And Comparing these relaxed objects to the exact versions, we can gauge the possible optimization of the implementations when applying different relaxations to shared objects.

Specifically, we focus in this thesis on the *k-multiplicative-accurate* relaxation first introduced in [7] for the counter object. We study the relaxation applied to common shared objects, namely the counter and max register. The relaxed sequential specification of the k-multiplicative-accurate counter allows for the return value of a call to *CounterRead* to fall within an approximative range of the value returned by the exact counter. Specifically, a call to *CounterRead* returns x such that $v/k \leq x \leq k \cdot v$ where

v is the exact value of the counter. We also study the same *k-multiplicative-accurate* relaxation applied to the max register. Similarly, an instance of the *MaxRead* operation returns an approximate value x' within a k multiplicative range of the maximum value v' written to the register (i.e. $v'/k \leq x' \leq k \cdot v'$).

Following the results on the relaxed versions of the counter and max register, we shift focus to the FIFO queue and consider the relaxation denoted *multiplicity* and introduced in [14] which allows multiple concurrent *Dequeue* operations to return the same element.

In the remainder of this chapter, we present the model of computation considered throughout the thesis and then give a detailed synthesis of our contributions and their position from the standpoint of the state of the art.

1.2 Model and Preliminaries

1.2.1 Model

We consider the standard asynchronous shared memory model with a set $\mathcal{P}$ of n processes $p_1,\ldots,p_n$. Each process p_i is identified by a unique integer i.

We consider that the processes are prone to crashes. Thus, a process could stop due to a crash at any moment during an execution. In an asynchronous setting, the physical time between two instructions is unknown, making it impossible to know with certainty that another process has crashed. Any distributed algorithm in the asynchronous model must take into consideration the fact that it is impossible to distinguish between the case where a process crashes and the case where it might resume its execution.

In a concurrent setting, the problems of synchronization between processes that arise are modeled using *shared objects*. These objects are defined by a *sequential specification* describing the set of operations that can be invoked on the implemented object as well the correct behavior of the operations in the absence of concurrency.

Formally, a high-level shared object O is a concrete representation of a data type T, composed of a set of states $\mathcal{S}$, a finite set of operations $\mathcal{O}$ and a set of transitions σ between states. A transition $\sigma(s, op(arg)) = (s', res)$ describes the sequential behavior of the object when an instance *op* of an operation is invoked with the argument *arg*, causing the object to move from its current state s to a new state s', and resulting in a response *res* to the operation from the object. We say that *op* is an *update* operation if it changes the state of O.

We say that an object or data type is deterministic if the set of transitions σ is a function; meaning that a specific invocation of an operation on the object from a state would always (in any execution) result in the object transitioning to the same new state and having the same response.

To solve a problem in a distributed system is to present a correct implementation of the shared object under a specific model of computation.

Implementation and execution An *implementation* of a shared object provides a specific data representation for the object from a set of shared *base objects*, each of which is assigned an initial value; the implementation also provides algorithms for each process in $\mathcal{P}$ to apply each operation to the object being implemented. To avoid confusion, we

call operations on the base objects *primitives* and reserve the term *operations* for the objects being implemented.

An *execution fragment* is a (finite or infinite) sequence of steps performed by processes as they follow their algorithms. In each *step*, a process applies at most a single primitive to a base object (possibly in addition to some local computation). An *execution* is an execution fragment that starts from the *initial configuration.* This is a *configuration* in which all base objects have their initial values and all processes are in their initial states. More generally, at any moment during the execution, the *configuration* of E represents the state of all the base objects. We say that an operation is complete in an execution, if it returns within this execution. Otherwise, we say that the operation is pending. If an operation op_1 returns before a second operation op_2 is invoked, we say that op_1 is before op_2 in the real-time execution order, and write $op_1 <_{ro} op_2$.

A set of primitives is *historyless* if all the nontrivial primitives in the set overwrite each other; we also require that each such primitive overwrites itself. A primitive is *nontrivial* if it may change the value of the base object to which it is applied.

In the shared memory model, the processes communicate with each other by applying *primitives* to *base objects.*

The processes are sequential. Meaning that when executing an operation, a process will execute the instructions in order and is not able to execute them in parallel. Since we consider an asynchronous model, the physical time required for the execution of a step might differ from process to process and from one instance of an operation to another. Therefore, any complexity analysis we present is based on the number of steps executed by a process during an operation.

Shared objects In practice, there is no distinction between the shared objects being implemented and the base objects. On a case-by-case basis, the same shared object can play both roles. For instance, it is possible to use a shared counter as a base object for the implementation of a queue, as it is just as possible to have a new implementation of the counter itself.

Aside from the atomic *Read/Write* registers, the following is an exhaustive list of the shared objects we consider in this thesis:

- **Fetch&Inc:** the only primitive executed on the $Fetch\&Inc$ object, is the identically named primitive $Fetch\&Inc$ that increments the value of the object by 1 and returns the value prior to the incrementation.

- **Test&Set:** is set initially to 0, and the first call to the primitive $test\&set$ changes its value to 1. All instances of $test\&set$ return the previous value of the object and we consider that it also takes the simple *Read* primitive.

- **Swap:** takes the primitives $Swap$ and $Read$, such that an instance $Swap(v)$ writes v to the object and returns its previous value.

- **Max register:** takes the two primitives $ReadMax$ and $WriteMax$ such that $ReadMax$ returns the maximum value written to the register through the calls to $WriteMax$.

- **Counter:** can be incremented by calling the primitive $CounterIncrement$ and a call to $CounterRead$ returns the number of calls to $CounterIncrement$ before it.

- **Snapshot:** defined by the two primitives $Update$ and $Scan$. Each process has a corresponding component in the snapshot and is the only one allowed to modify through a call to $Update$. And a process is able to obtain a coherent state of all the components in the snapshot object using the $Scan$ primitive.

- **CAS:** takes the *Read* primitive as well as the `Read-Modify-Write` primitive CAS, such that the call $CAS(old, new)$ writes *new* to the object only if the previous value of the object was *old*.

- **FIFO queue:** provides the two high-level operations *Enqueue* and *Dequeue*. The sequential specification of the queue determines that an instance $Enqueue(v)$ adds the element v at the tail of the queue, while the $Dequeue()$ operation removes the element at the head of the queue and returns its value, if the queue is not empty, otherwise, it returns ϵ.

When limiting the available shared base objects to the "weaker" primitives, It is often the case that the implementation of new shared objects proves to be more difficult. For instance, implementations of the FIFO queue without the CAS object are rare and require clever algorithmic ideas. This "synchronization power" that the CAS has over other base objects is a great indicator of the existence of a hierarchy within the set of shared objects.

Consensus number The notion of *consensus number* was introduced by Herlihy in [25] to describe such a hierarchy for shared objects based on their ability to solve the consensus problem for a specific number of processes.

The *consensus* problem is fundamental in the field of fault-tolerant distributed computing since it models a large set of problems in which processes need to agree on a specific value. The consensus object takes the operation $Propose()$. When process p_i executes an instance $Propose(v_i)$, it signifies that p_i is proposing the value v_i to the consensus. The operation returns the agreed-upon value of the consensus. Formally, any implementation of the consensus object needs to fulfill the following requirements.

Definition 1.2.1. *Consensus: Let $\mathcal{A}$ be an implementation of the consensus object. Let $Propose(v_i)$ be an instance executed by process p_i in which p_i proposes the value $v_i \in V$ the set of possible values of the consensus object. $\mathcal{A}$ satisfies the conditions of safety (validity, agreement) and liveness (termination).*

- ***Validity:*** *If the agreed-upon value is v, a process has invoked $Propose(v)$.*

- ***Agreement:*** *No two processes decide on different values.*

- ***Termination:*** *All non-faulty processes decide on a value.*

Consensus Number	Object
1	read/write registers
2	test&set, swap, fetch&add, queue, stack
...	...
2n-2	n-register assignement
...	...
∞	memory-to-memory move and swap, augmented queue, compare&swap, fetch&cons, sticky byte

Table 1.1: Examples of consensus numbers of different shared objects (table from [25]).

Herlihy hierarchy The consensus number associated with an object is the number of processes we can solve the consensus problem for, using only the object and Read/Write registers. For instance, the Read/Write registers have a consensus number 1, and the $Fetch\&Add$, $Swap$ and $Stack$ objects all have consensus number 2. We say that the consensus number of an object is *infinite* if it can solve the consensus problem in an asynchronous system with n processes for any $n \in \mathcal{N}$. The CAS object is an example of an object with an infinite consensus number. Table 1.1 taken from [25], gives a more comprehensive list of different shared objects and their consensus number. In [25], it is also shown that given an object T with a consensus number i alongside Read/Write registers, it is impossible to have a wait-free implementation of any object with a consensus number $j > i$. However, Jayanti argues in [27] that the Herlihy hierarchy is not a *robust* wait-free hierarchy. Meaning that contrary to what might be assumed, it is possible to implement an object with a consensus number j using a combination of any number of objects with consensus numbers in $1, \ldots, j-1$. Meaning that combining weaker shared objects can result in the implementation of stronger ones.

1.2.2 Termination Conditions

An execution is *non-blocking* if the failure or crash of a process does not impede the progression of other processes. An execution is *lock-free* if there is a guarantee of system-wide progression but not necessarily a guarantee for each process to terminate. An execution is *wait-free* [25] if each process completes its operations if it performs a sufficiently large number of steps. We say that an implementation verifies any termination condition if all its executions do as well.

Definition 1.2.2. *Lock-freedom: Let $\mathcal{A}$ be an implementation of a concurrent object $\mathcal{O}$ and E an execution of $\mathcal{A}$. E is lock-free if for any operation α that takes infinite steps in E there exists infinitely many concurrent operations executed by other processes that terminate in a finite number of steps (system-wide progress).*

Definition 1.2.3. *Wait-freedom: Let $\mathcal{A}$ be an implementation of a concurrent object $\mathcal{O}$ and E an execution of $\mathcal{A}$. E is wait-free if all operations in E terminate in a finite number of steps (per-process progress).*

1.2.3 Consistency Conditions

Since shared objects are defined by sequential specifications, we require a means to relate the correct behavior of the object in a concurrent setting to its definition in the absence of concurrency. Consistency conditions define what behaviors are allowed during a concurrent execution.

Definition 1.2.4. *A consistency condition C is the set of all legal operation sequences of any data type T under C.*

Linearizability

As one of the most intuitive consistency conditions, *linearizability* is used throughout the literature. Roughly speaking, an execution is *linearizable* [26] if each operation appears to take effect atomically at some point between its invocation and response and behaves according to the sequential specification of the object.

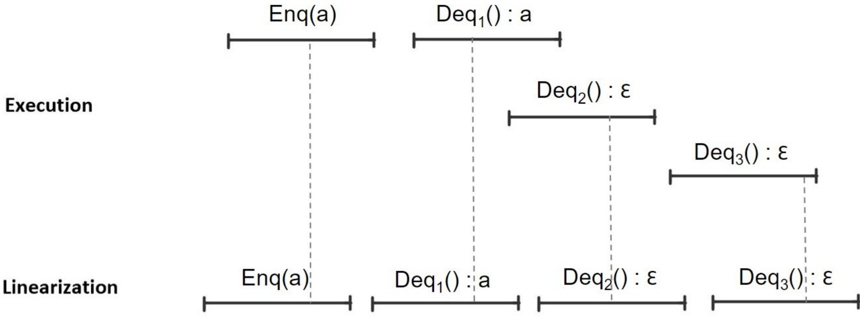
Definition 1.2.5. *Linearizability: Let $\mathcal{A}$ be an implementation of a concurrent object $\mathcal{O}$. An execution E of $\mathcal{A}$ is linearizable if there is a sequential execution S of $\mathcal{O}$ such that S contains every completed operation of E and might contain some pending operations, and the inputs and outputs of the invocations and responses of the operations in S agree with the inputs and outputs in E and behave according to the sequential specification of $\mathcal{O}$. Furthermore, if two completed operations op_1 and op_2 in E verify $op_1 <_E op_2$, then op_1 appears before op_2 in S.*
We say that an implementation is linearizable if all its executions are linearizable.

Weaker Consistency Conditions

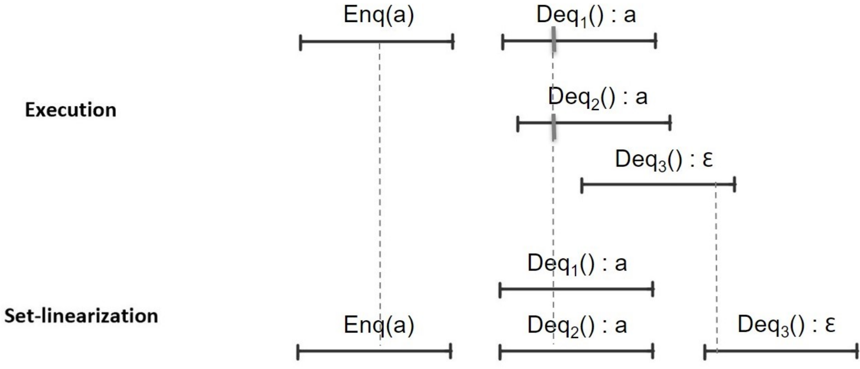
In recent years, the *relaxation* of the implementations of shared objects for the sake of solving scalability issues, has become the focus of many research topics. We present next some of the most common weaker consistency conditions.

Formally, the set-sequential specification of a shared object differs from its sequential specification in regards to the definition of the transition function σ between states of the object. In the set-sequential specification, instead of taking a single instance op of an operation as a parameter and having a single response res, a transition $\sigma(s, \mathcal{S}(op)) = (s', \mathcal{S}(res))$ between two states s and s', of the object takes a set of instances $\mathcal{S}(op) = \{op_1, \ldots, op_i\}$ of operations as a parameter and returns a set of responses $\mathcal{S}(res) = \{res_1, \ldots, res_i\}$ where each response corresponds to an instance of an operation from the parameter set. An execution E of a concurrent object is *set-linearizable* if there exists an equivalent set-sequential execution S that contains all the complete operations of E and possibly some pending operations, and the execution S verifies that if an operation op is before another operation op' in E (i.e. $op <_E op'$) then op is also before op' in S. Figure 1.2 illustrates the differences between a linearization of an exact FIFO queue and a set-linearization of a relaxed FIFO queue where multiple concurrent $mathitDequeue$ operations are allowed to return the same element. In Figure 1.2a, the linearization defines a sequential total order of all the operations, while in Figure 1.2b, multiple operations have the same linearization point and are executed concurrently in the set-linearization.

Definition 1.2.6. *Set-linearizability: Let $\mathcal{A}$ be an implementation of a concurrent object $\mathcal{O}$. An execution E of $\mathcal{A}$ is set-linearizable if there is a* set-sequential *execution S of $\mathcal{O}$ such that S contains every completed operation of E and might contain some pending operations, and the inputs and outputs of the invocations and responses of the operations in S agree with the inputs and outputs in E. Furthermore, if two completed operations op_1 and op_2 in E verify $op_1 <_E op_2$, then op_1 appears before op_2 in S.*
We say that an implementation is set-linearizable if all its executions are set-linearizable.



(a) Linearizable execution.



(b) Set-linearizable execution.

Figure 1.2: Example of a linearization of an execution of a FIFO queue and a set-linearization of an execution of a relaxed FIFO queue.

The consistency condition of *interval-linearizability* is introduced by Castañeda et al. [15] to take into consideration the set of problems in the distributed setting that cannot be represented with a sequential specification of an object. This is the case for the *write-snapshot* object as observed by Neiger [35]. Differently from the regular snapshot object defined by the two operations *Update* and *Scan*, the write-snapshot is defined by a single operation that concatenates the two: when a process invokes the instance *write-snapshot(x)* to add the value x to the object, the operation returns the state of the object. Neiger notes that it is impossible to represent the write-snapshot using a sequential specification and can only be modeled as a task. The execution in Figure 1.3 illustrates the case where an execution of a write-snapshot is not set-linearizable. No matter where the instance *write-snapshot(b)* is linearized, the set-linearization obtained has an operation that returns a value that appears to be predicting a future operation.

Hence, the need for a more flexible framework where it is possible to express that an operation happens over an interval of time that can be affected by multiple operations.

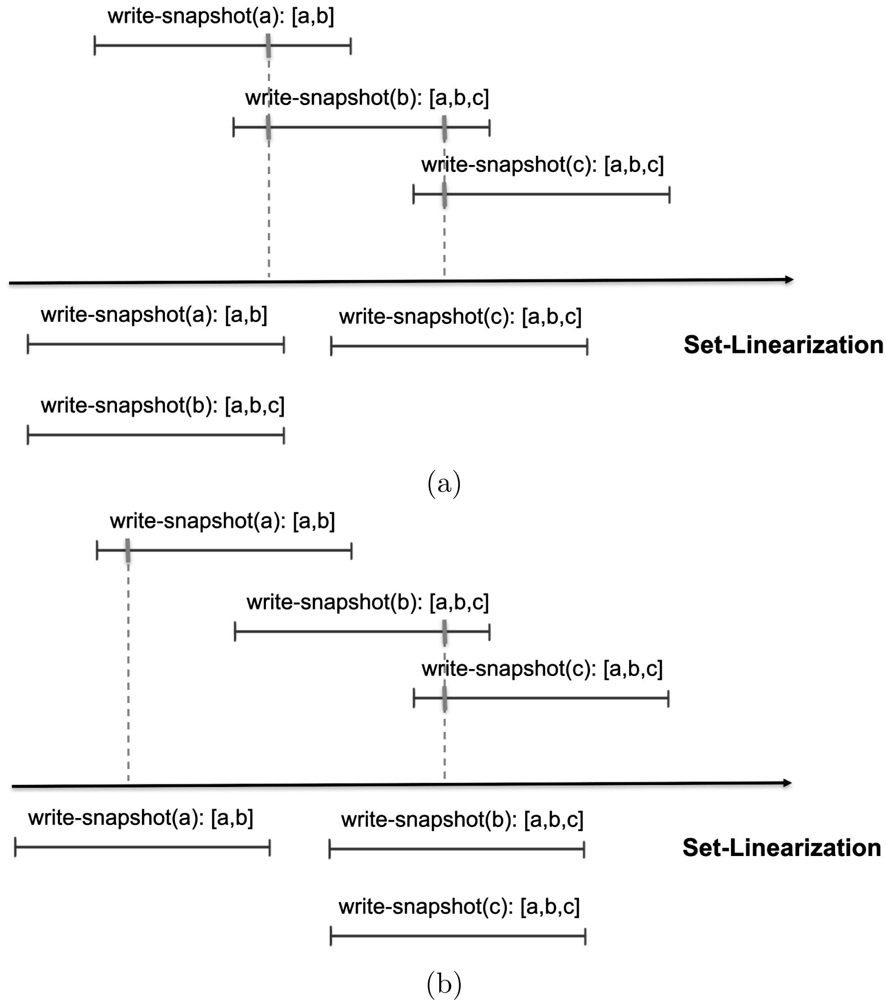


Figure 1.3: Example of an execution of write-snapshot that is not set-linearizable.

Formally, in the *interval-sequential specification* of an object, if an operation *op* is pending in a state q, and the transition σ is applied such that $\sigma(q, Inv) = (q', Res)$ where Inv is a set of operation invocations and Res is a set of responses; then *op* might still be pending in q'. Meaning that Res contains the responses of only the operations that are complete in q'.

Definition 1.2.7. *Interval-linearizability: Let $\mathcal{A}$ be an implementation of a concurrent object $\mathcal{O}$. An execution E of $\mathcal{A}$ is interval-linearizable if there is an* interval-sequential *execution S of $\mathcal{O}$ such that S contains every completed operation of E and might contain some pending operations, and the inputs and outputs of the invocations and responses of the operations in S agree with the inputs and outputs in E. Furthermore, if two completed operations op_1 and op_2 in E verify $op_1 <_E op_2$, then op_1 appears before op_2 in S.*
We say that an implementation is interval-linearizable if all its executions are interval-linearizable.

Other weaker consistency conditions include *quasi-linearizability* [2], which models legal executions with a distance function from sequential executions. *Intermediate value linearizability* [39] is defined through linearizability such that a read operation is allowed to return a value that is bounded by two values that are legal under linearizability. And similarly to abstract data type relaxations, some consistency conditions are introduced for specific data structures. For instance *local linearizability* [20] is defined for container-type data structures like queues and stacks, with a guarantee of a consistent view of the object only at the process level as opposed to regular linearizability which guarantees a consistent view overall (for local-linearizability a projection of the global execution onto a specific process is linearizable).

1.2.4 Sequential Specification Relaxations

While considering weaker consistency conditions is one way to implement relaxed shared objects, The second approach that has emerged is the relaxation of the sequential specification of the object.

Henzinger et al. [24] introduced a formal framework for obtaining new data structures by quantitatively relaxing existing ones. Intuitively, the framework defines a distance between sequences of operations such that a sequence that might not be permitted under the sequential specification of the original object, might be allowed for the relaxed version of the object if the sequence is at some distance k from a sequence of operations that is legal. Several authors [43, 31, 38, 45, 41, 42] have used this framework to give different implementations of relaxed data structure types or to study properties of specific relaxations.

The first general relaxation that results from this framework is the *Out-of-order* relaxation. When applied to an ordered data structured like the queue or stack, this relaxation allows the deleter operation (*Dequeue* and *Pop*, respectively), to return an element up to k places out-of-order.

The other generic relaxation presented in [24] is the *stuttering* relaxation. This relaxation allows some update operations to not take effect, meaning that the call to an operation that changes the state of the object "stutters" and does not succeed in modifying the state of the object. For a sequence of operations to be allowed under this relaxation, no more than k consecutive update operations can stutter before an operation succeeds in changing the state of the object.

The *k-atomicity* relaxation defined in [3], resembles the stuttering relaxation in that it allows read operations to return a "stale" value bounded by the parameter k to limit the number of write operations it can overlook. However, the definition of k-atomicity differs from stuttering in the sense that for the latter, an operation that changes the state of the object might "stutter" and fail to do so, while for the former, the relaxation affects the read operations alone.

Aside from this framework, there have been data type-specific relaxations that are defined with the sequential specification of the object in mind. For instance, Castañeda et al. [14] define a relaxed queue object with *multiplicity*, such that multiple concurrent instances of the *Dequeue* operation are allowed to return the same element in the queue. They also define a relaxation that allows an instance of *Dequeue* to return a special value *weak-empty* when the queue *might* be empty.

Relaxation Equivalence

Talmage and Welch show in [43] that in many ways the two approaches of considering weaker consistency conditions and relaxing the object's sequential specification are different ways to specify the same sets of allowed concurrent behaviors of a given shared object. They give in subsequent work [44] equivalent consistency conditions to different abstract data type relaxations, namely k-out-of-order, k-lateness, and k-stuttering. In doing so, they prove that in many cases both relaxing the sequential specification and considering weaker consistency conditions are two equivalent ways to represent the same set of concurrent problems faced when implementing shared data objects. Meaning that it is possible to focus on whichever approach is easier to achieve thanks to this equivalence.

1.3 Related Work and Contributions

In this section, we present the related work to the different problems we investigate in this thesis as well as our contributions. Starting with some lower bounds results justifying the limitations of step complexities of the implementations of the shared objects in the absence of any relaxation of the consistency condition or sequential specification. We then present some relaxations of shared objects with a focus on the objects we are interested in (counter, max register). Finally, we present the general landscape of wait-free FIFO queue implementations.

1.3.1 Counter and Max Register

Jayanti, Tan and Toueg [29] show for any deterministic non-blocking n-process implementation I of a shared object in a large set A using a set B of primitives where $A =$ {increment, fetch&add, modulo k counter (for any $k \geq 2n$), LL/SC bit, k-valued compare&swap (for any $k \geq n$), single-writer snapshot} and $B =$ {{resettable consensus} $\bigcup$ {historyless objects such as registers and swap registers}}, that I has a lower bound for both time and space complexities of $n-1$. To illustrate the intuition behind this lower bound, take the example of a simple implementation of a counter object for n processes p_i with $i \leq n$. If p_n executes a $Read()$ operation op_n, it will need to read a single base object at least to retrieve the value of the counter. However, in the meantime, a different process p_l could execute an instance of $CounterIncrement()$ to change the value of a base object of the implementation of the counter. This would render the value read by p_n obsolete and forces it to read another base object. If this scenario occurs for every process p_i such that $i \neq n$, then p_n will need to read $n-1$ base objects.

However, this lower bound result does not consider restricted-use objects. And in many cases, there have been implementations of objects that beat the $\Omega(n)$ lower bound because the executions in the restricted-use context exceed the scope of the proof of the lower bound. For instance, Aspnes, Attiya, and Censor-Hillel [5] show the possibility of implementing exact counting algorithms whose step complexity is sub-linear when the number of operations is bounded. In particular, they presented a wait-free exact counter for which the worst-case step complexities of the *CounterIncrement* and *CounterRead* operations are $O\big(min(\log n \log v, n)\big)$ and $O\big(min(\log v, n)\big)$, respectively, where v is the

object's current value. In [5], they also give an implementation of a max register that can write v in $O(min(\log v, n))$ steps.

For this reason, Aspnes, Censor-Hill, Attiya, and Hendler [7] generalize the lower bound results in [29] to bounded shared objects. More specifically, through this generalization, they propose a new lower bound for both time and space complexities in $\Omega(min(\log L, n))$ for deterministic implementations from historyless primitives of bounded objects where L is the bound parameter. This lower bound also proved that the m-bounded max register implementation in [5] is optimal.

For shared objects that manipulate numerical values, a natural relaxation might consist of allowing an additive margin of error for the return value of the read operation. This is the case for the *k-additive-accurate* counter introduced in [5] as a counter for which any *CounterRead* operation returns a value that is within $\pm k$ of the number of *CounterIncrement* operations linearized before it. It is then shown that for any deterministic solo-terminating implementation from atomic registers by n processes of an m-bounded k-additive-accurate counter, there is a *CounterRead* operation that takes $min(n-1, \lceil \log m \rceil - \log(\lceil \log m \rceil + k))$ steps. Meaning that allowing the *CounterRead* operation to have some additive error accounts for the cost of some of the accumulating pending operations. This lower bound is improved in [7], where it is shown that the m-bounded k-additive-accurate counter has a lower bound of $\Omega(min(\log m - \log k, n))$.

	Step complexity	max(steps, stalls)	Space complexity
max register	$\Omega(\min(\log m, n))$	$\Omega(\min(\log m, n))$	$\Omega(\min(m, n))$
Counter	$\Omega(\min(\log m, n))$	$\Omega(\min(\log m, n))$	$\Omega(\min(\sqrt{m}, n))$
k-additive counter	$\Omega(\min(\log m - \log k, n))$	$\Omega(\min(\log m - \log k, n))$	$\Omega(\min(\sqrt{\frac{m}{k}}, n))$

Figure 1.4: Lower bounds on restricted use objects where m is the maximum value assumed by the object or the bound on the number of operations applied to it, from [7]

.

The first relaxation we consider is the *k-multiplicative-accurate* relaxation introduced in [7].

Contribution: k-multiplicative-accurate max register

We define the *k-multiplicative-accurate* max register, where the *MaxRead* operation returns an approximate value x' within a k multiplicative range of the maximum value v' written to the register (i.e. $v'/k \leq x' \leq k \cdot v'$).

we have shown that relaxing the semantics of the bounded max register by allowing inaccuracy of even a constant multiplicative factor yields an exponential improvement in the worst-case step complexity. Then, we present a novel m-bounded k-multiplicative-accurate max register algorithm whose worst-case step complexity matches this lower bound. We then easily "plug-in" our bounded k-multiplicative-accurate max regis-

ter into the construction proposed by Baig et al. [12] to obtain an unbounded k-multiplicative-accurate max register with sub-logarithmic amortized step complexity.

Contribution: k-multiplicative-accurate counter

Similarly to the k-multiplicative-accurate max register, reading the of value the k-multiplicative-accurate counter through a call to the operation *CounterRead* returns an approximation x of the exact value v of the counter by a multiplicative factor of k (i.e. $v/k \leq x \leq k \cdot v$).

We implement a wait-free linearizable k-multiplicative-accurate counter for $k \geq n$ where n is the number of processes, with *constant* amortized step complexity for executions of arbitrary length. We also give an implementation of the m-bounded variant of the k-multiplicative-accurate counter for $k \geq \sqrt{n}$ with a worst-case step complexity in $O(min(\log(\log(m+1)), n)$.

Then, by extension of the lower bound of Attiya and Hendler, [10], we prove that any implementation of a k-multiplicative-accurate counter from read/write and conditional primitive operations has amortized step complexity of $\Omega(\log(n/k^2))$, for $k \leq \sqrt{n/2}$. Our results together with the upper and lower bound on exact counting proved in [12] show that when the approximation parameter k does not depend on n, relaxing the counter semantics by allowing a multiplicative error cannot asymptotically reduce the amortized step complexity by more than a logarithmic factor.

We also show a lower bound for unbounded k-multiplicative-accurate counters for the worst-case step complexity in $\Omega(n)$. Meaning that the linear lower bound by Jayanti, Tan, and Toueg [29] for exact counters also holds in the case of the k-multiplicative-accurate counters.

1.3.2 FIFO Queue

There have been results showing the difficulties of implementing a linearizable wait-free queue because of the *"tail chasing"* problem. Roughly speaking, the "tail chasing" scenario occurs when a process is trying to retrieve an element from the queue but finds itself unable to return because it is invisible to other processes that keep modifying the state of the queue indefinitely by executing operation sequences that contain element insertions followed by dequeuing elements from the queue.

The difficulty to implement the queue in a wait-free manner is formalized by Attiya, Castañeda, and Hendler [8]. They categorize helping mechanisms into *trivial* and *non-trivial* helping. This distinction relies on the definition of *operation valency* introduced in [22] to describe the possible return values an operation can have. Roughly speaking, an implementation has helping if a process makes another process decide on a return value by executing a specific mechanism. In the case of queues, stacks, and similar data structures, the helping is nontrivial, if the non-decided process is made to return a value different than ϵ (the empty state of the object). This often requires delicate communication between the processes to ensure that a value reserved for a specific undecided process is not taken by a process unaware of the helping taking place. The main result from [8] is the distinction between stack and queue implementations: A wait-free queue implementation requires nontrivial helping while a stack can be implemented in a wait-free manner without nontrivial helping [1].

An example of such a helping mechanism is used by Li [33] to implement a wait-free multiple enqueuer 2 dequeuer queue. In this implementation, each instance of the *Dequeue* operation is represented by a node in a linked list denoted *DeqCell*. The position of an instance in the list is dictated by a sequence number that defines a total order for the *Dequeue* operations. In addition to the sequence number of the operation and the id of the process invoking the operation (i.e. d_0 or d_1), a node in *DeqCell* also stores both the index and the value of the element returned by the operation (Figure 1.5).

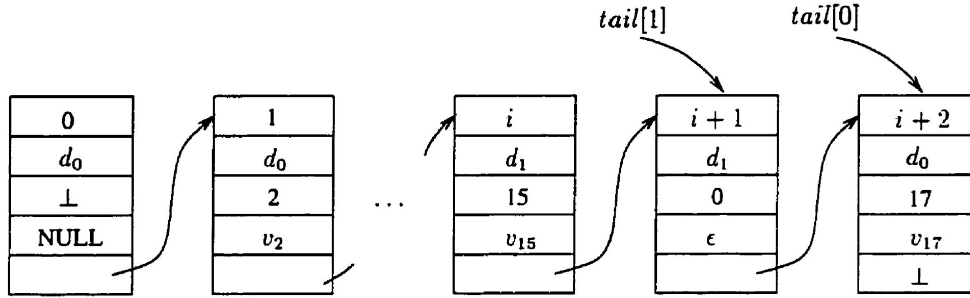


Figure 1.5: The linked list data structure storing instances of the *Dequeue* operation, from [33].

The *Dequeue* operation uses a 2-process consensus object to communicate between the two dequeuer processes. They are used to propose the index of the instance of *Dequeue* that needs to be executed first as well as to agree on its return value. When a process executes an instance of *Dequeue*, it creates a node in *DeqCell* with a new sequence number. Then, it verifies that there is no pending *Dequeue* operation from the other dequeuer process. If there is, the process proposes the index of the pending operation, otherwise, it proposes the index of its own operation. After a response is received from the consensus object, the process executes the corresponding *Dequeue* operation. By the end of this execution, the process verifies if its own *Dequeue* operation has been assigned a return value, and returns if it is the case. Otherwise, it repeats the same steps previously executed to help the pending operation, but for the node in *DeqCell* it created.

A characteristic of the queue implementation by Li [33] is that it only uses primitives with consensus level 2. In fact, it attempts to answer the open question of whether the queue object belongs into *Common2* and is implementable with consensus 2 level primitives only. There exist other implementations based on registers and Common2 objects [17, 16]. However, all these implementations rely on concurrency conditions that limit the number of either enqueuer (e.g. [17, 16]) or dequeuer (e.g. [33]) processes.

Even when considering strong primitives like the *CAS* primitive, it is often necessary to compromise between the concurrency constraints and the complexity of the wait-free queue implementations.

Using *Compare&Swap*, some practical wait-free queue implementations that support multiple enqueuers and multiple dequeuers have been proposed [32, 34, 46, 19]. Some of these implementations are wait-free [32, 46, 19]; while some are only lock-free [34]. All these solutions have been evaluated empirically and do not have formal complexity analysis. Nonetheless, the worst-case step complexity of either the *Enqueue* or

of the *Dequeue* operation is not sublinear.

The best-known upper bound for the worst-case step complexity of wait-free queue implementations is given by Khanchandani and Wattenhofer [30]. They present an algorithm in which both the *Enqueue* and *Dequeue* operations take $O(\sqrt{n})$ steps and require $O(nm)$ registers of $O(max(\log n, \log m)$ bits, where n is the number of processes and m is the bound on the number of *Enqueue* operations. The previous upper bound prior to their work was in $O(n)$ and relied only on the strong primitive CAS.

Inspired by the algorithm proposed by Ellen et al. [18] to solve the consensus for infinitely many processes in $O(1)$ by combining the functionalities of weak primitives, Khanchandani et al. aimed to show through their implementation, that it is possible to improve the complexity of a shared object implementation by using a combination of strong and weak primitives. For that reason, they introduce the *register TH*: a new data type that takes two operations *half-increment* and *half-max*. This register is composed of two components, i.e. (t, h). A call to *half-increment()* increments the value of t as long as $t \leq h$, and a call to *half-max(i)* writes the maximum between i and the previous value of h, to h. It is shown that the two operations *half-max* and *half-increment* have a consensus number of 1 and 2 respectively. Register TH is used to represent the head and tail of the queue.

In addition to the register TH, they use an array to store the queue elements as well as the data structure *counting set* they introduce to manage possible concurrency between *Enqueue* and *Dequeue* operations. More specifically, the counting set takes the two operations *insert*() and *remove*(). An instance *insert*(x) adds the element x into the set and returns the number of total inserts completed (i.e. it also counts apart from inserting the elements). The call to *remove*(i) will remove the i-th inserted element to set if and only if this insert was the last one executed by the corresponding process. For the queue implementation, the counting set is needed when there is a call to a *Dequeue* operation that is concurrent with a pending *Enqueue* operation which has yet to insert an element into the array.

The counting set has two main functions. First, it defines a global order for all *Enqueue* operations. An instance of *Enqueue*(x) invokes *insert*(x) on the counting set and retrieves the index it uses to insert the element into the array. The second purpose of the counting set consists of ensuring that a fast *Dequeue* instance that reaches an index of the array that has not been filled yet by a pending *Enqueue*, is capable of executing a call to *remove*(i) on the counting set to retrieve the element and return, guaranteeing the wait-freedom of the implementation.

The main difficulty in implementing the counting set object resides in transforming a local value of the counter of *Enqueue* operations of a single process, into a global index defined for all processes. A log system is used to store information regarding every *Enqueue* operation in order to compute global indexes. The sublinear complexity is obtained through an optimization of the log by limiting the concurrency during write operations to $\sqrt{n}$ processes instead of all n processes. Therefore, it seems that this approach of using the counting set is limited by this complexity and may not be easily transferable to implement other shared objects or to investigate logarithmic complexity queue implementations.

To the best of our knowledge, all other wait-free queue implementations with sublinear worst-case step complexity in the literature rely on limiting the number of processes allowed to execute either *Enqueue* of *Dequeue* operations (e.g. [28, 16, 33, 17]). Jayanti

and Petrovic [28], for instance, give an implementation of a queue that supports a single dequeuer process and any number of enqueuers. Their implementation has a worst-case step complexity of $O(\log n)$ for both *Enqueue* and *Dequeue* operations, where n is the number of processes.

Contribution: FIFO queue

In this thesis, we were interested in the open question of whether it is possible to have a wait-free queue implementation in logarithmic worst-case step complexity with no concurrency constraints. In particular, we proposed a wait-free FIFO queue implementation that supports n enqueuers and k dequeuers where the worst-case step complexity of an *Enqueue* operation is in $O(\log n)$ and of a *Dequeue* operation is in $O(k \log n)$. But then, by considering a relaxation of the FIFO queue where multiple concurrent *Dequeue* operations are allowed to return the same element, we have shown that it is possible to implement a wait-free FIFO queue with no concurrency constraints in logarithmic step complexity.

We have also investigated the possibility of implementing an exact wait-free FIFO queue using only objects of consensus number 2. As a preliminary approach to solving the question, we limited the execution to 2 processes and presented a wait-free implementation of the FIFO queue based on such objects without relying on universal constructions or on the consensus object which cannot be used to generalize the implementation to more processes without losing the property of having a consensus number 2.

1.4 Organization

The thesis is structured as follows. First, we present in Chapter 2 the implementations of both the counter and max register objects under the relaxed semantics of k-multiplicativity. We investigate different variants of these implementations under the properties of wait-freedom and linearizability. Specifically, we present both an unbounded and bounded approximate k-multiplicative-accurate counter and max register implementations. We then present different lower bounds results for these objects: mainly we prove a lower bound on the amortized step complexity for the unbounded k-multiplicative-accurate counter by extension of a lower bound by Attiya and Hendler [10]. Additionally, we give a lower bound for the worst-case step complexity of the m-bounded k-multiplicative-accurate max register and counter objects.

Then, in Chapter 3, we present a wait-free linearizable FIFO queue implementation for n-enqueuer and k-dequeuer processes with a worst-case step complexity of $O(\log n)$ for the *Enqueue* operation and $O(k \log n)$ for the *Dequeue* operation. Then, we consider the relaxed semantics of the FIFO queue introduced in [14] where multiple concurrent *Dequeue* operations are allowed to return the same element, denoted *multiplicity*. We give an implementation of set-linearizable FIFO queue with multiplicity where both the *Enqueue* and *Dequeue* operations are in $O(\log n)$.

Finally, in Chapter 4 we offer some overall insights on the work in retrospect while discussing possible leads and prospects for future work.

Chapter 2

K-multiplicative-accurate Counter and Max Register

Abstract

Relaxing the sequential specification of shared objects has been proposed as a promising approach to obtain implementations of shared objects with better complexities.

By considering the case study of two common shared objects: max register and counter, we study the possible improvement in step complexity of their relaxed implementations compared to implementations of the corresponding exact objects. In particular, in the classical shared memory model, we investigate the extent to which allowing wait-free linearizable implementations of these objects to return approximate values, rather than accurate ones, may improve their step complexity.

We consider the k-multiplicative-accurate max register and the k-multiplicative-accurate counter, where read operations are allowed to return an approximate value within a multiplicative factor k of the accurate value (for some $k \in \mathbb{N}$). More specifically, reads are allowed to return an approximate value x of the maximum value v previously written to the max register, or of the number v of increments previously applied to the counter, respectively, such that $v/k \leq x \leq v \cdot k$. We provide upper and lower bounds on the complexity of implementing these objects in a wait-free manner in the shared memory model.

We give an implementation of the k-multiplicative-accurate counter that has an exponentially better amortized step complexity than the best implementation of the exact counter in the state of the art when the approximation parameter $k \geq n$.

We also implement the k-multiplicative-accurate max register with an exponentially better worst-case step complexity compared to the exact max register implementation.

We give lower bounds on the worst-case step complexity of the bounded variant of both the relaxed counter and max register, as well as a lower bound on the amortized step complexity of the unbounded counter.

An earlier version of this work containing the lower bound results was presented during the 41st IEEE International Conference on Distributed Computing Systems (ICDCS 2021) [23].

2.1 Introduction

With the ubiquitousness of multi-core and multi-processor systems, there is a growing need to gain a better understanding of how to implement concurrent objects with improved complexity, while maintaining the natural correctness guarantee provided to programmers by linearizability. Relaxing the sequential specification of linearizable concurrent objects is one promising approach to achieving this [2, 24]. An object's *sequential specification* defines its correct behavior in sequential executions. Roughly speaking, *linearizability* [26] guarantees that any concurrent execution is equivalent to a sequential one.

There is empirical evidence that relaxing the sequential specification of some common objects, e.g. queues and counters, yields improved performance of linearizable implementations, e.g [24, 40]. However, the theoretical principles to implement concurrent objects more efficiently by relaxing their sequential specification are not yet clear.

We study relaxed-semantics variants of two well-known concurrent objects – counters and max registers, in the classical shared memory model. In particular, we investigate the extent to which allowing wait-free linearizable implementations of these objects to return approximate values, rather than accurate ones, may improve their step complexity.

A counter is a linearizable object that supports a *CounterIncrement* operation and a *CounterRead* operation. The sequential specification of a counter requires that a *CounterRead* operation returns the number of *CounterIncrement* operations that precede it. A relaxed variant of the counter is the *k-multiplicative-accurate* counter, defined by Aspnes, Censor-Hill, Attiya, and Hendler in [7], where a *CounterRead* operation returns an approximate value x of the number v of *CounterIncrement* operations that precede it, such that $v/k \leq x \leq v \cdot k$ for some parameter $k > 0$.

A max register r supports a $Write(v)$ operation that writes a non-negative integer v to r and a *Read* operation that returns the maximum value previously written to r, [7]. We define the k-multiplicative-accurate max register by allowing a *Read* operation to return an approximate value x of the largest value v written before it, such that $v/k \leq x \leq v \cdot k$ for some parameter $k > 0$.

k-multiplicative-accurate counter

To the best of our knowledge, we present the first deterministic approximate counter with *constant amortized complexity*. More precisely, we present a wait-free linearizable k-multiplicative-accurate counter for $k \geq n$ where n is the number of processes, with *constant* amortized step complexity for executions of arbitrary length. Then, by extension of the lower bound of Attiya and Hendler, [10], we prove that any n-process solo-terminating implementation of a k-multiplicative-accurate counter from read/write and conditional primitive operations (including k-word compare-and-swap) has amortized step complexity of $\Omega(\log(n/k^2))$, for $k \leq \sqrt{n/2}$. Our results together with the upper and lower bound on exact counting provided by Baig et al. in [12], show that when the approximation parameter k does not depend on n, relaxing the counter semantics by allowing a multiplicative error cannot asymptotically reduce the amortized step complexity by more than a logarithmic factor. Table 2.1a compares the amortized

Unbounded (Amortized complexity)	Lower bound	Upper bound
Exact Counter	$\Omega(\log n)$ [12]	$O(\log^2 n)$ [12]
k-multiplicative-accurate Counter	$\Omega(\log n/k^2)$ $k \leq \sqrt{n/2}$ (Section 2.7)	$O(1)$ $k \geq n$ (Section 2.2)

(a) Unbounded counter results.

Bounded (Worst-case complexity)	Lower bound	Upper bound
Exact Counter	$\Omega(min(\log m, n))$ [5]	$O(min(\log n \log m, n))$ for Inc $O(min(\log m, n))$ for Read [5]
k-multiplicative-accurate Counter	$\Omega(min(\log(\log_k m), n)$ (Section 2.6)	$O(min(\log(\log(m+1)), n)$ $k \geq \sqrt{n+1}$ (Section 2.3)

(b) Bounded counter results.

Table 2.1: k-multiplicative-accurate counter implementations and lower bounds results (n is the number of processes and m is the bound on the object).

complexity of the implementation of our unbounded k-multiplicative-accurate counter and the lower bound result to the results from [12].

Then, we consider the bounded version of the k-multiplicative accurate counter. More precisely, we give a wait-free linearizable m-bounded k-multiplicative-accurate counter for $k \geq \sqrt{n+1}$ where n is the number of processes and m is the bound on the number of *CounterIncrement* operation instances that can be performed on the counter. The implementation has a worst-case step complexity of $O(min(\log(\log(m+1)), n)$. We also prove that a lower bound on the *worst-case step complexity* of obstruction-free implementations of m-bounded k-multiplicative-accurate counters from *historyless* primitives in $\Omega(\min(n, \log_2 \log_k m))$. Meaning that our implementation of the m-bounded k-multiplicative-accurate counter is optimal. This also implies that for unbounded k-multiplicative-accurate counters, the worst-case step complexity is in $\Omega(n)$, and we fall back to the linear lower bound by Jayanti, Tan and Toueg [29]. Table 2.1b summarizes the results for the bounded approximate counter and compares them to upper and lower bound results for the exact bounded counter from Aspnes, Attiya and Censor-Hillel [5].

k-multiplicative-accurate max register

We prove that relaxing the semantics of the bounded max register by allowing inaccuracy of even a constant multiplicative factor yields an exponential improvement in the worst-case step complexity compared to the exact max register (Table 2.2). In particular, we prove that the worst-case step complexity of obstruction-free read/write implementations of m-bounded k-multiplicative-accurate max registers is $\Omega(\min(n, \log_2 \log_k m))$, where n is the number of processes. A max register is m-bounded, if it can only represent values in $\{0, \ldots, m-1\}$. Then, we present a novel m-bounded k-multiplicative-accurate max register algorithm whose worst-case step complexity matches this lower bound.

We then "plug in" our bounded k-multiplicative-accurate max register into the construction proposed by Baig et al. [12] to obtain an unbounded k-multiplicative-accurate max register with sub-logarithmic amortized step complexity.

Bounded (Worst-case complexity)	Lower bound	Upper bound
Exact Max Register	$\Omega(min(\log m, n))$ [5]	$O(min(\log m, n))$ [7]
k-multiplicative-accurate Max Register	$\Omega(min(\log_2 \log_k m, n))$ (Section 2.6)	$O(min(\log_2 \log_k m, n))$ (Section 2.4)

(a) Bounded max register results.

Unbounded (Amortized complexity)	Upper bound
k-multiplicative-accurate Max Register	$O(log_2(log_k(m)))$ $m \geq n^2$ (Section 2.5)

(b) Unbounded max register result (m is the parameter of the bounded max register used in the unbounded max register implementation).

Table 2.2: k-multiplicative-accurate max register implementations and lower bounds results (n is the number of processes and m is the bound on the object).

Hereafter is the chapter organization. In Section 2.2, we present the unbounded k-multiplicative-accurate counter implementation with the wait-freedom and linearizability proofs and the complexity analysis. In Section 2.3, we give the implementation of the bounded variant of the k-multiplicative-accurate counter alongside the proofs of progression and correctness. Then we present the implementations of the bounded and unbounded k-multiplicative-accurate max register in Section 2.5 and Section 2.4, respectively. Finally, we give lower bound results for the worst-case step complexity of the bounded k-multiplicative-accurate counter and max register in Section 2.6, and the lower bound result for the amortized step complexity of the k-multiplicative-accurate counter in Section 2.7.

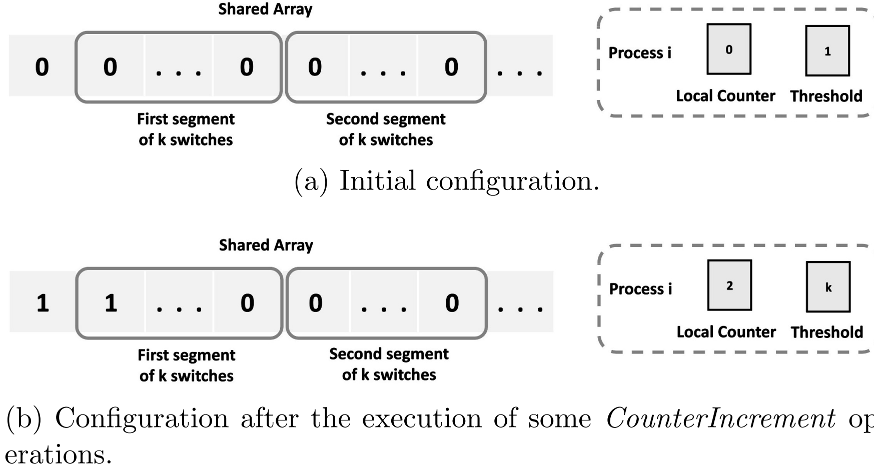
2.2 Unbounded k-multiplicative-accurate Counter

Algorithm 1 describes a wait-free linearizable unbounded k-multiplicative-accurate counter with $k \geq n$ whose amortized step complexity is constant.

2.2.1 Algorithm Description

Figure 2.1 represents the main data structure of the implementation. The algorithm uses an unbounded sequence of bits initially equal to 0, denoted $switch_0, switch_1, \ldots$ to approximately keep track of the number of increments that have been performed by the processes. For each $i \geq 0$, $switch_i$ can be accessed by *test&set* and *read* operations. $switch_i.test\&set()$ sets the value of $switch_i$ to 1 and returns its previous value. A *read* simply returns the value of $switch_i$.

In a nutshell, each process locally keeps an accurate count of the number of *CounterIncrement* operations it performs that are not yet known by the other processes. When this count reaches a certain threshold, the process tries to inform other processes of the number of increments it has performed locally, by attempting to set to 1 a switch in an appropriate bounded range. When a process succeeds in setting a switch to 1, it will restart the



(a) Initial configuration.

(b) Configuration after the execution of some *CounterIncrement* operations.

Figure 2.1: The main data structure for the implementation of the k-multiplicative-accurate counter in Algorithm 1.

local count from 0. *switch* bits are set in increasing order with regard to their index, one after the other.

In particular, the initial value of the threshold is 1 and after their first call to *CounterIncrement*, each process will attempt to set $switch_0$. Afterward, the sequence of $switch_i$ with $i \geq 1$ is partitioned into consecutive intervals of size k. For any such interval $[qk+1, (q+1)k]$, where k is an integer, and for any $j \in [qk+1, (q+1)k]$, $switch_j$ equals to 1 indicates that k^{q+1} instances of *CounterIncrement* have been performed by some process. In other words, a process p locally performs k^{q+1} instances of *CounterIncrement* before attempting to set a switch in the interval $[qk+1, (q+1)k]$ and it increments its local threshold only if it knows that the last switch in this interval is set to 1 (i.e.; at least $k \cdot k^{q+1}$ instances of *CounterIncrement* have been performed). The threshold is multiplied by a factor of k. There is no guarantee that p will succeed in setting to 1 one of the switches. But in this case, sufficiently many increments have been performed by the processes so that a *CounterRead* operation can safely ignore the increments kept locally by p_i and still returns a value within a bounded factor of the actual number of increments.

By using test&set to modify a *switch* from 0 to 1, we ensure that the *CounterIncrement* instances accounted for by $switch_j$ are distinct from those accounted for by $switch_{j'}$, for any $j' \neq j$.

Performing an instance of a *CounterRead* operation *op* consists in traversing the sequence of switches until 0 is found. An approximation of the total number of *CounterIncrement* is then deduced from the index of the last $switch_j$ that *op* finds equal to 1. The value returned is the sum of the *CounterIncrement* operations represented by each switch from $switch_0$ to $switch_j$. In particular, $switch_0$ counts for one *CounterIncrement*, and each $switch_i$ in an interval $[qk+1, (q+1)k]$ for some integer $q \leq q_j$ counts for k^{q+1} *CounterIncrement* operations, where $switch_j$ belongs to the interval $[q_jk+1, (q_j+1)k]$.

The *CounterIncrement* operation: Each process i is equipped with two persistent local variables, $\texttt{lcounter}_i$ and $\texttt{limit}_i$. The former stores the number of *CounterIncrement*

instances performed by process i not yet announced to the other processes, and the latter stores the threshold on the number of *CounterIncrement* that can be performed by process i without informing the other processes.

When a *CounterIncrement* operation is invoked by a process i, $\texttt{lcounter}_i$ is first incremented (line 11). To ensure that a *CounterRead* operation instance returns a value that is within a multiplicative factor k of the actual number of increments, when $\texttt{lcounter}_i$ reaches a certain threshold stored in $\texttt{limit}_i$, process i tries to inform the other processes of the number of increments it has performed locally (lines 12). The value of $\texttt{limit}_i$ is initially 1 and is multiplied by k each time it is modified (line 21 and line 28). When $\texttt{lcount}_i = \texttt{limit}_i = k^{q+1}$ for some integer q, process i tries to set to 1 one of the k $switch_j$ whose index j is in the corresponding range $[qk+1,(q+1)k]$ (lines 15- 23). If it succeeds, it resets the local counter $\texttt{lcounter}_i$. The number of *CounterIncrement* instances it has performed locally has been announced to the other processes, and thus will be taken into account by future *CounterRead* operations.

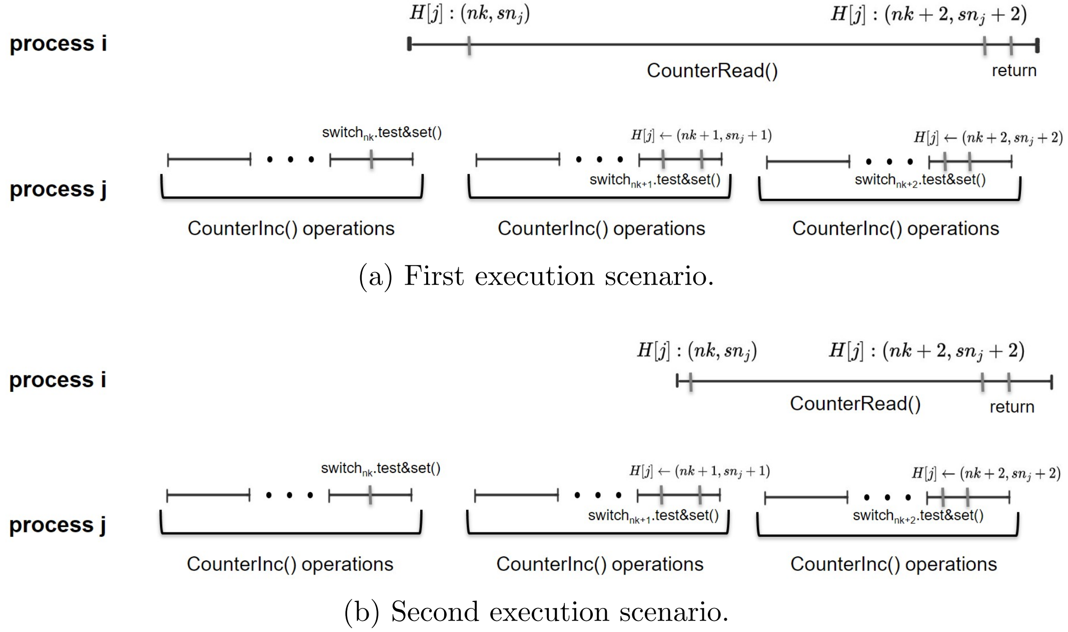
Additionally, process i writes the index of the switch it sets together with a sequence number into a shared variable $H[i]$ (lines 17 and 18). As explained later this pair is intended to help *CounterRead* operation instances to complete. Finally, the process will also update the value of the local persistent variable l_0 to indicate the index of the switch it managed to set within the interval (line 22). By doing so, we ensure that the process will avoid attempting to reset the same switches every time it reaches the threshold of $limit_i$ in the current interval by starting from the index $qk+l_0$ in the next attempt. If it does not succeed, every $switch_j$, where $j \in [qk+1,(q+1)k]$ is set. We show in the proof that for $k \geq n$, this number is sufficiently large for allowing *CounterRead* operations to return values within a factor k of the total number of *CounterIncrement* instances (Section 2.2.2). The threshold $\texttt{limit}_i$ is then multiplied by a factor k (line 28) and the value of l_0 is reset to 1 (line 24).

The *CounterRead* operation: When a *CounterRead* operation is invoked, process i scans the first and last *switch* of each interval of k switches, looking for the first one that is not yet set to 1. When such a switch is found, the index h of the last switch read that was equal to 1 is stored in the persistent local variable $\texttt{last}_i$ to avoid scanning the sequence from the beginning each time. We compute the value ret returned by the *CounterRead* operation in the function $ReturnValue(p,q)$ where $h = q \cdot k + p$ (line 30). First, we consider the required increments needed to set all the switches in the current interval $[qk+1,(q+1)k]$ by adding to ret the value $p \cdot k^{q+1}$ (line 31). Next, we add 1 to ret to account for the first $switch_0$ (line 31), and then for each previous interval $[(l-1)k+1, lk]$ where $1 \leq l \leq q$, we add k^{l+1} to ret (line 33). Finally, we return the computed value ret multiplied by a factor k to ensure ret falls in the approximation range of the k-multiplicative-accurate counter.

However, it may be the case that the condition at line 37 is never verified, as other processes may concurrently keep executing *CounterIncrement* operations. Thus, to ensure wait-freedom, we employ the following helping mechanism: a *CounterIncrement* operation by a process i that succeeds to set a $switch_j$, writes the index j of this switch together with a sequence number in the shared register $H[i]$ (lines 17 and 18). A *CounterRead* operation *op* that fails to find a switch to 0 after $\theta(n)$ steps, reads all the n shared registers $H[i]$ with $i \in 1,\ldots,n$. If a consistent value is found, then it returns at line 55. Otherwise, it executes another $\theta(n)$ steps. The first time *op* scan the array H,

Algorithm 1: k-multiplicative-accurate unbounded counter, pseudo-code for process i.

```
1  Shared variables
2    switch_j ∈ {0,1} : for each j ∈ ℕ, a 1-bit register that supports test&set and
     read primitives, initially all 0
3    H[n] : an array of n integer pairs (val, sn)
4  Persistent local variables
5    last_i ∈ ℕ_0 : largest index of a switch accessed by i, initially 0
6    lcounter_i: number of unannounced CounterIncrement by process i, initially
     0
7    limit_i : number of CounterIncrement that process i can perform locally,
     initially 1
8    sn_i: number of switches set to 1 by process i, initially 0
9    l_0: index of last switch accessed by the process i in the current set of
     switches, initially 1
10 Function CounterIncrement()
11   lcounter_i ← lcounter_i + 1
12   if lcounter_i = limit_i then
13     j ← log_k(lcounter_i)
14     if j > 0 then
15       for ℓ ← (j − 1)k + l_0, ..., j · k do
16         if switch_ℓ.test&set() = 0 then
17           sn_i ← sn_i + 1
18           H[i] ← (ℓ, sn_i)
19           lcounter_i ← 0
20           if ℓ = jk then
21             limit_i ← k · limit_i
22           l_0 ← 1 + ℓ mod k
23           return
24       l_0 ← 1
25     else
26       if switch_0.test&set() = 0 then
27         lcounter_i ← 0
28     limit_i ← k · limit_i
29   return
30 Function ReturnValue(p,q)
31   ret ← 1 + p · k^(q+1)
32   if q ≥ 1 then
33     ret ← ret + Σ_{l=1}^{q} k^(l+1)
34   return k · ret
35 Function CounterRead()
36   c ← 0
37   while switch_{last_i} ≠ 0 do
38     p ← last_i mod k
39     q ← ⌊last_i / k⌋
40     if last_i mod k = 0 then
41       last_i ← last_i + 1
42     else
43       last_i ← last_i + k − 1
44     c ← c + 1
45     if c mod n = 0 then
46       if c = n then
47         for j ← 1, ..., n do
48           help_i[j] ← H[j].sn
49       else
50         for j ← 1, ..., n do
51           (val, sn) ← H[j]
52           if sn − help_i[j] ≥ 2 then
53             p ← val mod k
54             q ← ⌊val / k⌋
55             return ReturnValue(p, q)
56   if last_i = 0 then
57     return 0
58   return ReturnValue(p, q)
```



(a) First execution scenario.

(b) Second execution scenario.

Figure 2.2: Example of two executions where a *CounterRead* operations returns through the helping mechanism.

it stores the sequence number read in each $H[j]$, denotes sn_j. When scanning H again, *op* will select a pair whose timestamp is greater than or equal to sn_j $+2$. This ensures, that the corresponding switch has been set by process j in the execution interval of *op*. This requirement is illustrated in Figure 2.2. In the first execution scenario in Figure 2.2a, it would be possible for the *CounterRead* operation to return after reading $H[j]$ for the first time, since the corresponding *CounterIncrement* is executed within the execution interval of the *CounterRead* operation. However, as depicted in Figure 2.2b, it is also possible for the *CounterIncrement* to have set its corresponding switch prior to the invocation of the *CounterRead* operation. Thus, to ensure the step of setting the switch is within the execution interval of the *CounterRead* operation, the operation is not allowed to return until the second update of $H[j]$.

2.2.2 Wait-freedom and Technical Lemmas

Let E be an execution of the k-multiplicative-accurate unbounded counter implemented in Algorithm 1.

Lemma 2.2.1. *Operations CounterIncrement and CounterRead are wait-free.*

Proof. Let op_r and op_w denote a *CounterRead* and *CounterIncrement* instance respectively in E. The number of steps taken during op_w is bounded since at most the process will attempt to set k switches during a call to *CounterIncrement* and there are no other loops or function calls in the *CounterIncrement* operation.

Suppose by contradiction that op_r does not terminate. Meaning that every bit $switch_\ell$ it reads has been set to 1. Since the bits are initially 0, there is at least one process q that infinitely often performs a successful test&set operation on these bits. Note that each time this occurs, q increments its sequence number $\texttt{sn}_q$ and reports the

new value in the helping array H (lines 17- 18). As every n iterations of the **while** loop, op_r scans the array H, it will eventually detect that the sequence number of q has been incremented at least twice, hence op_r terminates via the helping mechanism (lines 50-55). Therefore, operations *CounterIncrement* and *CounterRead* are wait-free. □

We continue with a few technical lemmas.

Lemma 2.2.2. *Switches are set to 1 in E in increasing order of their index, starting from* switch_0.

Proof. For each process p the initial value of $limit_p$ is 1 and of $counter_p$ is 0, thus the first *CounterIncrement* operation by process p applies a test&set primitive to $switch_0$ according to lines 11, 12, 13, and 27. We now prove that for any given process p and for any $j \geq 1$, p applies a test&set primitive (if any) on each of the switches with indexes in the interval $[(j-1) \cdot k + 1, \ldots j \cdot k]$ in an increasing order of their index, starting from $switch_{(j-1)\cdot k+1}$. First observe that for any process p, the initial value of l_0 is 1, and l_0 is set to 1 iff the value of $limit_p$ is multiplied by a factor k (lines 24,28 and lines 21,22). This implies that when a new j is computed at line 13, the value of l_0 is 1.

Then the first iteration of the **for loop** at line 15 starts at $l = (j-1) \cdot k + 1$. Also, the value of l is incremented by one at each iteration of the for loop at line 15 unless p successfully sets a $switch_{(j-1)k+i}$ with $i \in \{1, \ldots, k\}$. In this latter case, the value of l_0 is modified at line 22 and takes the value $i+1$ if $i < k$, or 1 otherwise (we reach the end of the set). If l_0 takes a value different from 1, that is $l \neq j \cdot k$, (otherwise, the claim is proved), then the *CounterIncrement* operation returns at line 23 without modifying the value of $limit_p$. Thus, in the execution of a successive *CounterIncrement* operation (if any), process p will apply the next test&set primitive (if any) to $switch_{jk+i+1}$ (because of lines 12, 13, 15).

The value of $limit_i$ is multiplied by k (and then the value of j is incremented by one) only after a process has applied a test&set primitive (both successfully or not) to the last switch in the current interval $[(j-1) \cdot k + 1, \ldots, j \cdot k]$ with $log_k(limit_i) = j$ (lines 21, 28). This completes the proof.

□

Lemma 2.2.3. *For any given execution E, if a CounterRead operation op returns the value computed in ReturnValue(p, q) at line 55, then $switch_{q\cdot k+p}$ was equal to 0 before the invocation of op and the test&set primitive that sets $switch_{q\cdot k+p}$ to 1 is applied during the execution interval of op.*

Proof. At line 51, *op* reads a pair (val, σ) from an entry $H[p']$ of the helping array H where $val = q \cdot k + p$. According to lines 16, 17, and 18, a unique process p' sets to 1 the $switch_{val}$ and associates with val the sequence number σ computed at line 17, before writing the pair (v, σ) to $H[p']$ in the execution of a *CounterIncrement* operation op'.

Let p be the process that executes the *CounterRead* operation *op*. Denote by σ' the value of $H[p'].sn$ read by p at line 48 in the execution of *op*. According to line 52, $\sigma - \sigma' \geq 2$. This means that process p' executes line 17 at least twice during the execution interval of *op*. In particular p' executes the step that set $switch_{val}$ to 1 after *op* was invoked by p. This proves the claim. □

2.2.3 Linearizability

We next define the linearization L of the operations in E by first removing any *CounterRead* operation that did not complete and any incomplete *CounterIncrement* operation that has not successfully executed line 16.

Let OP_W be the set of (complete and incomplete) *CounterIncrement* operations that successfully set a switch while executing line 16. Let OP_{LO} be the remaining complete *CounterIncrement* operations in E and OP_R be the set of complete *CounterRead* operations in E. Observe that each *CounterIncrement* operation successfully sets at most one switch, and each switch is successfully set by at most one process. Thus we can univocally associate each operation in OP_W with the switch it sets. We order the operations in $OP_W \cup OP_{LO} \cup OP_R$, according to the following rules :

1. We linearize each operation in OP_W at the step where it sets its corresponding switch. From claim 2.2.2, operations in OP_W are totally ordered and this order respects the real-time order. In the following, we denote opw_i the i-th operation in OP_W according to our linearization order with $i \geq 0$.
2. We linearize a *CounterRead* operation *opr* according to whether it returns normally or through the helping mechanism:
   (a) If *opr* returns $ReturnValue(p, q)$ normally at line 58, then it is linearized at the step where it reads the value 1 of $switch_{q \cdot k+p}$ at line 37 . This is well-defined because this read primitive exists and it is unique (it is easy to check from the pseudo-code).
   (b) If *opr* returns $ReturnValue(p, q)$ via the helping mechanism at line 55, then the operation is linearized immediately after $opw_{q \cdot k+p}$.
3. Let L_{WR} denote the linearization of all operations in $OP_W \cup OP_R$ according to rules 1 and 2, we linearize an operation *op* in OP_{LO} immediately before the first operation *op*′ in L_{WR} that follows *op* in the real-time order or at the end of L_{WR} if *op*′ does not exist.

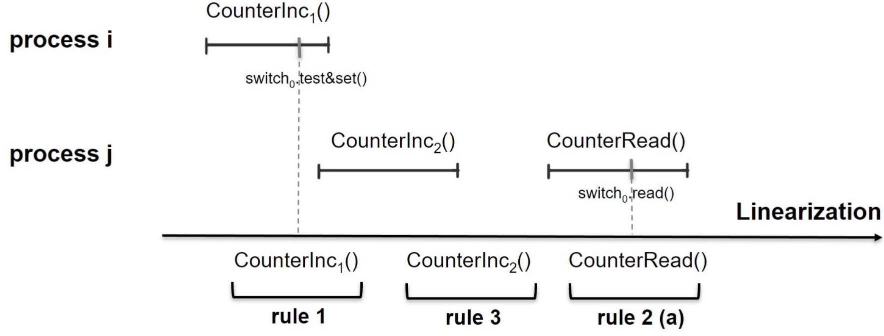
CounterRead operations that returns 0 after reading $switch_0 = 0$ are linearized before opw_0. If several operations are ordered at the same position, they are ordered respecting their real-time order. Figure 2.3a and Figure 2.3b give two examples of how the linearization rules are applied to different executions. Figure 2.3a describes the case where a *CounterRead* returns normally, and Figure 2.3b illustrates the case where a *CounterRead* needs to be linearized through the specific rule for the helping mechanism (rule 2 (b)).

Linearization rule 2 and Lemma 2.2.3 imply the following claim.

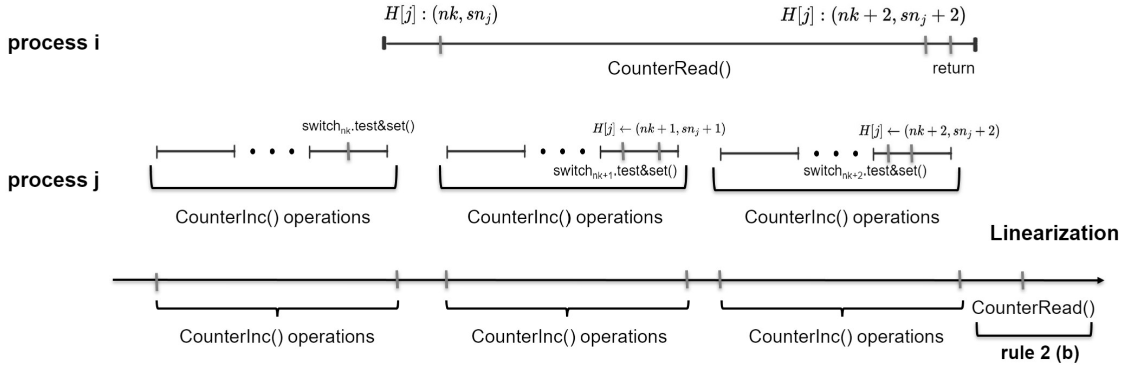
Claim 2.2.4. *Let opr be a CounterRead operation. We have that opr is linearized at some point after its invocation.*

Lemma 2.2.5 (Linearizability). *Algorithm 1 is a linearizable implementation of a k-multiplicative-accurate unbounded counter.*

Proof. Let op_1 and op_2 be two operations in E such as op_1 ends before op_2 is invoked. We prove that the linearization order L respects the real-time order, thus op_1 precedes op_2 in L. First, we have the following claim:



(a) Linearization of a simple execution following the proposed rules.



(b) Linearization of the execution in Figure 2.2a where a *CounterRead* operation returns through the helping mechanism.

Figure 2.3: Applications of the proposed linearization rules.

- Let op_1 and op_2 be two *CounterIncrement* operations. If at least one of these operations is in OP_{LO}, the claim trivially follows from rule 3. Otherwise, it is already proved in rule 1.

- Let op_1 and op_2 be two *CounterRead* operations. If both op_1 return normally the claim trivially holds from rule 2a and claim 2.2.4. So consider that op_1 returns through the helping mechanism and let $h_1 = q \cdot k + p$ be the index of the switch read by op_1 at line 54, the last time before returning. According to rule 2b, op_1 is linearized immediately after opw_{h_1}. Also, by Lemma 2.2.3 and rule 2, op_2 is linearized after opw_{h_1}. The claim follows since according to our linearization rules, If several operations are ordered at the same position, they are ordered respecting their real-time order.

- Consider that op_1 is a *CounterIncrement* and op_2 is a *CounterRead* operation. The claim follows from rules 1 and 2 and claim 2.2.4 (the reverse follows a similar reasoning).

The next claim will be useful for proving that the ordering L is consistent with the sequential specification of the k-multiplicative-accurate counter.

Claim 2.2.6. *Let op be a CounterRead operation invoked by a process p_i that returns $ReturnValue(p, q)$. The number of CounterIncrement operations linearized before op in*

L, denoted v, is at least $u_{min} = 1 + \sum_{l=1}^{q} k^{l+1} + p \cdot k^{q+1}$ *and at most* u_{max} *such that* $u_{max} \leq 1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1} + n(k^{q+1} - 1)$ *where n is the number of processes.*

Proof. Let *op* be a *CounterRead* operation invoked by a process p_i that returns $ReturnValue(p, q)$ and let $h = q \cdot k + p$ with $p \geq 0$. Consider the *CounterIncrement* operation by p_j that set to 1 the switch$_h$, denoted opw_h.

op is linearized at the step where it reads $switch_h$ if it returns normally, or immediately after opw_h. Thus, from our linearization rules, the minimal number of *CounterIncrement* operations that are linearized before *op* includes each opw_i in OP_W with $0 \leq i \leq h$, and every *CounterIncrement* in OP_{LO} linearized before *op*.

We have by construction that each $switch_s$ in the $(l+1)$-th set of k switches indexed in the interval $[l \cdot k + 1 \ldots (l+1)k]$ with $l \geq 0$, requires a process to perform k^{l+1} *CounterIncrement* operation instances before attempting to set $switch_s$ to 1. In other words, a process p_i needs its local variable $lcounter_i$ to be equal to k^{l+1} before it can attempt to set any $switch_s$ in $[l \cdot k+1 \ldots (l+1)k]$ (line 12). Since the value of $lcounter_i$ is reset to 0 after a successful *test&set* primitive is applied on a switch (line 19), the sets of *CounterIncrement* operation instances associated with any pair of successful *test&set* primitives are disjoint. Thus, $u_{min} = 1 + k\sum_{l=0}^{q-1} k^{l+1} + p \cdot k^{q+1} = 1 + k\sum_{l=1}^{q} k^{l} + p \cdot k^{q+1}$ since we account for, in addition to the p switches in the $(q+1)$-th set and $switch_0$, all k switches in each of the sets indexed from 1 to q.

Similarly, we compute an upper bound u_{max} on the maximum number of *CounterIncrement* linearized before *op*. First, suppose that *op* returns normally. As already said, *op* is linearized at the step where it reads $switch_h$ with $h = qk + p$. We have two possible cases either p is equal to 0 or it is equal to 1 because the process checks the first and last switch of each set during the *CounterRead*() instance. These two cases are depicted in Figure 2.4 a) and b) respectively. If p is equal to 0, then process p_i read $switch_{kq+1} = 0$ in the execution of *op*, and according to our linearization rules opw_{kq+1} is linearized after *op*. In a similar way, if p is 1, p_i read $switch_{(q+1)k} = 0$ and $opw_{(q+1)k}$ is linearized after *op*. However, in this second case, all the $k-1$ switches$_j$ with $j \in [q \cdot k+2 \ldots (q+1)k-1]$ may have been set to 1 before *op* applied its read to switch$_{qk+1}$, and all the corresponding opw_j may be linearized before *op*. Thus, the number of *opw* linearized before *op* is smaller than or equal to $1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1}$. It remains to count the number of *CounterIncrement* in OP_{LO} linearized before *op*. For every process p_i the value of $lcounter_i$ is smaller than k^{q+1} immediately before p read either $switch_{kq+1} = 0$ or $switch_{(q+1)k} = 0$ in the execution of *op*. Since a process resets the value of its local counter only when it succeeds to set a switch to 1 (line 19), $lcounter_i$ defines the number of *CounterIncrement* instances by p_i in OP_{LO} that are linearized before *op*. Therefore, $u_{max} \leq 1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1} + n(k^{q+1} - 1)$ where n is the number of processes. If *op* returns via the helping mechanism, then according to rule 2b, it is linearized immediately after $opw_{q \cdot k+p}$ with $0 \leq p < k$. Thus, $1 + \sum_{l=1}^{q} k^{l+1} + pk^{q+1}$ is the number of *CounterIncrement* in OP_W linearized before *op*. Since $p < k$ the local counter of every process immediately after $opw_{q \cdot k+p}$ sets the corresponding switch is smaller than k^{q+1}. Since $k > 1$, the claim follows. □

Let *op* be a *CounterRead* operation and let $v_{op} = ReturnValue(p, q)$ be the value it returns. According to lines 31, 33 and 34 of Algorithm 1, $v_{op} = k(1+\sum_{l=1}^{q} k^{l+1}+p \cdot k^{q+1})$; that is $v_{op} = k \cdot u_{min}$. According to claim 2.2.6, the number of *CounterIncrement* operations linearized before *op* in L, denoted u, is at least $u_{min} = 1+\sum_{l=1}^{q} k^{l+1}+p \cdot k^{q+1}$

and at most $u_{max} \leq 1 + \sum_{l=1}^{q} k^{l+1} + p(k-1)k^{q+1} + n(k^{q+1} - 1)$ (where n is the number of processes). And we have:

$$\frac{u_{max}}{k} \leq \frac{1}{k} + \sum_{l=1}^{q} k^l + p\frac{k-1}{k}k^{q+1} + \frac{n}{k}(k^{q+1} - 1)$$

$$\frac{u_{max}}{k} \leq \sum_{l=1}^{q} k^l + p \cdot k^{q+1} + n \cdot k^q$$

$$\text{And } v_{op} = k(1 + k\sum_{l=1}^{q} k^l + p \cdot k^{q+1})$$

Consider a short execution where $q = 0$, then $v_{op} = k(1 + p \cdot k)$ and $\frac{u_{max}}{k} \leq p \cdot k + n$. Therefore, for $k \geq n$, we have $\frac{u_{max}}{k} \leq v_{op}$.

Otherwise, if $q \geq 1$, we have the following

$$v_{op} = k(1 + k\sum_{l=1}^{q-1} k^l + k^{q+1} + p \cdot k^{q+1})$$

$$= k + k\sum_{l=2}^{q} k^l + p \cdot k^{q+2} + k^{q+2}$$

Thus, for $k \geq \sqrt{n}$, $\frac{u_{max}}{k} \leq v_{op}$.

Since $p < k$, $\forall q \geq 0$, we have $\frac{u}{k} \leq \frac{u_{max}}{k} \leq v_{op} \leq k \cdot u_{min} \leq k \cdot u$ for any $k \geq n$. This completes the proof. □

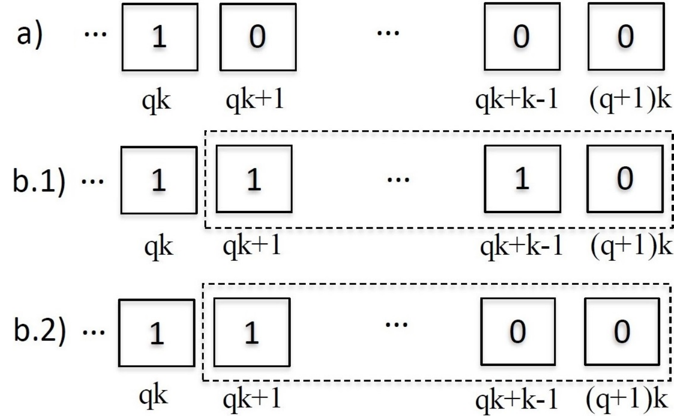


Figure 2.4: Switches state for the proof of claim 2.2.6. The dotted line indicates the $q+1$-th interval of consecutive switches. When $p = 1$, *op* does not distinguish between cases b.1) and b.2)

2.2.4 Complexity Analysis

Lemma 2.2.7. *If process p applies a test&set() primitive to a switch*$_\alpha$ *with* $i \cdot k + 1 \leq \alpha \leq (i+1) \cdot k$ *for some integer* $i \geq 0$*, then p has performed at least* k^{i+1} *CounterIncrement() operations.*

Proof. Suppose that p has executed a *test&set*() primitive to a switch$_\alpha$ with $i \cdot k + 1 \leq \alpha \leq (i+1) \cdot k$ in the execution of a *CounterIncrement*() operation *op*. According to line 15, j was equal to $i+1$ when computed at line 13, meaning that $\texttt{lcounter}_p$ was equal to k^{i+1}. The claim holds because $\texttt{lcounter}_p$ is incremented only at line 11, that is once for each *CounterIncrement*() operation performed by p. □

Lemma 2.2.8 (Amortized complexity)**.** *For $k \geq \sqrt{n}$, the amortized complexity of Algorithm 1 is constant.*

Proof. Let E be a finite execution of the unbounded k-multiplicative-accurate counter object implemented in Algorithm 1. Let r denote the number of *CounterRead*() instances in E and s be the number of *CounterIncrement*() instances in E. We additionally denote $Ops_W(E)$ the set of *CounterIncrement*() operations that execute at least one step in E, and $Ops_R(E)$ the set of *CounterRead*() operations in E. We want to compute

$$AmtSteps(E) = \frac{\sum_{op \in Ops_W(E) \cup Ops_R(E)} Nsteps(op, E)}{r+s}$$

where $Nsteps(op, E)$ is the number of steps executed by *op* in E.

Let $Ops_{W_p}(E)$ denote the *CounterIncrement*() operations in $Ops_W(E)$ executed by process p and s_p denote the total number of *CounterIncrement*() operations executed by process p. Let α_p be the index of the furthest switch accessed by a process p when executing any of the *CounterIncrement*() operations in $Ops_{W_p}(E)$. We have that $i_p \cdot k + 1 \leq \alpha_p \leq (i_p + 1) \cdot k$ for some integer $i_p \geq 0$ (the case where $\alpha_p = 0$ is trivial).

In the worst case, process p applies a *test&set*() primitive to switch$_h$ for every $h \in [0, \ldots \alpha_p]$ and one additional step to write into $H[p]$ (line 18) each time p successfully set one of those switches. On the other hand, by Lemma 2.2.7 if process p applies a *test&set*() primitive to the switch$_{\alpha_p}$, then it has performed at least $k^{i_{p+1}}$ *CounterIncrement*() operations. Therefore,

$$\sum_{op \in Ops_{W_p}(E)} Nsteps(op) \leq 2 \cdot (i_p + 1)k + 1$$

$$\text{And } s_p \geq k^{i_p+1}$$

Thus, the total number of steps executed by the set of all processes $\mathcal{P}$ in order to perform the *CounterIncrement*() operations in E is :

$$\sum_{op \in Ops_W(E)} Nsteps(op) = \sum_{p \in \mathcal{P}} \sum_{op \in Ops_{W_p}(E)} Nsteps(op)$$
$$\leq \sum_{p \in \mathcal{P}} 2 \cdot (i_p + 1)k + 1$$

$$\text{And } s = \sum_{p \in \mathcal{P}} s_p \geq \sum_{p \in \mathcal{P}} k^{i_p+1}$$

Now we consider the number of steps applied by each process to perform *CounterRead* operations. Let α be the index of the furthest switch set to 1 by any process in $\mathcal{P}$. If $\alpha = 0$ then the claim follows. Then suppose $i \cdot k + 1 \leq \alpha \leq (i+1) \cdot k$ for some

integer $i \geq 0$. For any sequence of switches with the index in $[j \cdot k + 1, \ldots, (j+1) \cdot k]$ with $0 \leq j \leq i$ a process p only reads the first and the last switch in such interval (i.e., $switch_{j \cdot k+1}$ and $switch_{(j+1) \cdot k}$). This is because at the beginning $last_p$ is equal to 0 and it is incremented by 1 if it is a multiple of k (at line 41), by $k-1$ otherwise (line 43). Also, $last_p$ is a persistent variable, thus a process p reads a given switch that has been set to 1 at most once. This implies that the total number (in all its *CounterRead* operations) of read primitives applied by a process p to the switches is less or equal to $2(i+2)$ (2 per each of the $i+1$ intervals, plus $switch_0$ and $switch_{\alpha+1}$). Furthermore, any *CounterRead*() operation executes $O(n)$ steps of the **for** loop at line 47 or line 50 once every n iterations of the **while** loop (when the condition of line 45 is satisfied). This means that the total number of steps executed by a process p when performing its *CounterRead*() operations is less or equal to $4(i+2)$. Thus,

$$\sum_{op \in Ops_R(E)} Nsteps(op) \leq \sum_{p \in \mathcal{P}_r} 4(i+2) \quad \leq 4(i+2) \cdot n_r$$

where $\mathcal{P}_r$ is the set of processes that have invoked at least one *CounterRead*() operation and n_r is the cardinality of $\mathcal{P}_r$. Consider $n_r > 0$, the other case is trivial. Therefore:

$$AmtSteps(E) \leq \frac{\sum_{p \in \mathcal{P}} 2(i_p+1)k + 1}{\sum_{p \in \mathcal{P}} k^{i_p+1} + r} + \frac{4(i+2) \cdot n_r}{s+r}$$

Furthermore, by lemma 2.2.7 the minimum number of instances of the *CounterIncrement*() operation executed to set the switch α is k^{i+1}. Thus,

$$AmtSteps(E) \leq \frac{\sum_{p \in \mathcal{P}} 2(i_p+1) + \frac{1}{k}}{\sum_{p \in \mathcal{P}} k^{i_p} + \frac{r}{k}} + \frac{4(i+2) \cdot n_r}{k^{i+1} + r}$$

We have $k^x \geq x+1$ for $k \geq e$ and $\forall x \in \mathbf{R}$, it follows:

$$\frac{\sum_{p \in \mathcal{P}} 2(i_p+1) + \frac{1}{k}}{\sum_{p \in \mathcal{P}} k^{i_p} + \frac{r}{k}} \leq \frac{\sum_{p \in \mathcal{P}} 2(i_p+1) + \frac{1}{k}}{\sum_{p \in \mathcal{P}} (i_p+1)}$$

If $i = 0$, and since $r \geq n_r$ we have:

$$\frac{4(i+2) \cdot n_r}{k^{i+1} + r} \leq \frac{8 \cdot n_r}{k+r} \quad \leq 8$$

If $i \geq 1$, because $n_r \leq n$ and $k^{i+1} \geq i \cdot k^2$ we have:

$$\frac{4(i+2) \cdot n_r}{k^{i+1} + r} \leq \frac{4(i+2) \cdot n}{i \cdot k^2 + r}$$

Resulting in an amortized complexity of $O(1)$ for $k \geq \sqrt{n}$. □

From Lemma 2.2.1, 2.2.5 and 2.2.8 we conclude:

Theorem 2.2.9. *Algorithm 1 is a wait-free linearizable implementation of a k-multiplicative-accurate unbounded counter with a constant amortized complexity for $k \geq n$.*

2.3 Bounded k-multiplicative-accurate Counter

We present a wait-free linearizable m-bounded k-multiplicative-accurate counter with a worst-case step complexity of $O(\log(\log m+1))$ for both the *CounterRead* and *CounterIncrement* operations. (Algorithm 2).

2.3.1 Algorithm Description

Algorithm 2: Implementation of a k-multiplicative m-bounded counter.

```
1  Shared variables
2    Switch[log(m) + 1] : array of test&set objects initialized to 0 and indexed
     from 0 to log m.
3    MaxSwitch : Max register object that stores the index of the furthest switch
     in Switch[] set to 1, initially −1.
4  Local persistent variables
5    lcounter : locally counts the number of increments, initially 0.
6    threshold : stores the current required number of increments to set a switch,
     initially 1 .
7    index : stores the value of the last switch accessed, initially −1 .

8  Function CounterIncrement()
9    lcounter ++
10   if lcounter == threshold then
11     index ++
12     if (index ≥ 1) then
13       threshold ← 2 × threshold
14     if Switch[index].test&set() == 0 then
15       MaxSwitch.MaxWrite(index)
16       lcounter ← 0
17       return
18     MaxSwitch.MaxWrite(index)
19     if index == 0 then
20       index ++
21       threshold ← 2 × threshold
22       if Switch[1].test&set() then
23         MaxSwitch.MaxWrite(1)
24         lcounter ← 0
25         return
26       MaxSwitch.MaxWrite(1)
27 Function CounterRead()
28   r ← MaxSwitch.MaxRead()
29   if r == −1 then
30     return 0
31   return k · 2^r
```

To implement the k-multiplicative-accurate m-bounded counter, we use an array $Switch[]$ containing $\log(m) + 1$ *test&set* objects indexed from 0 to $\log m$. Henceforth, we call these *test&set* objects `switches`. Each time one of these switches is set to 1, its index is stored in the max register object $MaxSwitch$. Depending on the index of each switch within the array, a certain number of *CounterIncrement* operations need to be invoked by a process before it attempts to set the switch to 1. To keep track of the number of invocations, each process has a local persistent variable denoted *lcounter*. And since the number of invocations evolves with the index of the switches, we also use the variable *threshold* to store the current number required for each process. Finally, in the variable *index*, each process stores the value of the last switch it executed a *test&set*() primitive on.

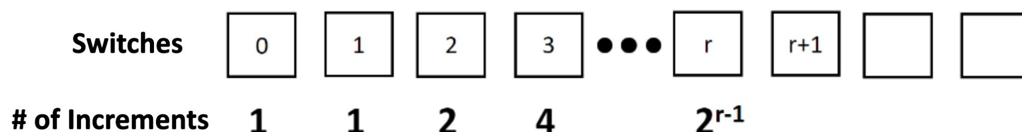


Figure 2.5: The array $Switch[]$ of *test&set* objects and the corresponding number of increments per switch.

Initially, the variable *threshold* is at 1. After a process invokes a *CounterIncrement* operation, it will increment its local counter *lcounter* and then compare it to the value of *threshold*. If the two match, the process will then increment the value of *index*, and if this value is greater or equal to 1, the process also doubles the value of its threshold. Then, the process attempts to set the switch at *index* in the array $Switch[]$. Regardless of whether it succeeds, the process will execute an instance $MaxWrite(index)$ on $MaxSwitch$. However, if it does succeed, the process will also reset the value of its local counter *lcounter* to 0 because it has informed the other processes of the increments by writing a switch. If the instance of *CounterIncrement* is the first instance invoked by the process, and it fails to set the switch with the index 0, then the process will repeat the steps for the switch with the index 1, since the two first switches both require a single *CounterIncrement* instance.

During an instance of *CounterRead*, a process simply reads the value r of $MaxSwitch$, and if $r > -1$, then it will return $k \cdot 2^r$. Otherwise, the process will return 0. We show that the return value falls within the approximation range defined by the sequential specification of the k-multiplicative-accurate counter.

2.3.2 Linearizability

Let E be an execution of the k-multiplicative-accurate m-bounded counter implemented in Algorithm 2. We construct a linearization L of E by removing some specific instances of the *CounterIncrememnt* and *CounterRead* operations, then ordering the remaining operations in E.

Let *op* be an incomplete *CounterIncrement* operation in E. We remove *op* from E in all but the following scenario: *op* succeeds in setting a switch of index i to 1, and the value of the max register $MaxSwitch$ reaches i during E. We also remove from E, any incomplete *CounterRead* operation.

From the remaining operations in E, we denote OP_w the set of *CounterIncrement* operations that set a switch to 1, and OP_l the set of remaining *CounterIncrement* operations. And let OP_r denote the set of *CounterRead* operations in E. We order the operations in $OP_w \bigcup OP_l \bigcup OP_r$ according to the following rules:

1. Let op denote a *CounterIncrement* operation in OP_w such that op sets to 1 the switch with the index r, and let op' be the first *CounterIncrement* operation to write r to $MaxSwitch$. Such an operation exists because we remove any incomplete *CounterIncrement* operation for which the index of the switch set by the operation is never written to $MaxSwitch$. op is linearized at the step of op' in which it writes r to $MaxSwitch$ (line 15, 18, 23, or 26 of Algorithm 2).

2. The *CounterRead* operations in OP_r are linearized at the step of reading the max register $MaxSwitch$ at line 28 of Algorithm 2.

3. We consider the partial order of the *CounterIncrement* operations in OP_l where for two operations op_1 and op_2 such that op_1 ends before op_2 is invoked, op_1 is before op_2 in OP_l.

   We linearize the operations in OP_l according to this partial order. op_1 will be linearized first according to the following rule: op_1 is linearized before the first operation already in L that follows op_1 in the real-time execution order, or at the end of L if such operation does not exist.

Lemma 2.3.1. *Let op denote a CounterIncrement operation in OP_w such that op sets to 1 the switch with the index r, and let op' be the first CounterIncrement operation to write r to $MaxSwitch$. The step executed by op' at line 15, 18, 23, or 26 of Algorithm 2, to write $MaxSwitch$ is executed within the execution interval of op.*

Proof. First, we prove that op' exists. Since we assume that op terminates (any operation that does not is removed from E), op will execute line 15, 18, 23, or 26 of Algorithm 2 to write r to the max register $MaxSwitch$. Therefore, there exists an operation op' in E that writes r to $MaxSwitch$.

If $op = op'$, the claim is trivial. We suppose that op' is different than op. As already mentioned, op will write r to $MaxSwitch$, and since op' is the first operation to do so, op' needs to write r to $MaxSwitch$ before op. Thus, op' invokes $MaxWrite(r)$ during the execution interval of op. □

Lemma 2.3.2 (Linearizability)**.** *Let op_1 and op_2 be two operations in E such that op_1 ends before op_2 is invoked. We have that op_1 precedes op_2 in L.*

Proof. We consider four separate cases depending on whether op_1 and op_2 are *CounterIncrement* or *CounterRead* operations:

- Let op_1 and op_2 be two *CounterIncrement* operations. If both operations are in OP_w, then they are linearized according to rule 1 at a point within their execution interval (Lemma 2.3.1). If both operations are in OP_l, then op_1 is linearized before op_2 according to the linearization rule 3 which follows the partial order of the operations in OP_l. Otherwise, consider that op_1 is in OP_w and op_2 in OP_l. op_1 is linearized first, then op_2 is inserted after op_1 according to linearization rule 3

since op_1 ends before op_2 begins. Similarly, if op_1 is in OP_l and op_2 is in OP_w, then op_2 will be linearized before op_1 since it is inserted before the first operation already in L that starts after op_1 ends.

- Let op_1 and op_2 be two *CounterRead* operations. From linearization rule 2, both op_1 and op_2 are linearized at line 28 of Algorithm 2. Since they are linearized at a point during their execution intervals, and we assume that op_1 ends before op_2 begins, the claim follows.

- Consider that op_1 is a *CounterIncrement* and op_2 is a *CounterRead* operation. If op_1 is in OP_w, then the claim follows since both op_1 and op_2 are linearized within their execution intervals (Lemma 2.3.1 and linearization rule 2). Suppose that op_1 is in OP_l. Based on linearization rule 3, it is inserted before the first operation already in L that ends before op_1 starts (or the end of L if such operation does not exist). Since op_1 also ends before op_2 starts, op_1 is linearized before op_2.

- Lastly, suppose that op_1 is a *CounterRead* and op_2 is a *CounterIncrement* operation. The same arguments from the previous case hold. If op_2 is in OP_w, both operations are linearized at a point during their execution intervals. Otherwise, op_2 is linearized before the first operation in L that starts after op_2 ends. Meaning that this operation is also linearized after op_1 since op_1 ends before op_2 begins. The claim follows.

□

Next, we show that the implementation respects the sequential specification of the k-multiplicative-accurate counter.

Lemma 2.3.3. *Each new value of $MaxSwitch$ during E is an increment by* 1 *of the previous value of $MaxSwitch$.*

Proof. Let E be an execution of Algorithm 2 and consider process p during E. We have that process p starts with a local threshold value of 1 stored in the variable *threshold*. Throughout E, each time p invokes enough *CounterIncrement* operations such that the value of its local counter matches the threshold, p increments the variable *index* by 1 and then eventually writes the new value of *index* to $MaxSwitch$. Each time p reaches a new threshold, this behavior repeats and the process only ever attempts to write the previous value of index plus one to the max register. Therefore, on the global scale of the execution, all processes will do the same and the new value of $MaxSwitch$ at any point during E is an increment by 1 of the previous value of $MaxSwitch$. □

Lemma 2.3.4. *Let op denote an instance of the CounterRead operation that returns x, and let v be the number of CounterIncrement operations before op in L. We have $v/k \leq x \leq k \cdot v$ for $k \geq \sqrt{n+1}$.*

Proof. Let r denote the value of $MaxSwitch$ read during *op* at line 28 of Algorithm 2. From Lemma 2.3.3, the values written to $MaxSwitch$ before *op* reads the value r, are increments of 1 starting from -1 to r. Since the max register is a linearizable object, the number of *MaxWrite* operations linearized before *op* is at least r. Therefore, the minimum number of *CounterIncrement* operations necessary to reach this value

of $MaxSwitch$ is $v_{min} = 1 + \sum_{j=1}^{r} 2^{j-1} = 2^r$. Indeed, to set the first two switches a single *CounterIncrement* instance is required for each. Afterward, the number of invocations required is multiplied by a factor of 2 each time it is reached. Furthermore, the maximum number of *CounterIncrement* operations invoked before *op* is $v_{max} = 1+\sum_{i=1}^{r} 2^{i-1}+n(2^r-1) = (n+1)\cdot 2^r-n$. The value corresponds to the minimum number of invocations required, and an additional $2^r - 1$ instances per process to represent the maximum number a process can count locally after the execution has reached the switch at the index r.

We have that *op* returns $x = k \cdot 2^r$, thus $v_{max}/k \leq x$. And we have $x \leq k \cdot v_{min}$ as long as $k \geq \sqrt{n+1}$. The claim follows. □

2.3.3 Complexity Analysis

We consider the m-bounded max register implementation given by Aspnes et al. [5] which has a step complexity of $O(\log m)$ for both $MaxWrite$ and $MaxRead$ operations.

Lemma 2.3.5. *A process executes* $O(\log(\log m+1))$ *steps during a call to the CounterRead or CounterIncrement operation.*

Proof. An instance of *CounterRead* calls the operation $MaxWrite$ once and then computes the return value. Similarly, the *CounterIncrement* operation calls the operation $MaxWrite$ a constant number of times and also computes a constant number of steps. Since, We use a $(\log m + 1)$-bounded max register in the implementation of the k-multiplicative m-bounded counter, the claim follows. □

2.4 Bounded k-multiplicative-accurate Max Register

Algorithm 3 represents an implementation of a k-multiplicative-accurate max register. The algorithm is wait-free, and has asymptotically optimal worst-case step complexity. Indeed, we prove later on in the chapter a matching lower bound.

The key idea of our algorithm is to consider the k-base representation of values written to the register and have `Write` operations store only the index of the bit preceding (i.e., to the left of) the most significant bit (MSB) of their arguments. These indices are stored in an (accurate) $\big((\lfloor \log_k(m-1)\rfloor) + 1\big)$-bounded max register implemented in a wait-free manner [5]. A *Read* operation R reads the value p of the accurate max register. If it equals 0 (implying that it was not written to yet), R returns 0. Otherwise, p is the largest index written so far to the accurate max register and R returns k^p. The pseudo-code is presented by Algorithm 3.

We now prove that Algorithm 3 is a correct wait-free implementation of a k-multiplicative-accurate max register.

Observation 2.4.1. *Algorithm 3 is a wait-free implementation of a k-multiplicative-accurate m-bounded max register.*

Proof. Follows directly from the wait-freedom of the max register algorithm of [5]. □

Algorithm 3: A k-multiplicative-accurate m-bounded max register

```
1  Shared variables
2  M: ((⌊log_k(m − 1)⌋) + 1)-bounded max register initially 0
3  Function Read()
4  |  p ⟵ M.read()
5  |  if p=0 then return 0;
6  |  else return k^p;
7  end
8  Function Write(v)
9  |  p ⟵ ⌊log_k v⌋ + 1;
10 |  M.write(p);
11 end
```

Lemma 2.4.2. *Algorithm 3 is a linearizable implementation of a k-multiplicative-accurate m-bounded max register.*

Proof. Let M_m^k denote a k-multiplicative-accurate m-bounded max register implemented by Algorithm 3 and let E be an execution of M_m^k. We now specify how operation instances on M_m^k in E are linearized. First, all the instances of `Read` that did not execute line 4 in E and all the instances of `Write` operations did not execute line 10 in E do not appear in the linearization. We say these are *removed operations*. Note that none of the removed operations has completed in E. For all remaining instances, we define the linearization point of a `Read` operation on M_m^k to be the linearization point of the *read* operation it invoked on M in E (in line 4) and the linearization point of a `Write` operation on M_m^k as the linearization point of the *write* operation it invokes on M (in line 10). Since each non-removed operation instance on M_m^k in E is linearized at a step it performs (hence during its execution interval), the linearization order we have define, denoted by L, respects the real-time order of the operation instances in E.

It remains to show that L satisfies the sequential specification of a k-multiplicative-accurate m-bounded max register. First note that since values written to M_m^k are from $\{1, ..., m-1\}$ and from lines 9-10, only values from $\{1, ..., \lfloor \log_k(m-1) \rfloor + 1\}$ are written to M. Let R denote a `Read` instance in L that returns 0 in line 5. Since only positive values are ever written to M, it follows that R is not preceded in L by any `Write` instance, hence the value of M_m^k when R is linearized is its initial value 0, so R returns the exact value of M_m^k.

Assume, then, that R *is* preceded in L by one or more `Write` instances and returns a positive value $x = k^p$ for some $p \geq 1$. We need to prove that $v/k \leq x \leq vk$ holds, where v is the maximum value written by any `Write()` instance linearized before R in L. Since M is linearizable and since we have linearized all non-removed instances applied to M_m^k in E according the order of the operations they applied to M (in line 4 or in line 10), there exists a `Write` operation that writes some value w and appears before R in L, such that $\lfloor \log_k w \rfloor = p - 1$ and p is the maximum value written to M by any `Write` instance that precedes R in L. Let $V = \{w \big| \lfloor log_k(w) \rfloor = p - 1\}$ be the set of all the values written to M_m^k in L before R whose MSB equals $p - 1$. Let $v = max(V)$. It follows that v is the maximum value written to M_m^k by any `Write()` instance linearized in L before R. We have $v \in [k^{p-1}, k^p - 1]$ and $x = k^p$. Consequently, $v \leq x \leq v \cdot k$ and the sequential specification of the k-multiplicative m-bounded max register is satisfied. □

Theorem 2.4.3. *Algorithm 3 is a wait-free linearizable implementation of a k-multiplicative-accurate m-bounded max register with worst case operation step complexity* $O\Big(min\big(\log_2(\log_k m), n\big)\Big)$.

Proof. Wait-freedom and linearizability follow from Observation 2.4.1 and Lemma 2.4.2, respectively. As for step complexity – the worst case operation step complexity of the wait-free implementation of an m-bounded max register of [5] is $O\big(min(\log m, n)\big)$ for both *Read* and *Write* operations. Each operation of Algorithm 3 applies a single operation on a $\big((\lfloor\log_k(m-1)\rfloor)+1\big)$-bounded max register and a constant number of additional steps. The theorem follows. $\square$

2.5 Unbounded k-multiplicative-accurate Max Register

We present in this section a wait-free linearizable implementation of the unbounded k-multiplicative-accurate max register with $O(log_2(log_k(m)))$ amortized step complexity, based on the bounded variant presented in Section 2.4.

2.5.1 Algorithm Description

We consider the implementation on an exact unbounded max register presented in [12] and we "plug-in" our bounded k-multiplicative-accurate max register into their construction to implement an unbounded k-multiplicative max register with amortized step complexity of $O(log_2(log_k(m)))$ for $m \geq n^2$.

The correctness of the resulting Algorithm 4 is guaranteed only in executions in which the max register's value is increased in bounded increments. This requirement is formalized by the following definition.

Definition 2.5.1 (**ℓ-Bounded-Increment Execution**). *Let E be an execution and let M be an unbounded k-multiplicative max register object. We say that E is an ℓ-bounded-increment execution for M if for each write operation $op = \texttt{Write}(v)$ on M in E, with $v > \ell$, there exists a write operation $op' = \texttt{Write}(v')$ on M in E that precedes op, such that $v - \ell \leq v' < v$.*

To implement the unbounded k-multiplicative-accurate max register, we rely on an infinite set of m-bounded k-multiplicative-accurate max registers (previously implemented in Section 2.4) denoted max_j for $j \in \mathbb{N}_0$. To each max_j is associated a 1-bit register denoted $switch_j$.

When a process invokes a $Write(v)$ instance, it will compute the index of the number of m-bounded max registers necessary to represent the value v. This is done by simply doing the computation $j \leftarrow \lfloor \frac{v}{m} \rfloor$. Then, the process will write the remainder of the division of v by m to the bounded max register max_j, if $switch_j == 0$ which signifies that the bound m has not been reached yet for max_j.

The process will also set $switch_{j-1}$ to 1. Because we consider a bounded-increment execution, all the switches with an index smaller than $j-1$ have also been set to 1 (the proof of this claim follows).

For the *Read* operation, the process traverses the set of switches until it finds the first one that has not been set to 1. Then, it reads the value v of the corresponding bounded max register and computes the return value $v + last_i \cdot m$ based on the index of the switch $last_i$. To ensure wait-freedom, we employ the helping mechanism introduced

by Baig et al. [12] and which we describe in detail in Section 2.2 where it is also used for the implementation of the unbounded k-multiplicative-accurate counter.

2.5.2 Linearizability and Wait-freedom

We show in this section that the implementation of the unbounded k-multiplicative-accurate max register is wait-free and linearizable and has an amortized step complexity in $O(log_2(log_k(m)))$ for $m \geq n^2$.

Claim 2.5.1. *All the switches $switch_j$ in Algorithm 4 are set to* 1 *in an increasing order starting from $switch_0$ for an m-Bounded-Increment execution.*

Proof. Let E denote an m-Bounded-Increment execution on the unbounded max register implemented in Algorithm 4 and let *op* denote a $Write(v)$ operation in E such as $j = \lfloor v/m \rfloor \geq 1$. During the execution of the lines 6 to 11 of the $Write()$ operation *op*, we know that $switch_{j-1}$ is going to be set to 1. Furthermore, because E is an m-Bounded-Increment execution, there exists another $Write(v')$ operation op' that was before *op* in E and such as $\lfloor v'/m \rfloor = j - 1$. During the execution of op', similarly to *op*, the $switch_{j-2}$ is set to 1. By recurrence on j, we therefore have that every switch from o to $j - 1$ is set to 1. □

Lemma 2.5.2. *Algorithm 4 is a linearizable implementation of a k-multiplicative unbounded max register.*

Proof. To prove the linearizability of the k-multiplicative unbounded max register, we consider Lemma 2 [12] which proves the linearizability of the unbounded max register. This proof guarantees the linearizability of the object under the assumption (*Claim 1*) that the values written to the register are not too far apart, ensuring that the switches are set to 1 consecutively. This condition is satisfied when $m \geq n$

We define a linearization order for all operations that terminated in E and remove any that have not finished. We start by defining the linearization point of the $Write()$ operations that execute line 12, and `Read()` operations that execute line 23 as the access point to the max register object. Then, a $Write()$ operation that do not access the max register object is positioned in L following the last linearized $Write()$ operation that precedes it in the execution order of E. Finally a `Read()` operation that invokes the `GetHelp()` function and does not access the max register object is positioned before the linearized $Write()$ instance that occurs afterwards in the execution order of E. We need to prove that this linearization L satisfies the sequential specification. Let *op* denote a `Read()` operation in L that returns $x = j \cdot m + r$. There exists a $Write()$ operation linearized before *op* and that writes t to the j-th max register such as $t/k \leq r \leq t \cdot k$, because the k-multiplicative m-bounded max register employed in the algorithm is linearizable and the value returned by *op* is either read directly from this max register or through the call to the `GetHelp()` function which accesses an array containing a copy of the max register value. Let OP_W be the set of $Write()$ operations linearized before *op* with such input values (i.e. $OP_W = \{Write(v), v = j \cdot m + t \text{ AND } t/k \leq r \leq t \cdot k\}$). And let $op' \in OP_W$ be the $Write()$ operation with the maximum input value u. We assume the existence of a $Write()$ operation linearized before *op* with an input value $w = h \cdot m + g$ such as $h > j$. The fact that this operation is linearized before *op* in

Algorithm 4: k-multiplicative unbounded max register based on Algorithm 1 [12]

```
1  Shared variables
2    switch_j ∈ {0,1} : a 1-bit register for each j ∈ ℕ_0, initially all 0.
3    max_j : a k-multiplicative m-bounded max register object for each j ∈ ℕ_0,
     initially all 0.
4    H[n] initially all (0,0) : a size n array storing tuples, H[i] used by process i to
     help other processes.
5  Local persistent variables
6    last_i ∈ ℕ_0 : stores the largest index j such that process i accessed max_j,
     initially 0.
7    sn_i, an integer counting the number of write operations done by process i,
     initially 0.

8  Function Write(v)
9      v' ⟵ v mod m;
10     j ⟵ ⌊v/m⌋;
11     if switch_j == 0 then
12         max_j.write(v');
13         if j > 0 then
14             curMax ⟵ max_{j-1}.read() + (j - 1) · m;
15             if switch_{j-1} == 0 then
16                 sn_i ⟵ sn_i + 1;
17                 H[i] ⟵ (sn_i, curMax);
18                 switch_{j-1} ⟵ 1;
19     last_i ⟵ max(j, last_i);
20 end
21 Function Read()
22     c ⟵ 0 ;
23     while switch_{last_i} ≠ 0 do
24         last_i ⟵ last_i + 1;
25         c ⟵ c + 1;
26         if c mod n == 0 then
27             if (hVal ⟵ GetHelp(c)) > 0 then
28                 return hVal;
29     end
30     v ⟵ max_{last_i}.read();
31     return v + (last_i · m);
32 end
```

Algorithm 5: The `GetHelp` utility function for process i. [12]

```
1  Local persistent variables
2    HR_i[n] : an array of integers, stores local copies of the i-th row of the H array.
3    SN_i[n] : an array of integers, counting the number of writes by each process
       that helps process i.
4  Function GetHelp(c)
5    if c == n then
6      for (j = 0; j < n; j + +) do
7        HR_i[j] <-- H[j] ;
8        SN_i[j] <-- HR_i[j].sn;
9      end
10   else
11     for (j = 0; j < n; j + +) do
12       HR_i[j] <-- H[j] ;
13       if HR_i[j].sn - SN[j] >= 2 then
14         return HR_i[j].val;
15     end
16   end
17   return 0;
18 end
```

L, ensures that line 5 of Algorithm 4 is executed before the `Read()` operation if it is a $Write()$ operation that modifies a max register object (we assume it is with no loss of generality because otherwise, there exists a previous $Write()$ operation in L with a larger than or equal input). Meaning that the return value of *op* would have to be $h \cdot m + w$ with $g/k \leq w \leq g \cdot k$ either from directly accessing the k-multiplicative m-bounded max register that corresponds to the h-th switch and not the j-th or from the return value from the call to the `GetHelp()` function, which contradicts the order of linearization in L. Furthermore, we have $u = j \cdot m + s$ and $s/k \leq r \leq s \cdot k$, therefore $j \cdot m + s/k < x = j \cdot m + r < j \cdot m + s/k$ satisfying the sequential specification of the k-multiplicative unbounded max register $u/k = (j \cdot m + s)/k \leq x \leq k \cdot u = (j \cdot m + s)k$. □

Lemma 2.5.3. *Algorithm 4 is an implementation of a k-multiplicative unbounded max register with an amortized step complexity of $O(log_2(log_k(m)))$ when $m \geq n^2$.*

Proof. The complexity of the k-multiplicative max register implemented by Algorithm 2 [12] is a direct result of the cost of the operations on the max register employed in the implementation (lines 2 and 7), we follow a similar reasoning to bound the amortized step complexity *AmtSteps* of the execution E of the k-multiplicative unbounded max register given by the following formula:

$$AmtSteps(E) = \frac{\sum_{op \in Ops(E)} Steps(op, E)}{|Ops(E)|}$$

With $Ops(E)$ the set of all operations that appear in E and $Steps(Op, E)$ the number of steps performed by an operation Op in E. Let $Ops_W(E)$ denote the set of w

$Write()$ operations and $Ops_r(E)$ the set of r `Read()` operations in E, and let $loop_{op}$ be the cost of the loop in the `Read()` operation. Furthermore, we note that the execution scenario of the `Read()` operation in which `GetHelp()` is invoked requires an additional cost of $O(n)$ steps. The call to $GetHelp()$ happens once every $k \cdot n$ steps for $k > 1$ when $c = 0 \bmod n$. Therefore, the number of steps taken during a `Read()` operation inside the `GetHelp()` function is $O(loop_{op})$. When substituting the exact m-bounded max registers objects with the k-multiplicative m-bounded max registers, the cost of accessing or modifying the max registers employed in the implementation drops from $log(m)$ to $log_2(log_k(m))$. Therefore we have:

$$AmtSteps(E) = O\Big((\sum_{op \in Ops_W(E)} \log(\log_k m) + \sum_{op \in Ops_R(E)} \log(\log_k m) + loop_{op})/(w+r)\Big)$$

If $r = 0$, then $AmtSteps(E) = O(\log(\log_k m))$ trivially, so assume that $r > 0$. From lines 16 and 17, for every process i, $last_i$ is never decreased and is incremented once in every iteration of the while loop, therefore:

$$\sum_{op \in Ops_R(E)} loop_{op} = O\Big(r + \sum_{i \in \mathcal{P}} last_i\Big).$$

Consequently,

$$AmtSteps(E) = O\Big((w \cdot \log(\log_k m) + r \cdot \log(\log_k m) + (r + \sum_{i \in \mathcal{P}} last_i))/(w+r)\Big).$$

Assume that max register $\mathtt{max}_\alpha$ is accessed in E. Since E is an n-bounded-increment execution and all $\mathtt{max}_\mathtt{J}$ registers are m-bounded, at least $m \cdot (\alpha - 1)/n \; Write()$ operations have completed prior to this access. Letting $\mathcal{L} = \max_{i \in \mathcal{P}} last_i$ denote the maximum value of all $last_i$ variables at the end of E, we get that $w \geq \frac{m}{n}(\mathcal{L} - 1)$. Furthermore, $\sum_{i \in \mathcal{P}} last_i \leq n \cdot \mathcal{L}$. Thus,

$$\begin{aligned} AmtSteps(E) &= O\Big(\frac{w \log(\log_k m) + r \log(\log_k m) + (r + n \cdot \mathcal{L})}{w + r}\Big) \\ &= O\Big(\log(\log_k m) + \frac{n \cdot \mathcal{L}}{\frac{m}{n}(\mathcal{L} - 1) + r}\Big) \\ &= O\Big(\log(\log_k m) + \frac{\frac{n^2}{m}\mathcal{L}}{(\mathcal{L} - 1) + \frac{n}{m} r}\Big) \end{aligned}$$

we have an amortized step complexity of $O(log_2(log_k(m)))$ for the unbounded k-mutliplicative max register when $r > 0$ and $m \geq n^2$. □

From Lemma 2.5.2, and 2.5.3 we have

Theorem 2.5.4. *Algorithm 4 is a wait-free linearizable implementation of a k-multiplicative unbounded max register with an amortized step complexity of $O(log_2(log_k(m)))$ when $m \geq n^2$.*

Proof. The proof for the property of wait-freedom of Algorithm 2 [12] still holds when we substitute the m-bounded max register with the k-multiplicative m-bounded max register since we prove this latter to be wait-free. The linearizability and complexity results are from Lemma 2.5.2 and Lemma 2.5.3 respectively. □

2.6 Worst-case Step Complexity Lower bound for k-multiplicative-accurate m-bounded Max Register and Counter

Aspnes et al. [7] proved a worst-case step complexity on the lower bound of a class of concurrent objects called *L-perturbable*, that includes objects such as max registers, counters and snapshots. L is called the *perturbation bound*. Roughly speaking, an object is L-perturbable if, for every implementation of the object, there exists an operation Op and an execution E, in the course of which Op is "perturbed" L times. An outstanding operation Op by process p is said to be perturbed by a process q, if a solo execution by q can change the response of a solo execution by p. They prove [7, Theorem 1] that any obstruction-free implementation of an L-perturbable object O from *historyless* primitives has an execution in which some process accesses $\Omega\big(min(\log_2 L, n)\big)$ distinct base objects during a single operation instance. Specifically, this implies that the worst-case step complexity of such implementations is $\Omega\big(min(\log_2 L, n)\big)$.

For the sake of presentation completeness, we restate the definition of an L-perturbable object from [7].

[5], Definition 2. *Let $\mathcal{I}$ be an obstruction-free implementation of an object. The set S_k of k-perturbing executions with respect to an operation instance op_n by process p_n is defined inductively as follows:*

1. *S_0 is the singleton set containing the empty sequence.*

2. *If $\alpha_{k-1}\lambda_{k-1}$ is in S_{k-1}, where λ_{k-1} consists of $n-1$ events, one by each of the processes $p_1, \ldots, p_{n-1}$, then $\alpha_{k-1}\lambda_{k-1}$ is in S_k. In this case, we say that $\alpha_{k-1}\lambda_{k-1}$ is* saturated.

3. *Suppose $\alpha_{k-1}\lambda_{k-1}$ is in S_{k-1}, no process has more than one event in λ_{k-1}, and there is a sequence γ of events by a process p_l different from p_n and the processes that have events in λ_{k-1}, such that the sequences of events by p_n as it performs op_n after $\alpha_{k-1}\lambda_{k-1}$ and $\alpha_{k-1}\gamma\lambda_{k-1}$ differ. Let $\gamma = \gamma' e \gamma''$, where e is the first event of γ such that the sequences of events taken by p_n as it performs op_n by itself after $\alpha_{k-1}\lambda_{k-1}$ and after $\alpha_{k-1}\gamma' e \lambda_{k-1}$ differ. Let λ be some permutation of the event e together with the events in λ_{k-1}, and let λ', λ'' be any two sequences of events such that $\lambda = \lambda'\lambda''$. Then the execution $\alpha_k\lambda_k$ is in S_k, where $\alpha_k = \alpha_{k-1}\gamma'\lambda'$ and $\lambda_k = \lambda''$.*

[5], Definition 3. *An obstruction-free implementation of an object is L-perturbable if there is an operation instance op_n such that the set S_L of L-perturbing executions with respect to op_n by p_n is nonempty.*

An object $\mathcal{O}$ is *perturbable* if all its obstruction-free implementations are perturbable.

[5], Theorem 1. *Let A be an n-process obstruction-free implementation of an L-perturbable object $\mathcal{O}$ from historyless primitives. Then A has an execution in which some process accesses $\Omega(\min(\log_2 L, n))$ distinct base objects during a single operation instance.*

Lemma 2.6.1. *A k-multiplicative-accurate m-bounded max register is $\Theta(log_k m)$-perturbable for $k > 1$.*

Proof. Let O be a k-multiplicative-accurate m-bounded max register and consider an obstruction-free implementation of O. We show that O is $(\frac{1}{2}log_k(m-1))$-perturbable for a `Read()` operation instance op_n by process p_n. We proceed by induction where the base case for $r = 0$ is immediate. Let $r < \frac{1}{2}log_k(m-1)$ and let $\alpha_{r-1}\lambda_{r-1}$ be an $(r-1)$-perturbing execution of O. If $\alpha_{r-1}\lambda_{r-1}$ is saturated, then it is also an r-perturbing execution. Otherwise, denote by v_{r-1} the maximum input to the `write()` operations linearized before op_n in the execution sequence $\alpha_{r-1}\lambda_{r-1}$. Since $\alpha_{r-1}\lambda_{r-1}$ is not saturated, there exists a process $p_l \neq p_n$ that does not take steps in λ_{r-1}. Let γ be the execution fragment by p_l where it finishes any incomplete operation in α and then performs a `write()` operation to the max register with the value $v_r = k^2 v_{r-1} + 1$. Then op_n must return a value x such that $kv_{r-1} < v_r/k \leq x \leq kv_r$ when run after $\alpha_{r-1}\gamma\lambda_{r-1}$. It follows that an r-perturbing execution can be constructed from $\alpha_{r-1}\lambda_{r-1}$ and γ as specified by [5], Definition 2. Because O is an m-bounded max register, during the rth step of the induction, the value written to the max register must satisfy $v_r \leq m-1$. Consequently it suffices to have:

$$v_r \leq (k+1)^{2r} \leq m-1 \implies r \leq \frac{1}{2}log_{k+1}(m-1) = \Theta(log_k m)$$

□

from Lemma 2.6.1 and [5], Theorem 1 we have the following theorem:

Theorem 2.6.2. *The worst-case step complexity of a k-multiplicative m-bounded max register is $\Omega\big(min(\log_2(\log_k m), n)\big)$*

Lemma 2.6.3. *A k-multiplicative-accurate m-bounded counter is $\Theta(log_k(m))$-perturbable for $k > 1$.*

Proof. Let O be a k-multiplicative m-bounded counter and consider an obstruction-free implementation of O. We show that O is $(\frac{1}{2}log_k(m-1))$-perturbable for a *CounterRead*() operation instance op_n by the process p_n. We proceed by induction where the base case for $r = 0$ is immediate. Let $\alpha_{r-1}\lambda_{r-1}$ be an $(r-1)$-perturbing execution of O. If $\alpha_{r-1}\lambda_{r-1}$ is saturated, then it is also an r-perturbing execution. Otherwise, let I_r denote the number of *CounterIncrement*() operation instances performed by the perturbing process in iteration r. We have that $I_1 = 1$ in order for op_n to return a value greater than 0. For $r > 1$, if op_n runs after $a_{r-1}\lambda_{r-1}$ it can return a value that is as large as

$k \cdot \sum_{j=1}^{r-1} I_j$. Therefore, we need the number of complete *CounterIncrement*() operation instances after $a_{r-1}\gamma\lambda_{r-1}$ to be at least $k^2 \cdot \sum_{j=1}^{r-1} I_j + 1$ for op_n to return a value greater than $k \cdot \sum_{j=1}^{r-1} I_j$.

Besides the *CounterIncrement*() operation instances in γ, at least $\sum_{j=1}^{r-1} I_j - (r-1)$ have finished, therefore setting $I_r = (k^2-1) \cdot \sum_{j=1}^{r-1} I_j + r$ implies that op_n returns at least $\frac{1}{k}(\sum_{j=1}^{r-1} I_j - (r-1) + I_r) = \frac{1}{k}(\sum_{j=1}^{r-1} I_j - (r-1) + (k^2-1) \cdot \sum_{j=1}^{r-1} I_j + r) = \frac{1}{k}(k^2 \cdot \sum_{j=1}^{r-1} I_j + 1)$ which is greater tha $k \cdot \sum_{j=1}^{r-1} I_j$ as needed.

$$
\begin{aligned}
I_r &= \sum_{i=0}^{r-1} (r-i)(k^2-1)^i = \sum_{i=1}^{r} i \cdot (k^2-1)^{r-i} \\
&= (k^2-1)^r \sum_{i=1}^{r} \frac{i}{(k^2-1)^i} \\
&= \frac{(k^2-1)((k^2-1)^r - 1) + r(2-k^2)}{(k^2-2)^2} \leq k^{2r} \leq m \\
&\implies r \leq \frac{1}{2} log_k(m) = \Theta(log_k m)
\end{aligned}
$$

□

From Lemma 2.6.3 and [5], Theorem 1, we prove the following Theorem

Theorem 2.6.4. *The worst-case step complexity of a k-multiplicative m-bounded counter is* $\Omega\big(min(\log_2(log_k m), n)\big)$

2.7 Amortized Step Complexity Lower bound for k-multiplicative-accurate Counter

In this section, we prove that the total step complexity of solo-terminating implementations of k-multiplicative accurate counters is $\Omega(n \log_{2q+1} \frac{n}{k^2})$ for $k \leq \sqrt{n/2}$, assuming the implementation uses base objects that support only read, write and either reading or regular conditional primitives of arity q or less.

For sake of completeness, in the following, we remember the definitions and the statement of lemmata presented in [10] that are used to prove our lower bound. In particular, only Lemma 2.7.2, Corollary 2.7.2.1, Lemma 2.7.4 and Theorem 2.7.5 differ from the original work.

Preliminaries

From now on, execution fragments are defined as (finite or infinite) sequences of events, with the understanding that each execution fragment is the projection of a single corresponding sequence of steps.

If a process has not completed its operation instance, it has exactly one *enabled* event, which is the next event it will perform, as specified by the algorithm it is using to apply its operation instance to the implemented object. We say that an execution E is *quiescent* if every instance that starts in E completes in E.

Processes communicate with one another by issuing events that apply *read-modify-write* (RMW) primitives to vectors of base objects. We assume that a primitive is always applied to vectors of the same size. This size is called the *arity* of the primitive. RMW primitives of arity 1 are called *unary* or *single-object* RMW primitives. RMW primitives of arity larger than 1 are called *multi-object RMW primitives.* For presentation simplicity we assume that all the base objects to which a primitive is applied are over the same domain. A RMW primitive, applied to a vector of k base objects over some domain D, is characterized by a pair of functions, $\langle g, h\rangle$, where g is the primitive's *update function* and h is the primitive's *response function.* The update function $g : D^k \times W \to D^k$, for some input-values domain W, determines how the primitive updates the values of the base objects to which it is applied.

In the following definitions, when we refer to an event as issued *after execution* E, we mean it is issued immediately after execution E. Similarly, when we refer to the state of an object after execution E, we refer to its state immediately after E. Let e be an event, issued by process p after execution E, which applies the primitive $\langle g, h\rangle$ to a vector of base objects $\langle o_1, \ldots, o_k\rangle$. Then e atomically does the following: it updates the values of objects $o_1, \ldots, o_k$ to the values of the components of the vector $g(\langle v_1, \ldots, v_k\rangle, w)$, respectively, where $\overrightarrow{v} = \langle v_1, \ldots, v_k\rangle$ is the vector of values of the base objects after E, and $w \in W$ is an input parameter to the primitive. We call $\overrightarrow{v}$ the *object-values vector* of e after E. The RMW primitive returns a response value, $h(\overrightarrow{v}, w)$, to process p. If W is empty, we say that the primitive *takes no input.*

A *k-compare-and-swap* (k-CAS), for some integer $k \geq 1$, is an example of a RMW primitive.

Next, we revise the concept of conditional synchronization primitives.

Definition 2.7.1. *A RMW primitive $\langle g, h\rangle$ is* conditional *if, for every possible input w, $\left|\{\overrightarrow{v} | g(\overrightarrow{v}, w) \neq \overrightarrow{v}\}\right| \leq 1$. Let e be an event that applies the primitive $\langle g, h\rangle$ with input w. The* change point *of e is the unique vector $\overrightarrow{c_w}$ such that $g(\overrightarrow{c_w}, w) \neq \overrightarrow{c_w}$; any other vector is a* fixed point *of e.*

In other words, a RMW primitive is a conditional primitive if, for every input w, there is at most one vector $\overrightarrow{c_w}$ such that $g(\overrightarrow{c_w}, w) \neq \overrightarrow{c_w}$. k-CAS is a conditional primitive for any integer $k \geq 1$. The single change point of a k-CAS event with input $\langle old_1, \ldots, old_k, new_1, \ldots, new_k\rangle$ is the vector $\langle old_1, \ldots, old_k\rangle$. Read is also a conditional primitive, since read events have no change points.

The next definition captures the extent to which processes are aware of the participation of other processes in an execution. Intuitively, a process p is aware of the participation of another process q in an execution if there is information flow from q to p in that execution; that is, p reads a shared-memory value that was either directly written by q or indirectly influenced by a value written by q. The following definitions formalize this notion.

Definition 2.7.2. *Let e_q be an event by process q in an execution E, which applies a non-trivial primitive to a vector v of base objects. We say that an event e_p in E by process p is* aware *of e_q if e_p accesses a base object o such that at least one of the following holds:*

- *There is a prefix E' of E such that e_q is visible on o in E' and e_p is a RMW event that applies a primitive other than write to o, and it follows e_q in E', or*

- *there is an event e_r that is aware of e_q in E and e_p is aware of e_r in E.*

If an event e_p of process p is aware of an event e_q of process q in E, we say that p is aware *of e_q and that e_p is* aware *of q in E.*

The following definition quantifies the extent to which a process is aware of the participation of other processes in an execution.

Definition 2.7.3. *Process p is* aware *of process q after an execution E if either $p = q$ or p is aware of an event of q in E. The* awareness set *of p after E, denoted $AW(E, p)$, is the set of processes that p is aware of after E.*

We use the following technical definition and lemma.

Definition 2.7.4. *Let $S = \{e_1, \cdots, e_k\}$ be a set of events by different processes that are enabled after some execution E, each about to apply write or a conditional RMW primitive. We say that an ordering of the events of S is a* weakly-visible schedule of S after E*, denoted by $\sigma(E, S)$, if the following holds. Let $E_1 = E\sigma(E, S)$, then*

1. *at most a single event of S is visible on any one object in E_1. If $e_j \in S$ is visible on a base object in E_1, then e_j is issued by a process that is not aware of any event of S in E_1,*
2. *any process ia aware of at most a single event of S in E_1, and*
3. *all the read events of S are scheduled in $\sigma(E, S)$ before any event of $\sigma(E, S)$ changes a base object.*

Weakly-visible schedules are used in the sequel for constructing executions that slow down the rate in which processes become aware of other processes. The following lemma shows that every set of outstanding write and conditional events has a weakly-visible schedule.

Lemma 2.7.1. *Let $S = \{e_1, \cdots, e_k\}$ be a set of events by different processes that are enabled after some execution E, each about to apply write or a conditional RMW primitive. Then there is a weakly-visible schedule of S after E.*

Lower bound

The key intuitions behind the following lower bound proofs are that first, in any n-process execution of a k-multiplicative accurate counter implementation, 'many' processes need to be aware of the participation of 'many' other processes in the execution, and second, if processes only use read, write and conditional primitives, then a scheduling adversary can order events so that information about the participation of processes in the computation accumulates 'slowly'. We use Definitions 2.7.2 and 2.7.3, as well as Lemma 2.7.1, to capture this intuition.

The following lemma proves a relation between the value returned by a *CounterRead* operation instance of a process in some execution and the size of that process' awareness set after that execution.

Lemma 2.7.2. *Let E be an execution of a solo-terminating k-multiplicative accurate counter object implementation where each process executes one instance of the CounterIncrement() operation followed by one instance of the CounterRead() operation. If the CounterRead() instance by a process p returns i in E then $|AW(E,p)| \geq \frac{i}{k}$.*

Proof. Assume, by way of contradiction, that there is an execution E where each process executes one instance of the *CounterIncrement*() operation followed by one instance of the *CounterRead*() operation, and a process p such that a *CounterRead*() instance by p, namely *op*, returns i and $|AW(E,p)| < \frac{i}{k}$.

We construct a new execution E' as follows: for any process $q \notin AW(E,p)$, we first remove all the events of q from E; then, for any process q', we remove all the events of q' that are aware of q. Note that if an event $e_{q'}$ of q' is aware of q, then all following events by q' are also aware of q and are removed. Also, no events of p are removed since p is aware only of processes in $AW(E,p)$.

We prove that E' is an execution, and that it is indistinguishable from E. We consider events in the order they appear in E'. Let e'_q be an event by process q' that appears in E', namely $E' = E'_1 e'_q E'_2$. Since e'_q is also in E, we can also write $E = E_1 e'_q E_2$. For the induction, assume that E'_1 is an execution and that it is indistinguishable to every process that appears in it from E_1. In particular, q' does not distinguish between E'_1 and E_1 and takes the same step after both of them. To see why q' obtains the same response in e'_q after E'_1 and after E_1, note that it can return a different response only if in E, e'_q is aware of an event e that was removed from E_1. This happens only if e is aware of some process $q \notin AW(E,p)$, meaning that in E, e'_q is also aware of q, contradicting the fact that e'_q was not removed. Hence $E'_1 e'_q$ is an execution and q' does not distinguish between $E'_1 e'_q$ and $E_1 e'_q$.

This implies that the *CounterRead*() instance by p returns i also in E'; on the other hand, less than $\frac{i}{k}$ processes participate in E'. Let E" be the extension of E' in which the processes that participate in E' complete their operation instances, one at a time. This execution exists by solo-termination, and results in a quiescent execution. However, less than $\frac{i}{k}$ instances of *CounterIncrement*() operations completed in E", and we have that p returns i when invoking *op*. Thus, the response of the *op* is not linearizable. In particular, consider any linearization L of E" and let v be the number of *CounterIncrement*() instances linearized before *op* in L, we have that $\frac{v}{k} \leq i \leq k \cdot v < k \cdot \frac{i}{k} = i$. □

Similar to Corollary 6 in [10], the following corollary is an immediate consequence of Lemma 2.7.2.

Corollary 2.7.2.1. *Let E be a quiescent n-process execution of a solo-terminating k-multiplicative counter implementation, where each process executes one instance of the CounterIncrement() operation followed by one instance of a CounterRead() operation. Then, the awareness sets of $\frac{n}{2}$ processes contain at least $\frac{n}{2k^2}$ other processes after E.*

Proof. Let L denote any linearization of E, and let *op* be the i-th *CounterRead*() instance in L. Since *op* is the i-th instance of *CounterRead*() in L, it returns v such as $v \geq \frac{i}{k}$. By considering the last $\frac{n}{2}$ processes linearized and by Lemma 2.7.2, the claim follows. □

Information about processes that participate in an execution is transferred through base objects. The following definition quantifies the number of other processes a process can become aware of when it reads a base object.

Definition 2.7.5. *Let E be an execution, o be a base object, and q be a process. We say that o* has record *of q after E if there is an event e, visible on o in E, such that the following hold:*

1. $E = E_1 e E_2$,
2. *e is an application of a non-trivial primitive to an objects-vector that contains o by some process r such that $q \in F(E_1 e, r)$.*

The familiarity set *of o after E, denoted $F(E, o)$, contains all processes that o has record of after E.*

Definition 2.7.6. *Let E be an execution. We let $\mathcal{M}(E) = \max_{p,o}(\{|AW(E,p)| \big| p \in \boldsymbol{P}\} \cup \{|F(E,o)| \big| o \in \boldsymbol{B}\})$ denote the maximum size of a process awareness set and object familiarity set after E.*

Definition 2.7.7. *Let $\mathcal{P}$ be a set of synchronization primitives. We say that $\mathcal{P}$ is* c-bounded*, for some constant c, if for every execution E and for every set S of events that are enabled after E, applying primitives from $\mathcal{P}$, there is a schedule σ of S such that $\mathcal{M}(E\sigma)/\mathcal{M}(E) \leq c$ holds.*

From Definition 2.7.7, it is clear that the smaller c is, the more can a scheduling adversary slow down the rate in which processes become aware of others.

Lemma 2.7.3. *The set of primitives that contains write and all the conditional primitives of arity c or less is $(2c+1)$-bounded.*

Lemma 2.7.4. *Let A be an n-process solo-terminating implementation of a k-multiplicative counter from base objects that support only primitives from a c-bounded set $\mathcal{P}$ and $0 < k \leq \sqrt{n/2}$. Then A has an execution E that contains $\Omega(n \log_c \frac{n}{k^2})$ events, in which every process performs a single CounterIncrement() instance and a single CounterRead() instance.*

Proof. We construct an n-process execution, E, with $\Omega(n \log_c \frac{n}{k^2})$ events, in which every process performs a single *CounterIncrement*() instance and a single *CounterRead*() instance. The inductive construction proceeds in rounds, indexed by the integers $1, 2, \cdots, r$, for some $r \in N$, and it maintains the following invariant: before round i starts, the size of the awareness set of any process and the size of the familiarity set of any base object is at most c^{i-1}.

If a process p has not completed its operation instances before round i starts, we say that p is *active in round i*. All processes are active in round 1. All the processes that are active in round i have an enabled event in the beginning of round i. We denote the set of these events by S_i. We denote the execution that consists of all the events issued in rounds $1, \ldots, i$ by E_i. We also let E_0 denote the empty execution.

For the induction base, note that, before execution starts, objects have no record of processes and processes are only aware of themselves. Thus $\mathcal{M}(E_0) = 1$ holds.

For the induction step, assume that $\mathcal{M}(E_{i-1}) \leq c^{i-1}$ holds. Since $\mathcal{P}$ is c-bounded, there is an ordering σ_i of the events of S_i such that $\mathcal{M}(E_{i-1}\sigma_i) \leq c\mathcal{M}(E_{i-1}) \leq c^i$. We let $E_i = E_{i-1}\sigma_i$.

By Corollary 2.7.2.1, the awareness sets of $\frac{n}{2}$ processes contain at least $\frac{n}{2k^2}$ other processes after E with $1 \leq \frac{n}{2k^2} \leq n$, meaning that $k \leq \sqrt{n/2}$. Therefore, each of these processes is active in at least the first $log_c(\frac{n}{2k^2} - 1)$ rounds, performing at least $log_c(\frac{n}{2k^2} - 1)$ events in E. □

Our step complexity lower bound is immediate from Lemma 2.7.4 and Lemma 2.7.3.

Theorem 2.7.5. *Let A be an n-process solo-terminating implementation of a k-multiplicative counter from base objects that support only read, write and either reading or regular conditional primitives of arity q or less. Then A has an execution E that contains $\Omega(nlog_{q+1}(n/k^2))$ events for $k \leq \sqrt{n/2}$, in which every process performs a single CounterIncrement() instance and a single CounterRead() instance.*

2.8 Discussion

We have presented upper and lower bounds on the step complexity of a variant of deterministic approximate counters and max registers.

Specifically, we presented a wait-free linearizable k-multiplicative-accurate counter for $k \geq n$ with constant amortized step complexity. While the condition on the approximation parameter k is necessary to ensure the return value of *CounterRead* operations remains valid for executions of any length, it is worth noting that for executions where more than $1 + n(k-1)$ *CounterIncrement* operations are executed prior to the first *CounterRead* operation, the condition lessens to $k \geq \sqrt{n}$.

We also show that by bounding the execution, we are able to implement the k-multiplicative-accurate counter for $k \geq \sqrt{n}$ in a wait-free linearizable manner and with a worst-case step complexity of $O(min(\log(\log(m+1)), n))$. The step complexity of our implementation approaches the lower bound on the worst-case complexity implementation of an m-bounded k-multiplicative-accurate counter which we prove to be $\Omega(min(\log(\log_k m), n))$.

We have also proved the possibly counter-intuitive result that when the accuracy parameter k does not depend on n, relaxing counter semantics by allowing inaccuracy of a multiplicative factor cannot asymptotically reduce the amortized step complexity of unbounded counters by more than a logarithmic factor.

The behavior of the counter in an unbounded relaxed setting for a parameter $k \in]\sqrt{(\frac{n}{2})}, n[$ remains an open question. The maximum improvement in the worst-case step complexity of the bounded variant of k-multiplicative-accurate counters remains an open question. Also, when k is constant, it is unclear whether there exists a deterministic wait-free k-multiplicative-accurate counter implementation with $o(\log^2 n)$ amortized step complexity.

We also show that relaxing the semantics of max registers by allowing inaccuracy of even a constant multiplicative factor yields an exponential improvement in the worst-case step complexity of the bounded variant and in the amortized step complexity of the unbounded one.

Chapter 3

Efficient Queue Implementations

Abstract

Despite the widespread usage of `FIFO` queues in distributed applications, designing efficient `wait-free` implementations of queues remains a challenge. Although the literature contains a variety of FIFO queue implementations, the vast majority rely on concurrency constraints: for a given implementation, not all processes are allowed to execute either/or *Enqueue* and *Dequeue* operations.

These restrictions on the number of dequeuers or the number of enqueuers that can operate on the queue hold even when the implementations use strong synchronization primitives, like the *Compare&Swap*.

The best upper bound for a multiple enqueuer wait-free FIFO queue implementation is given by Jayanti and Petrovic in [28] where both the *Enqueue* and *Dequeue* operations are in $O(\log n)$ with n the total number of processes. However, their implementation risks violating the sequential specification of the queue for executions with multiple dequeuer processes because multiple *Dequeue* operations might return the same element. If we do not limit the number of processes that can perform enqueue and dequeue operations, the best-known upper bound on the worst-case step complexity for a wait-free queue is given by Khanchandani and Wattenhofer [30]. In particular, they present an implementation of a multiple dequeuer multiple enqueuer wait-free queue whose worst-case step complexity is in $O(\sqrt{n})$, where n is the number of processes.

In this work, we investigate whether it is possible to improve this bound. In particular, we are interested in a logarithmic worst-case step complexity wait-free implementation that does not suffer from concurrency constraints. Therefore, we present a wait-free FIFO queue implementation that supports n enqueuers and k dequeuers where the worst case step complexity of an *Enqueue* operation is in $O(\log n)$ and where the complexity of the *Dequeue* operation depends on the level of concurrency during the execution and is $O(k \log n)$ in the worst-case scenario where all dequeuer processes are concurrent at a certain point during the execution.

We then rely on the relaxation of the FIFO queue semantics to show that allowing concurrent *Dequeue* operations to retrieve the same element results in an implementation with $O(\log n)$ worst-case step complexity for both the *Enqueue* and *Dequeue* operations.

An iteration of this work was presented during the 2022 *Conference On Principles Of Distributed Systems* (OPODIS).

3.1 Introduction

Shared FIFO queues are an important building block for the design of many concurrent applications.

So in order to have high-performing applications, it is crucial to have efficient implementations of the FIFO queue. These implementations also need to satisfy system-wide progress in the case of a failure. Oftentimes, implementations are content with the non-blocking condition of *lock-freedom* which allows individual threads to starve but guarantees system-wide progress. Imposing the stricter guarantee of `wait-freedom` where all operations finish in a finite number of steps, is often costly and requires intricate helping mechanisms which can complicate the algorithms.

The design of efficient wait-free and linearizable concurrent queues is a difficult task even if the implementation is allowed to rely on strong synchronization primitives like $Compare\&Swap$. However, many implementations of concurrent FIFO queues have been proposed using shared objects provided by multiprocessor architectures, e.g. $Compare\&Swap$, registers, $Fetch\&Add$, and so on.

Most implementations with sublinear step complexity have limited concurrency, meaning that they limit either the number of enqueuers or dequeuers. For instance, David [16] presents a wait-free linearizable queue with a single enqueuer and multiple dequeuers with constant step complexity. Jayanti and Petrovic [28] provide an implementation of a multiple enqueuer, single dequeuer queue with $O(\log n)$ worst-case step complexity, where n is the number of processes. More recently, Khanchandani and Wattenhofer proposed a multiple enqueuer and multiple dequeuer wait-free queue implementation where both the enqueue and the dequeue operations have a worst-case step complexity of $O(\sqrt{n})$.

Previous solutions leave open the question of whether there exists a wait-free multiple enqueuer and multiple dequeuer queue with logarithmic worst-case step complexity. We investigate the step complexity cost requirements of a FIFO queue implementation with no limitations on the number of processes that can apply *Enqueue* and *Dequeue* operations.

By extension of algorithmic ideas from [28], we first show that a better complexity can be achieved even with multiple enqueuers and multiple dequeuers. In particular, we present a wait-free linearizable concurrent queue for n processes from which all n are enqueuers and $k \leq n$ are dequeuers. In our implementation, the step complexity of an *Enqueue* operation is in $O(\log n)$, while the complexity of a *Dequeue* operation is in $O(k \log n)$. Our implementation has logarithmic complexity as long as k is a constant. Also, it improves on the implementation by Khanchandani and Wattenhofer solution as long as $k \in O(\frac{\sqrt{n}}{\log n})$.

Then, we show that both *Enqueue* and *Dequeue* operations can have worst-case step complexity in $O(\log n)$, if we allow concurrent *Dequeue* operations to return the same element. This relaxed semantic denoted *multiplicity* has been formalized and introduced for the FIFO queue in [14]. Table 3.1 summarizes the state of the art and compares it to the contributions in this work.

	Step complexity	Space complexity	Concurrency limit	CAS - LL/SC	Fetch&Inc - Swap
Khanchandani and Wattenhofer [30]	$O(\sqrt{n})$	$O(nm)$ of $O(max(\log n, \log m))$ registers	None	Y	Y
David [16]	$O(1)$	Unbounded	Single enqueuer	N	Y
Jayanti and Petrovic [28]	$O(\log n)$	$O(n+m)$	Single dequeuer	Y	N
Li [33]	$O(m)$	Unbounded	2 dequeuers	N	Y
Eisenstat [17]	$O(m)$	Unbounded	2 enqueuers	N	Y
Exact queue (this work)	$O(\log n)$ for Enq $O(k \log n)$ for Deq	Unbounded	k dequeuers	Y	Y
Relaxed queue (this work)	$O(\log n)$	Unbounded	None	Y	Y

Table 3.1: Comparing the contributions to state-of-the-art queue implementations (n is the number of processes and m is the number of enqueued elements).

3.2 Wait-Free Linearizable Queue

We present in this section our implementation of a multiple enqueuer multiple dequeuer FIFO queue. Then, we show that the implementation is linearizable and wait-free and that the worst-case step complexity of the *Enqueue* operation and the *Dequeue* operation is $O(\log n)$ and $O(k \log n)$, respectively, where k is the number of dequeuer processes and n the number of all processes.

3.2.1 Inspiration

Jayanti and Petrovic [28] give an implementation of a queue that supports a single dequeuer process and any number of enqueuers. Their implementation has a worst-case step complexity of $O(\log n)$ for both *Enqueue* and *Dequeue* operations, where n is the number of processes.

A preliminary step to their implementation is to present a single enqueuer single dequeuer queue. Implementing this object is simple because of the absence of concurrency between the two processes in any execution: an instance of *Dequeue* operates at the head of the queue while an instance of *Enqueue* acts on its tail. The difficulty arises when considering multiple dequeuer processes. In order to use the single enqueuer single dequeuer queue as a base object for the main algorithm, an additional function was necessary to allow dequeuer processes to read the front of the queue.

The main data structure for the multiple enqueuer single dequeuer queue (Figure 3.1 consists of a binary tree where each leaf is associated with a single enqueuer single dequeuer queue. The number of leaves also represents the number of total enqueuer processes. Henceforth, we denote the single enqueuer single dequeuer queues at the leaves *sub-queues*. The data structure used for all the tree nodes is the CAS object. For each node N of the tree, a *sub-tree* is defined as the substructure such that the root of the sub-tree is N and contains all the children nodes of N up to the leaf layer of the original tree.

Each enqueued element is attributed a unique timestamp. In the leaves of the tree is stored the smallest timestamp of all enqueued elements in the associated sub-queue (single enqueuer single dequeuer queue). Since there is a single enqueuer per sub-queue, it is easy to deduce that the smallest timestamp for a given sub-queue corresponds to

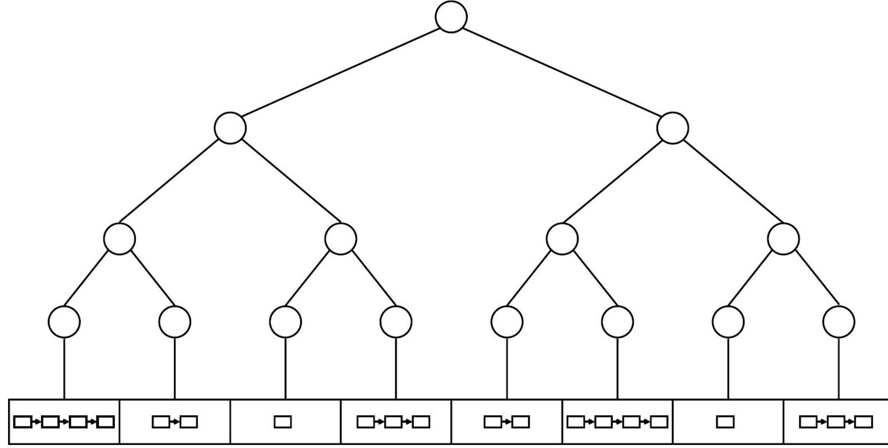


Figure 3.1: Main data structure of the wait-free queue implementation, from [28].

the timestamp of the element at the head of the sub-queue. Then recursively, each internal node of the tree stores the smallest timestamp between its children nodes. The smallest timestamp is propagated to the root of the tree after the execution of each operation. Hence the need for the auxiliary function that allows dequeuer processes to read the head of the sub-queue in order to be able to propagate the smallest timestamp value from a given sub-queue. The goal is to ensure that in the presence of enqueued elements (non-empty queue), the root of the tree stores the smallest timestamp overall. When the dequeuer process executes an instance of *Dequeue*, it reads the timestamp at the root of the tree and returns the corresponding element from the appropriate sub-queue. The *Dequeue* operation will also update the timestamps in the path from the leaf to the root.

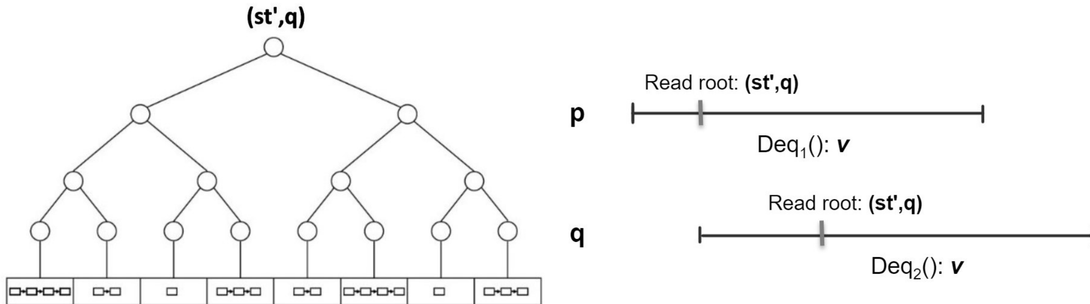


Figure 3.2: Sequential specification violation for the queue implementation in [28] in the case of multiple dequeuer processes.

The limitation of this implementation in multiple dequeuer executions derives from the timestamp-based computations during the *Dequeue* operations. More precisely, while the root of the tree stores the timestamp of a unique enqueued element, if multiple dequeuer processes read the same value at the root and race to return the equivalent value unaware of any concurrent operations, then multiple *Dequeue* operations could return the same element resulting in a violation of the sequential specification of the FIFO queue object. Figure 3.2 represents such a situation. When both processes p

and q read the root of the tree, they retrieve the value of timestamp (st', q). Since they have no knowledge of the other process executing a concurrent operation, both processes dequeue the same element v associated with the unique timestamp (st', q).

3.2.2 Algorithm Overview

We present hereafter a conceptual overview of the algorithm implementing the k-dequeuer n-enqueuer concurrent queue.

The queue object is divided into n different *sub-queue* objects such that each sub-queue i is accessed by the unique enqueuer process with the same id i along with any of the k dequeuer processes. Each sub-queue i is represented by an array of elements $items[i]$ and two pointers $head[i]$ and $tail[i]$ (meaning that *items*, *head*, and *tail* are all two-dimensional arrays). $head[i]$ points to the head of the sub-queue i where the first available element resides, and $tail[i]$ points to the end of the queue. When these two pointers coincide, the sub-queue is empty. Similarly to [28], we link the sub-queue objects together through a binary tree structure where each leaf corresponds to one of the n sub-queues. Whereas in [28], the leaves store single enqueuer single dequeuer queue objects, and the internal nodes are CAS objects. Our tree structure (Figure 3.3) contains CAS objects at every level and the sub-queue objects are implemented aside using the previously described arrays. We can envision that each enqueuer process i is associated with the sub-queue i and the i-th leaf in the binary tree T.

When an $Enqueue(v)$ operation is invoked by an enqueuer process p, the element v is enqueued in the corresponding p-th *sub-queue*. Each enqueued element is associated with a unique timestamp, used by the dequeuers to select the element to be returned (if any).

In particular, each enqueued element is associated to a pair (st, p) where st is the value of a max register, and p is the id of the process that invoked the corresponding *Enqueue* operation. Two processes executing concurrent $Enqueue(v)$ operations can retrieve the same value from the max register, but the process id makes each timestamp unique. Timestamps are totally ordered according to the lexicographical order. The timestamps associated with the elements in a given sub-queue reflect the real-time order of $Enqueue()$ operations by the same process. In particular, if an element e is enqueued in a sub-queue p before another element e', then e is associated with a smaller timestamp than e'. This also means that the head of the sub-queue has the smallest timestamp among the other elements in the same sub-queue.

For the sake of complexity, the timestamps are organized in a tree structure where the n leaves correspond to the timestamps of the elements at the head of the corresponding n sub-queues, and the root stores the smallest timestamp among the ones in the leaves. Our construction is similar to the one proposed by Jayanti and Petrovic in [28].

To manage concurrency in writing the nodes of the tree, we employ the same scheme proposed in [28]: a process writes a node of the tree by calling the CAS primitive. If this first attempt fails, the process tries a second time. Even in the scenario where this second instance of CAS fails, we prove later on, that the value written to the node guarantees the coherence of the values present on the tree structure.

Thus, a *Dequeue* operation simply reads the root of the tree and returns the corresponding element in the appropriate sub-queue in the same manner that this is done in

the *single dequeuer* queue in [28]. The *Dequeue* operation also updates the timestamps stored in the tree in the path from the leaf to the root. However, to support k different dequeuer processes, we need to manage the concurrency between their operations. This is done by introducing a helping mechanism for the *Dequeue* operation. In particular, each *Dequeue* operation has a unique sequence number. Before executing its instance of *Dequeue* operation, a process will first ensure that the instances with smaller sequence numbers are not more pending. If they are, the process will execute the steps necessary for them to finish, and it will update the tree before executing its own instance of *Dequeue*. Since there are k dequeuer processes, during an instance of *Dequeue*, there could be at most $k-1$ other processes executing a *Dequeue* operation concurrently.

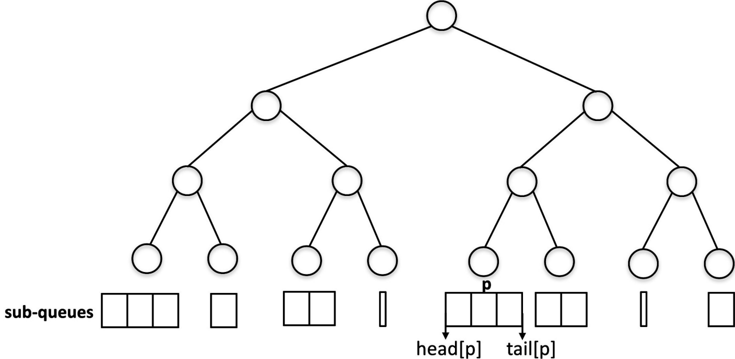


Figure 3.3: Data structure for the k-dequeuer n-enqueuer queue implementation.

3.2.3 Algorithm Pseudocode

In the implementation of the multiple dequeuer and multiple enqueuer queue in Algorithm 6-7, we use two main data structures: a two-dimensional array of registers, called *items*, where each row p together with two integers $head[p]$ and $tail[p]$ represents the sub-queue of process p; and a balanced binary tree T with n leaves where each node is a CAS object used to stores the timestamps of enqueued elements.

The sub-queue p contains the elements enqueued by process p that have not been dequeued, i.e. the current sub-queue p is defined by its values h and t of the max register $head[p]$ and the register $tail[p]$ respectively. If $h = t$, the sub-queue p is empty. Otherwise, it is the ordered list of $t-h$ elements : $items[p][h], \cdots, items[p][t-1]$.

Each *Enqueue* operation executed by process p is associated with a unique timestamp (st, p) where st is an integer obtained from the max register $enqCounter$, and p is the process id. The empty queue is associated with a special timestamp $(\epsilon, -1)$, and we consider that $\epsilon > i \ \forall i \in \mathbb{N}$. $items[p][i] = (val, (st, p))$ means that the i-th *Enqueue* operation by p has enqueued the value val, and that this *Enqueue* has the timestamp (st, p).

The smallest timestamp of a sub-queue p is the timestamp value of $items[p][h]$ where h is the current value of the head of the sub-queue. This timestamp is stored in the p-th leaf of the tree T associated with p, called p-leaf. The following details the different functions of the implementation in Algorithm 6-7.

- $Enqueue(v)$: when process p calls an instance of $Enqueue(v)$, it starts by constructing the corresponding timestamp (st, p) by reading the value of $enqCounter$.

Algorithm 6: Wait-free queue implementation (pseudo-code for process p).

```
1  Shared variables
2    enqCounter : Max register object, initially 0.
3    deqCounter :Fetch&Inc object, initially 1.
4    head[n] : Array of Max register objects, initially 0.
5    tail[n] : Array of registers where each register contains an integer, initially 0.
6    items[n][···] : Two dimensional array of registers, each register contains the
     uplet (val, (st, it)) initially (⊥, (⊥, ⊥)).
7    T : binary tree of CAS objects with n leaves, each node contains the pair
     (st, id), all initially (ϵ, −1).
8    deqOps[···] : Array of CAS objects, initially (⊥, ⊥). deqOps[j] = (i, id)
     means that the j-th Dequeue operation returns items[id][i].val if id ≠ −1,
     otherwise the operation returns ϵ.

9  Function Enqueue(v)
10     st ← enqCounter.MaxRead()
11     t ← tail[p]
12     items[p][t] ← (v, (st, p))
13     tail[p] ← tail[p] + 1
14     enqCounter.MaxWrite(st + 1)
15     Propagate(p)
16     return True

17 Function Dequeue()
18     num ← deqCounter.Fetch&Inc()
19     for (i ← max(1, num − k + 1); i ≤ num; i++) do
20         if deqOps[i].Read() = (⊥, ⊥) then
21             if i > 1 then
22                 UpdateTree(i − 1)
23             FinishDeq(i)
24     (j, id) ← deqOps[num].Read()
25     if id = −1 then
26         return ϵ
27     else
28         (ret, −) ← items[id][j]
29         return ret
```

Process p will then write $(v, (st, p))$ to $item[p][t]$ where t is the value of $tail[p]$. Then, it updates the value of $tail[p]$ to $t+1$. Afterward, the value $st+1$ is written to the max register *enqCounter* to ensure that all subsequent *Enqueue* operations will have a greater timestamp than (st, p). Finally, process p calls $Propagate(p)$ to update the timestamps in the nodes of the tree T from the p-leaf to the root, if necessary.

- $Refresh(node, isLeaf)$: this function is invoked during the execution of an instance

Algorithm 7: Auxiliary functions to the queue implementation

```
1  Function Propagate(id)
2  |  currentNode ← leaf(T, id)
3  |  if !Refresh(currentNode, True) then
4  |  |  Refresh(currentNode, True)
5  |  do
6  |  |  currentNode ← parent(currentNode)
7  |  |  if !Refresh(currentNode, False) then
8  |  |  |  Refresh(currentNode, False)
9  |  while currentNode ≠ root(T)

10 Function Refresh(node, isLeaf)
11 |  (st, id) ← node.Read()
12 |  if isLeaf then
13 |  |  h ← head[id].MaxRead()
14 |  |  t ← tail[id]
15 |  |  if h = t then
16 |  |  |  ret ← node.CAS((st, id), (ϵ, −1))
17 |  |  else
18 |  |  |  (−, (st′, −)) ← items[id][h]
19 |  |  |  ret ← node.CAS((st, id), (st′, id))
20 |  |  return ret
21 |  else
22 |  |  (min_st, min_id) ←read minimum
   |  |    timestamp in current node's children
23 |  |  return node.CAS((st, id), (min_st, min_id))

24 Function FinishDeq(num)
25 |  (−, id) ← root(T).Read()
26 |  if id = −1 then
27 |  |  deqOps[num].CAS((⊥, ⊥), (ϵ, −1))
28 |  else
29 |  |  h ← head[id].MaxRead()
30 |  |  deqOps[num].CAS((⊥, ⊥), (h, id))

31 Function UpdateTree(num)
32 |  (j, id) ← deqOps[num].Read()
33 |  if id ≠ −1 then
34 |  |  head[id].MaxWrite(j + 1)
35 |  |  Propagate(id)
```

of *Propagate* to reset the timestamp stored in a *node*. If the boolean $isLeaf$ is equal to $True$, the current node represents a leaf of the tree T. In this case, the operation computes the minimum timestamp in the corresponding sub-queue. This value is either (1) $(\epsilon, -1)$ if the sub-queue is empty (line 16 of Algorithm 7); or a timestamp (2) (st', i)(line 18 of Algorithm 7). If $isLeaf = False$ then

node is not a leaf; the operation reads the timestamps stored in the children of the current node to compute the minimal timestamp. Then, in both cases, the operation executes the CAS primitive on *node* to write the timestamp and returns the resulting boolean.

- *Propagate*(id): updates the nodes of the tree T in the path from the id-leaf node to the root. Specifically, the function relies on calls to *Refresh* while traversing the path to update each individual *node*. To ensure that the value written into a node is up to date, the call to the function *Refresh*$(node, -)$ is repeated if the first call fails because a concurrent instance r_1 of *Refresh*$(node, -)$ might have written an outdated value since r_1 started before the call to *Refresh*$(node, -)$ in *Propagate*(id). However, after the second call to *Refresh*$(node, -)$, we are certain that the value written is up to date because it can only be written by an instance invoked after *Propagate*(id). This technique is used in the implementation of the single dequeuer multiple enqueuer queue in [28].

- *Dequeue*: First, an instance of the *Dequeue* operation executed by a process p, computes its unique sequence number *num* by applying a $Fetch\&Inc$ primitive on $deqCounter$. Then, p executes the helping mechanism to assist any pending *Dequeue* operation with a sequence number $i \in [max(1, num - k + 1), num])$ in increasing order of i. If the operation with the index i is still pending (i.e. $deqOps[i]$ is still set to its initial value), p executes $UpdateTree(i - 1)$ if $i > 1$, to ensure that the root of the tree is updated to an accurate value. Then, p executes $FinishDeq(i)$ to decide on the operation's return value in $deqOps[i]$. After the return values have been decided for all *Dequeue* operations with indexes in $[max(1, num - k + 1), num])$, p reads $deqOps[num] = (i, j)$ and returns $items[j][i].val$, otherwise p returns ϵ.

- *FinishDeq*(num): The array *DeqOps* stores the information regarding the return values of each *Dequeue* operation. A call to *FinishDeq* with the parameter *num* decides a value and attempts to write it to $DeqOps[num]$ using a CAS primitive.FinishDeq(num) reads the timestamp at the root of the tree $T : (-, id)$. And if $id = -1$ (i.e. the queue is empty), then $(\epsilon, -1)$ is written to $DeqOps[num]$. Otherwise, the value (h, id) is written to $DeqOps[num]$ where h is the value of the head of the sub-queue id. In either scenario, if the CAS instruction fails, another process has succeeded in executing a CAS instruction on $DeqOps[num]$ and the return value for the corresponding *Dequeue* has been decided.

- *UpdateTree*(num): A simple function call that encapsulates the steps necessary before executing the *Dequeue* operation with the sequence number $num + 1$. If the *Dequeue* operation with the sequence number *num* returns ϵ, then there are no additional steps necessary. Otherwise, if an element has been returned, it is necessary to update the head of the sub-queue id from which the return value was retrieved; followed by a call to the function *Propagate*(id) to update the tree accordingly.

3.2.4 Proof

In this section, we establish that Algorithm 6-7 is a wait-free implementation of a k-dequeuer multi-enqueuer queue. We also establish that an *Enqueue* operation has a worst-case step complexity of $O(\log n)$ and a *Dequeue* operation has a worst-case step complexity of $O(k \log n)$.

Algorithm properties

Each *Dequeue* operation is associated with a unique sequence number that is the value obtained by applying the $Fetch\&Inc$ primitive on *deqCounter* at line 18 of Algorithm 6.

Lemma 3.2.1. *A total order between Dequeue operations is provided by their sequence number. This order respects the real-time order.*

Proof. Let deq_1 and deq_2 be two *Dequeue* operations by process p_1 and p_2 respectively. Let seq_1 be the sequence number of deq_1 and seq_2 be the sequence number of deq_2. We prove that if deq_1 precedes deq_2 in real-time order, then $seq_1 < seq_2$.

deq_1 completes before deq_2 is invoked, thus p_1 executes line 18 of Algorithm 6 before the invocation of deq_2 by p_2. The proof follows from the fact that *deqCounter* is a linearizable $Fetch\&Inc$ object. □

The *Dequeue* operation with the sequence number i is complete at a given configuration C if $DeqOps[i] \neq (\bot, \bot)$ (i.e.; the value of $DeqOps[i]$ at C is not the initial value). Otherwise, it is incomplete at C.

Observation 3.2.2. *Let deq denote a Dequeue operation with the sequence number i. Any call to $FinishDeq(i)$ is executed after the invocation of deq.*

Lemma 3.2.3. *Fix an execution E and let C be any configuration of E. $\forall h > 0$ and $\forall i \geq 1$, if the $h + i$-th Dequeue operation exists and it is complete at C, then the i-th Dequeue operation is complete at C.*

Proof. Consider the first configuration C where the $h+i$-th *Dequeue* operation is complete, i.e.; $deqOps[i + h] \neq (\bot, \bot)$. Assume by contradiction that $deqOps[i]$ has its initial value at C.

The value of $deqOps[i]$ is only set during the execution of $FinishDeq(i)$ at line 30 or 27 of Algorithm 7. According to the condition in the for-loop (line 19 of Algorithm 6), only a *Dequeue* operation with a sequence number $i + h \leq l \leq i + h + k - 1$ may change the value of $deqOps[i + h]$.

According to Lemma 3.2.1, the *Dequeue* operations with a sequence number smaller than or equal to l, and in particular $\in [i, l]$, have started at the configuration immediately before the value of $deqOps[i + h]$ is changed by the l-th *Dequeue* operation. Also, the *Dequeue* operations with a sequence number $num \in [i, i + k - 1]$ could not have returned at C otherwise $deqOps[i] \neq (\bot, \bot)$ at C (contradicting our assumption). This is trivially true for $num = i$. For $num \in [i + 1, i + k - 1]$, and since the condition at line 20 of Algorithm 6 is *true* for $deqOps[i]$, the *Dequeue* operation with sequence number *num* will execute the $FinishDeq(i)$ function and set $deqOps[i] \neq (\bot, \bot)$ before it returns.

Thus, l should be greater than $i+k-1$. But this means that there are $k+1$ pending *Dequeue* operations, which contradicts the fact that we can have at most k pending *Dequeue* operations. There is a contradiction. □

As $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$; the following observation is a consequence of Lemma 3.2.3.

Observation 3.2.4. *Before the first execution of $FinishDeq(i+h)$, $FinishDeq(i)$ has been executed.*

Each *Enqueue* operation *op* has a unique timestamp composed of an integer obtained by reading the Max register $enqCounter$ during the execution of line 10, and the *id* of the process that executed the operation *op*.

Observation 3.2.5. *For each p, the timestamps of the elements written in the sub-array $items[p]$ are monotonically increasing in accordance with their index in the array. In other terms, we have $items[p][i].ts < items[p][i+1].ts$.*

At any given configuration, the sub-queue of process p is the sub-array of $items[p]$ in the range $items[p][head[p].MaxRead()], ..., items[p][tail[p]-1]$.

Lemma 3.2.6. *Let enq_1 and enq_2 be two Enqueue operations such that enq_1 ends before enq_2 is invoked. Let (st_1, id_1) be the timestamp of enq_1 and (st_2, id_2) be the time stamp of enq_2. We have $st_1 < st_2$.*

Proof. After the execution of line 14 of Algorithm 6 during enq_1, any value returned by a `enqCounter`.$MaxRead$ is greater or equal to $st_1 + 1$. The claim follows from the fact that enq_2 executes line 10 of Algorithm 6 after enq_1 returned. □

We say that the i-th *Enqueue* operation by a process p matches the *Dequeue* operation with sequence number j, if $deqOps[j] = (i, p)$ at some point in the execution.

Meaning, if the *Dequeue* operation returns, it returns the element enqueued by the i-th *Enqueue* operation of process p (i.e. `items[p][i]`).

Lemma 3.2.7. *An Enqueue operation has at most a single matching Dequeue operation.*

Proof. Let *enq* be the i-th *Enqueue* operation by a process p. Assume by contradiction that there are two *Dequeue* operations, deq_1 and deq_2 that match *enq*. Let j_1 and j_2 be their corresponding sequence numbers. Then, $deqOps[j_1] = deqOps[j_2] = (i, p)$. By Lemma 3.2.1 and without loss of generality, let $j_1 < j_2$. Because of the Observation 3.2.4, $FinishDeq(j_1)$ returned before $FinishDeq(j_2)$ is invoked. According to lines 22 to 23 of Algorithm 6, $UpdateTree(j_1)$ is executed before $FinishDeq(j_1+1)$. This means that the value $i+1$ is written in the Max register $head[p]$ at line 34 before that a process read it during the $FinishDeq(j_1+1)$. And since $j_2 \geq j_1+1$, the claim follows. □

Lemma 3.2.8. *Let enq denote the i-th Enqueue operation by a process p. Let $ts = (st, p)$ be the timestamp of enq. Let s be any node in the tree T in the path from the p-th leaf to the root of the tree. At any configuration C after enq ends and such that $deqOps[j] \neq (i, p)$ for each $j \geq 0$, we have that the timestamp stored at s is smaller than or equal to ts at C.*

Proof. After enq, we have that $tail[p] \geq i+1$, because enq is the i-th $Enqueue$ operation executed by p.

We first prove that after enq, $head[p]$ is smaller than or equal to i as long as $deqOps[l] \neq (i,p)$ for any $l \geq 0$.

The value of $head[p]$ is updated only during the execution of the function $UpdateTree$ (line 34 of Algorithm 7). In particular, the value of $head[p]$ is set to a value $j+1$ where j is the value read from some $deqOps[num]$ at line 32. Also, the value of $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$ with a value read from $head[p]$ (lines 29 and 30). We prove by induction on j that if the value written in $head[p]$ is j then, all values $0,\dots j-1$ have been previously written in $head[p]$ (in increasing order) and to some $deqOps[num]$. The base case is for $j=1$. Consider the first MaxWrite() that writes 1 to $head[p]$ and let q be the process applying this primitive. According to line 34, q has read the value $(0,p)$ from some $deqOps[num]$, which has been updated with a value read from $head[p]$. The claim follows.

Suppose this is true for a value j, we show that the claim holds for $j+1$. Consider the first process, denoted q, that writes $j+1$ into $head[p]$. q has read (j,p) from some $deqOps[num]$ at line 32. By inductive hypothesis, and by the linearizability of $head[p]$ all the values $0,\dots j$ have been written in $head[p]$ and all the values $0,\dots j-1$ have been written in some $deqOps[num]$. The claim follows.

Hence, $head[p] \leq i$ as long as for any $l \geq 0$, we have $deqOps[l] \neq (i,p)$. This is because to write the value $i+1$ (and then any greater value), a process has to read $deqOps[l]=(i,p)$ for some l.

base case $k=0$. s is the p-th leaf. Since enq completes, there is at least one instance of $Propagate(p)$ performed after that process p has written the value i in $tail[p]$. The value of $head[p]$ is smaller than or equal to i, so any instance of $Propagate(p)$ that changes the value of s before C, will write a timestamp read in $items[p][j]$ for some $j \geq i$. By Observation 3.2.5, the timestamp read is smaller than or equal to $ts=(st,p)$.

It remains to prove that after an instance of $Propagate(p)$ completes, denoted $prop$, a value smaller than or equal to i has been written in the leaf corresponding to p. An instance of $Propagate(p)$ performs two $Refresh(s)$. Each $Refresh(s)$ reads the state of s, then the $head[p]$ and the corresponding timestamp ts and then applies a CAS to s to modify its value with ts. Suppose that both $Refresh(s)$ fail (and in particular the second one), otherwise the claim is trivial. The second $Refresh(s)$ fails because another an instance of $Propagate(p)$, denoted $prop'$ successfully applied a CAS on s. But $prop'$ has read $head[p]$ after $tail[p]$ is set to i. Meaning that it has read a value smaller than or equal to i and it writes in s the corresponding timestamp that is smaller than or equal to ts.

induction case $k+1 \leq \log n$. Suppose that the claim holds for $j \leq \log n$: the timestamp stored at s_j is smaller than or equal to ts where s_j is in the path from the p-th leaf to the root at a height of $j \leq k$. We prove that the claim holds for the parent of s_j, denoted s_{j+1}.

Any instance of $Propagate(p)$ updates the nodes in the path from the p-th leaf to the root, one by one, starting from the leaf and following the path to the root. Also, immediately after enq completes, there is at least one $Propagate(p)$ instance that passed through all the nodes in this path. Consider, the first $Propagate(p)$ that updated node s_{j+1} after s_j has been updated, denoted $prop$.

Observe that any process that executes the $Refresh$ function on node s_{j+1} writes

the minimum timestamp it reads from the children of s_{j+1}. And that the second $Refresh(s_{j+1})$ fails only if another $Propagate(p)$ has modified the state of this node with a value smaller than or equal to the value at s_j read by *prop*. □

Lemma 3.2.9. *Let enq be an Enqueue operation with the timestamp ts that enqueued* $items[p][i]$. *If* (i, p) *was written to deqOps*$[j]$ *by a process q, then the execution of line 25 of Algorithm 7 to read ts by q was executed after the invocation of enq.*

Proof. *enq* is the i-th enqueue operation by p. Let *deq* be the *Dequeue* operation executed by q that retrieves ts from the root of the tree (Line 25 of Algorithm 7) before writing (i, p) to *deqOps*$[j]$. *enq* must execute the line 13 of Algorithm 6 before ts can be propagated in the tree according to the code of function $Refresh$. The claim follows. □

Lemma 3.2.10. *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked. If* enq_2 *has a matching Dequeue operation* deq_2*, then* enq_1 *also has a matching Dequeue operation* deq_1*.*

Proof. By contradiction, we suppose that deq_2 exists and deq_1 does not. We denote ts_1 and ts_2 the timestamps associated with enq_1 and enq_2 respectively and num_2 the sequence number of deq_2. From Lemma 3.2.6, $ts_1 < ts_2$ because enq_1 ends before enq_2 begins.

And since enq_1 does not have a matching *Dequeue*, there is no $j \geq 0$ such that $deqOps[j] = (i, p)$ where $items[i][p]$ is enqueued by enq_1. Therefore, from Lemma 3.2.8, for any node s in the path in T from the p-th leaf to the root, the timestamp stored at s is smaller than or equal to ts_1. In particular, for the root of the tree, the timestamp stored is smaller or equal to ts_1. From Lemma 3.2.9, the step of line 25 of Algorithm 7 to read the root of the tree before writing $deqOps[num_2]$ is executed after the invocation of enq_2 which is after the invocation of enq_1. Meaning that during this step, the timestamp at the root was smaller or equal to ts_1 contradicting the fact that $ts_1 < ts_2$. □

Lemma 3.2.11. *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked and let* deq_1 *and* deq_2 *be the matching Dequeue operations to* enq_1 *and* enq_2 *respectively. We have that* deq_1 *has a lower sequence number than* deq_2*.*

Proof. We denote num_1 and num_2 the sequence numbers of deq_1 and deq_2 respectively, and ts_1 and ts_2 the timestamps of enq_1 and enq_2 respectively. By contradiction, we suppose that $num_1 > num_2$. Since enq_1 ends before enq_2 begins we have that $ts_1 < ts_2$ (Lemma 3.2.6).

And since $deqOps[i]$ are written in an increasing order of i according to Lemma 3.2.3, we have that $deqOps[num_2]$ is written before $deqOps[num_1]$. However, from Lemma 3.2.8, as long as $deqOps[num_1]$ has its initial value, then the timestamp stored at the root is smaller than or equal to ts_1. At the execution of line 25 of Algorithm 7 to compute the final value of $deqOps[num_2]$, the root has a timestamp smaller or equal to ts_1; contradicting the fact that $ts_1 < ts_2$. □

Lemma 3.2.12. *Let deq be a Dequeue operation and let enq be an Enqueue operation that ends before deq is complete. Let* C *be a configuration of* E *where enq does not have a matching Dequeue operation* deq' *or* deq' *is not complete at* C*. If deq is complete at* C*, then deq does not return* ϵ*.*

Proof. By contradiction, we suppose that deq returns ϵ. Let i denote the sequence number of deq and ts denote the timestamp of enq. Since deq returns ϵ, deq reads the value $(\epsilon, -1)$ in $deqOps[i]$ at line 24 of Algorithm 6. Therefore, during the execution of $FinishDeq(i)$, the process that writes $deqOps[i]$, reads $(\epsilon, -1)$ at the root of the tree (line 27 of Algorithm 7). However, By Lemma 3.2.8, the timestamp at the root of the tree after the end of enq is smaller than or equal to ts. Meaning that during the execution of line 25 of Algorithm 7 during the instance $FinishDeq(i)$ that writes $deqOps[i]$, the timestamp at the root of the tree was smaller than or equal to ts. We reach a contradiction because $(\epsilon, -1)$ is larger than any timestamp $(h, -)$ $\forall h \in \mathbb{N}$. □

Linearizability

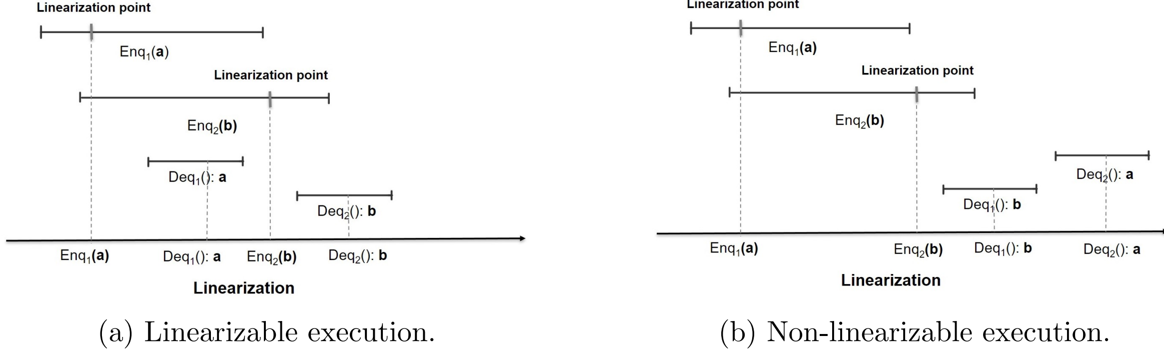
In the literature, a popular approach to defining the linearization of an execution of a shared object implementation consists of defining a *linearization point* for each operation in the execution. Simply speaking, a step executed during a high-level operation is chosen as the instant where the operation takes effect. Since each of these linearization points falls within the execution interval of its corresponding operation, it is possible to define a total order of the operations based on the linearization points. The linearization is the sequential execution of the operations following the total order defined. The linearization is correct if it is shown to follow the real-time execution order and all the operations behave according to the sequential specification of the object.

Using this technique to prove the linearizability of an implementation has the advantage of simplifying the proof of correctness in regard to the real-time execution order. If the execution intervals of two operations in the execution do not interweave, then it is simple to prove that the first of the two operations will be linearized first since the linearization point is defined in a segment of the execution prior to the invocation of the second operation.

However, it is not always possible to employ this method to define the linearization. In some cases, it is impossible to define the linearization point of an operation independently from the entire execution. Meaning that future operations in the execution might affect the correct order in which an operation needs to be inserted into the linearization to ensure its behavior is in accordance with the sequential specification of the object.

Consider, for example, the implementation of a FIFO queue. And assume that the *Enqueue* and *Dequeue* operations are linearized through the definition of a linearization point within their execution interval. Figure 3.4 represents different execution scenarios of such an implementation. While the first execution in Figure 3.4a is linearizable through the total order defined by the linearization points. In Figure 3.4b, this order violates the sequential specification of the FIFO queue since the elements a and b are returned out of order.

More specifically, The order of linearization of the two concurrent *Enqueue* operations depends on the order in which the elements enqueued were returned (i.e. the order of the *Dequeue* operations). If the linearization of the *Enqueue* operations is determined without taking into consideration the order of the returned elements, the linearization might not follow the FIFO order since the first *Dequeue* operation might not return the first available element in the linearization. Therefore, to prove the linearizability of our implementation, we follow a different method where the insertion of each operation



(a) Linearizable execution.

(b) Non-linearizable execution.

Figure 3.4: Linearization of different execution scenarios by considering linearization points.

from the execution is explicitly described in relation to the preexisting operations in the linearization.

First, we construct a permutation L of some of the *Dequeue* and *Enqueue* operations invoked such that L contains all operations that have terminated. Then, we prove that L preserves the real order as well as the semantics of a queue.

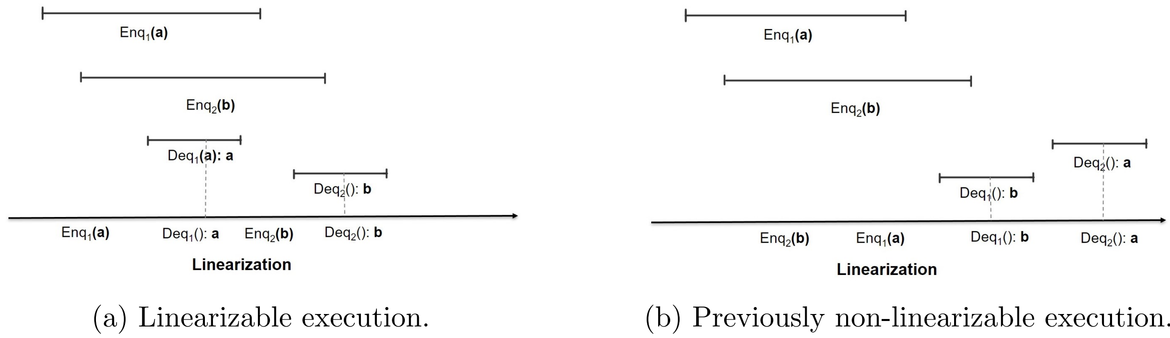
Linearization definition Let E denote a given execution of the wait-free queue implemented in Algorithm 6 and Algorithm 7. We classify every *Dequeue* operation *deq* that appears in E to exactly one of the following types :

1. *deq* does not execute line 18 of Algorithm 6 in E. Thus *deq* is not attributed a sequence number.
2. *deq* executes line 18 of Algorithm 6 in E, its sequence number is j and `deqOps[j]` has the initial value $(\perp, \perp)$ in E.
3. *deq* executes line 18 of Algorithm 6 in E, its sequence number is j and `deqOps[j]` $\neq (\perp, \perp)$ in E.

We remove from E, any *Dequeue* operation of type 1 and 2. We denote DEQ the set of *Dequeue* operations of type 3. Each operation in DEQ is associated with a unique sequence number $j \in \mathbb{N}_0$. We totally order all the operations in DEQ according to their sequence number. Also, let *deq* be any incomplete *Dequeue* operation in DEQ and let j be its sequence number. We complete *deq* by returning the value v if `deqOps[j]` $= (i, id)$ in E and `items[id][i]` $= (v, -)$. Otherwise, we complete *deq* by returning the empty queue value ϵ.

We remove every *Enqueue* operation that does not execute line 13 of Algorithm 6 in E. We denote ENQ the set of *Enqueue* operations that appear in E and that we do not remove. Every *Enqueue* operation *enq* in ENQ is uniquely identified by a pair (i, id) meaning that *enq* is the i-th *Enqueue* operation performed by the process id. We associate the *Dequeue* operation in DEQ with sequence number i with the *Enqueue* operation (j, id) such that `deqOps[i]` $= (j, id)$.

Let ENQ_d denote the *Enqueue* operations in ENQ that have an associated *Dequeue* operation in DEQ. We associate each *Enqueue* operations in ENQ_d with the sequence



(a) Linearizable execution. (b) Previously non-linearizable execution.

Figure 3.5: Linearization of different execution scenarios following proposed rules.

number of the corresponding *Dequeue*. Thus, *Enqueue* operations in ENQ_d are totally ordered according to the sequence number of their matching *Dequeue* operations.

We construct the linearization L of the operations in E as follow:

1. First we insert the *Enqueue* operations in ENQ_d one by one and according to their total order, denoted $enq_{i_1}, enq_{i_2} \ldots$ in L. Notice that enq_{i_h} is the *Enqueue* operation associated with the *Dequeue* operation having the sequence number i_h. Assuming that $enq_{i_{h+1}}$ exists, we have $i_h < i_{h+1}$; and all the *Dequeue* operations having a sequence number $i \in [i_h + 1, i_{h+1} - 1]$ return the value ϵ.

2. Then, we insert the *Dequeue* operations one by one according to their the sequence number. For any sequence number k, If deq_k returns ϵ it is inserted immediately after deq_{k-1} if it exists, or at the beginning otherwise. In the case where deq_k does not return ϵ, it is linearized immediately after the furthest point in L following: (i) the previous deq_{k-1}, (ii) the matching *Enqueue* operation enq_{i_l} with $i_l = k$, and (iii) the last *Enqueue* operation that ends before the invocation of deq_k.

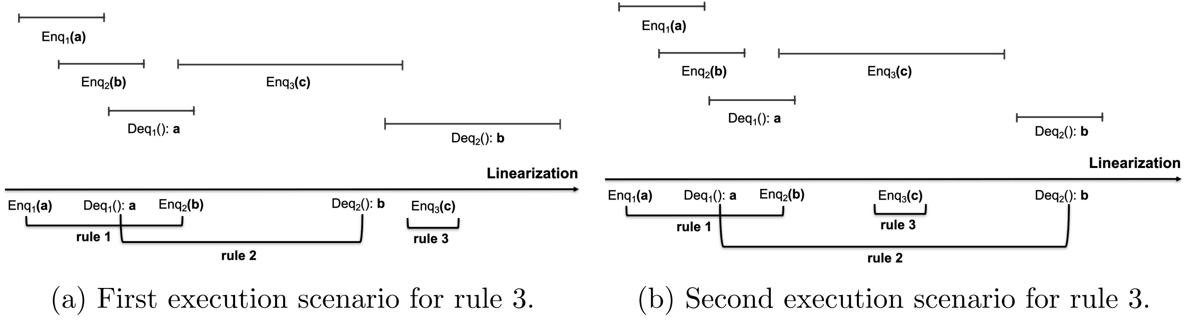
3. Let enq denote an *Enqueue* operation from the remaining *Enqueue* operations with no matching *Dequeue* operations (i.e. $ENQ \setminus ENQ_d$). We insert enq after the last operation in ENQ_d and before the first *Dequeue* operation that starts after enq ends (or at the end of L if such *Dequeue* does not exist). If multiple operations from $ENQ \setminus ENQ_d$ are linearized at the same point, then they are ordered according to their real-time order.

The execution shown in Figure 3.4b is now linearizable following the rules proposed since the *Enqueue* operations follow the order of the matching *Dequeue* operations (Figure 3.5b). And in Figure 3.6, we show how different executions are linearized by following the rules in order and we highlight in particular the two possible scenarios for rule 3.

For two operations op_1 and op_2, we denote $op_1 <_L op_2$ when op_1 precedes op_2 in the linearization L.

Linearization and real-time order We show that the linearization defined in the previous section respects the real-time execution order.

Lemma 3.2.13. *Let op_1 and op_2 be two Enqueue operations in E such that op_1 ends before op_2 is invoked. op_1 precedes op_2 in L.*



(a) First execution scenario for rule 3. (b) Second execution scenario for rule 3.

Figure 3.6: Linearization rules applied to two executions.

Proof. First, consider the case where both operations do not have matching *Dequeue* operations. From linearization rule 3, an *Enqueue* operation that does not have a matching *Dequeue* operation is linearized before the first *Dequeue* operation that starts after it ends or at the end of L if such *Dequeue* operation does not exist. If op_1 is linearized at the end of L, then op_2 is also linearized at the end of L after op_1, because op_2 starts after op_1 ends and there is no *Dequeue* operation that starts after op_1 ends. We suppose that there exists a *Dequeue* operation deq_1 such that op_1 is linearized immediately before deq_1. If op_2 is linearized at the end of L, the claim is trivial. So let deq_2 be a *Dequeue* operation such that op_2 is linearized immediately before deq_2. We have $op_1 <_{ro} op_2 <_{ro} deq_2$. Meaning that $deq_2 = deq_1$ or $deq_1 <_L deq_2$, because both operations start after op_1 ends, and deq_1 is the first such operation in L. Therefore, $op_1 <_L op_2$ according to their real time execution order following linearization rule 3.

Next, if op_1 has a matching *Dequeue* operation but op_2 does not, we have that op_2 is linearized after the last linearized *Enqueue* operation that has a matching *Dequeue* operation. The case where op_1 does not have a matching *Dequeue* operation but op_2 does, is impossible according to Lemma 3.2.10. We suppose that both op_1 and op_2 have matching *Dequeue* operations, named respectively deq_1 and deq_2. From Lemma 3.2.11, we have that deq_1 has a smaller sequence number than deq_2. Therefore, from linearization rule 1, op_1 is linearized before op_2. □

Lemma 3.2.14. *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq', then the sequence number of deq' is greater than j.*

Proof. We denote i the sequence number of deq'. By contradiction we suppose that $j > i$. We consider the configuration C where deq completes. According to Lemma 3.2.3, deq' also has been completed at C. Meaning that $deqOps[i] \neq (\perp, \perp)$ at C. However, from the hypothesis, enq has not started at C, as enq is not invoked until deq finishes. According to Lemma 3.2.9, deq' cannot match enq. The claim follows. □

Lemma 3.2.15. *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq', then any Dequeue operation with a sequence number $l < j$ is linearized before enq.*

Proof. By contradiction, we suppose that there exists *Dequeue* operations with sequence numbers strictly smaller than j that are linearized after enq, and let deq_l be the first of these operations in L. Thus , if deq_{l-1} exists, we have that $deq_{l-1} <_L enq$.

If deq_l returns ϵ, from linearization rule 2, deq_l is linearized immediately after deq_{l-1} if it exits, or at the beginning of L. Therefore, $deq_l <_L enq$. There is a contradiction.

Otherwise, deq_l has a matching *Enqueue* operation denoted enq_l. We denote i the sequence number of deq'. From Lemma 3.2.14, we have that $j < i$. Therefore, $l < j < i$. Thus, $enq_l <_L enq$ from linearization rule 1. Furthermore, we have $deq_{l-1} <_L enq$ (if it exists). Therefore, since $enq_l <_L enq$ and $deq_{l-1} <_L enq$, according to linearization rule 2, $enq <_L deq_l$ because $enq <_{ro} deq_l$ (rule 2.3 of linearization) . Consequently, $deq_j <_{ro} enq <_{ro} deq_l$. Contradicting the fact that $l < j$ (Lemma 3.2.1). □

Theorem 3.2.16. *Let op_1 and op_2 be two operations in E such that op_1 ends before op_2 is invoked. Then, op_1 precedes op_2 in L.*

Proof. Four cases have to be studied according to the type of operations.

1. op_1 and op_2 are two *Dequeue* operations. Since op_1 ends before op_2 begins, the sequence number i_1 of op_1 is strictly smaller than the sequence number i_2 of op_2 (Lemma 3.2.1). From linearization rule 2, we have op_1 is before op_2 in L.

2. The case where op_1 and op_2 are *Enqueue* operations is proved by Lemma 3.2.13.

3. op_1 is an *Enqueue* operation and op_2 is a *Dequeue* operation. First, consider the case that op_2 does not return ϵ. If $op_1 \in ENQ_d$, then from linearization rule 2, op_2 is linearized after op_1 because op_2 is inserted after the last *Enqueue* operation that ends before op_2 starts. Otherwise, If $op_1 \notin ENQ_d$, from linearization rule 3, it is linearized before the first *Dequeue* operation that starts after op_1 ends. Thus op_1 is linearized before op_2.

   Next, consider the case where op_2 returns ϵ, and let i denote its sequence number. By Observation 3.2.2 and Lemma 3.2.12, op_1 has a matching *Dequeue* operation deq, and deq is complete before op_2 is complete.

   Let j is the sequence number of deq. Since deq is complete before op_2 is complete, by Lemma 3.2.3, we have that $j < i$. Therefore, from linearization rule 2, deq is linearized before op_2. Thus, from linearization rule 1, $op_1 <_L deq <_L op_2$. The claim follows.

4. Finally, we suppose that op_1 is a *Dequeue* operation and that op_2 is an *Enqueue* operation. If op_2 does not have a matching *Dequeue* operation, from linearization rule 3, it is linearized before the first *Dequeue* operation that starts after op_2 ends or at the end of L if such operation does not exist. Thus, op_2 is linearized after op_1 because op_1 ends before op_2 starts.

   So consider that op_2 has a matching *Dequeue* operation deq and let i be its sequence number and j be the sequence number of op_1.
   If op_1 returns ϵ, from the linearization rule 2, we have $op_1 = deq_j$ is linearized immediately after deq_{j-1} (or beginning of L if it does not exist). And from Lemma 3.2.15, for each $l < j$, we have that deq_l is linearized before op_2. In particular, we have that deq_{j-1} is linearized before op_2. Therefore, op_1 is linearized before op_2.

   Otherwise, consider enq_j the matching operation of op_1. From linearization rule 2, op_1 is linearized after (i) deq_{j-1}, (ii) enq_j and after (iii) the last *Enqueue* enq' that ends before op_1 starts. We show that op_2 is linearized after all these three

operations. From Lemma 3.2.15, we have that deq_{j-1} is linearized before op_2 (i). From Lemma 3.2.14, we have that $j < i$ meaning that enq_j is linearized before op_2 according to the total order of the sequence numbers of their matching *Dequeue* operations (ii). And since op_1 ends before op_2 starts, $enq' <_{ro} op_2$. Therefore, $enq' <_L op_2$ because we have shown that the linearization of the *Enqueue* operations respects the real time execution order (Lemma 3.2.13) (iii). The claim follows.

□

Linearization and the Queue Sequential Specification

Lemma 3.2.17. *Let deq be a Dequeue operation that returns $v \neq \epsilon$. There exists an Enqueue(v) denoted enq that such that enq is linearized before deq and there is no Dequeue operation $deq' \neq deq$ that also returns v.*

Proof. First, we prove that *enq* exists. Since *deq* returns $v \neq \epsilon$, it has read a value (j, p) in $deqOps[i]$ where i is the sequence number of *deq* (line 24 of Algorithm 6). Meaning that $items[p][j] = v$ and the *Enqueue* operation that enqueued v denoted *enq*, is the j-th instance of *Enqueue* by process p. By linearization rule 2, *deq* is linearized after *enq*. And we have shown in Lemma 3.2.7 that each *Enqueue* operation has at most a single matching *Dequeue* operation. The claim follows. □

Lemma 3.2.18. *Let enq_1 and enq_2 be two Enqueue operations such that $enq_1 <_L enq_2$. If enq_2 has a matching Dequeue deq_2, then enq_1 has a matching Dequeue deq_1 and $deq_1 <_L deq_2$.*

Proof. By contradiction, we suppose that enq_1 does not have a matching *Dequeue* operation. From linearization rule 3, enq_1 is linearized after all *Enqueue* operations in ENQ_d. Especially, enq_1 is linearized after enq_2. There is a contradiction. And from linearization rule 1, enq_1 and enq_2 are linearized according to the total order of the sequence numbers of their matching *Dequeue* operations. The claim follows. □

From the two previous Lemmas 3.2.17-3.2.18, we have the following theorem.

Theorem 3.2.19. *Let deq be a Dequeue operation in L. If deq does not return ϵ, then it returns the element enqueued by the first Enqueue operation in L that does not have a matching Dequeue operation linearized before deq.*

Lemma 3.2.20. *Let deq_ϵ be a Dequeue operation that returns ϵ. And let enq be an Enqueue operation linearized before deq_ϵ. We have that enq has a matching Dequeue operation deq that is also linearized before deq_ϵ.*

Proof. First, we show that *enq* has a matching *Dequeue* operation *deq*. By contradiction, we suppose that *enq* is in $ENQ \setminus ENQ_d$. From linearization rule 3, *enq* is inserted before the first *Dequeue* operation deq' that starts after *enq* ends or at the end of L if deq' does not exist. The case where *enq* is linearized at the end of L is trivial because it contradicts the fact that *enq* is linearized before deq_ϵ. So deq' exists. By lemma 3.2.12 deq' does not return ϵ. Since $enq <_L deq_\epsilon$, we have $deq' <_L deq_\epsilon$ Hence, deq_ϵ has a greater sequence number than deq' from linearization rule 2. Thus, deq_ϵ is complete

after deq' is complete (Lemma 3.2.3). We conclude by lemma 3.2.12, that deq_ϵ does not return ϵ. There is a contradiction. Thus, *enq* has a matching *Dequeue* operation denoted *deq*.

In the following, we establish that *deq* is linearized before deq_ϵ. Let i denote the sequence number of deq_ϵ and let j be the sequence number of *deq*. By contradiction, we assume that $i < j$ (i.e. *deq* is linearized after deq_ϵ). Let deq_k be the first *Dequeue* operation linearized after *enq* with k its sequence number. Such an operation exists as $enq <_L deq_\epsilon$. We have $k \leq i$, according to the linearization rule 2. Assume that deq_k returns ϵ. If $k = 0$ then no operation is linearized before deq_k; in this case, there is a contradiction. Otherwise ($k \geq 1$), there is no *Enqueue* operation linearized after deq_{k-1} and before deq_k because deq_k is linearized immediately after deq_{k-1} (linearization rule 2). This contradicts the fact that deq_k is the first *Dequeue* operation linearized after *enq*. Hence deq_k does not return ϵ. We conclude that $k < i$. Therefore, deq_k is complete before deq_ϵ is complete (Lemma 3.2.3). deq_k does not match *enq* as we assume that *deq* is linearized after deq_ϵ. From linearization rule 2, deq_k can only be linearized after *enq* because *enq* terminates before the invocation of deq_k. Thus, by Lemma 3.2.12, deq_ϵ cannot return ϵ if $j > i$. There is a contradiction. □

Step Complexity

We show that the worst-case step complexity of an *Enqueue* and *Dequeue* operation is $O(\log n)$ and $O(k \log n)$, respectively. To do so, we establish the following Lemma. The main intuition is that while propagating the timestamp the process has to read a constant number of nodes per level going from a leaf to a root. Since there are n leaves, the high of the tree is in $O(log(n)$.

Lemma 3.2.21. *A process executes $O(\log n)$ steps during a call to the function Propagate(id).*

Proof. When a process calls the function *Propagate*(id), it will update the binary tree starting from the leaf that corresponds to the sub-array *items*$[id]$. Meaning, that the process first retrieves the values h and t of *head*$[id]$ and *tail*$[id]$ respectively, and then either realizes that there are no available elements in the sub-array anymore (line 16 of Algorithm 7), or retrieves the time stamp of the element indexed in h (line 18 of Algorithm 7). Since there are no loops during these computations, the process will execute them in constant time. Afterward, the process will traverse down-up the binary tree of height $\log n$ to propagate the information toward the root. During each step, the process reads the minimum timestamp of the node's children, and attempts to write that minimum to the current node using a CAS primitive. If the first attempt fails, the process will try a second time. Therefore, the entire journey from the leaves to the root of the tree is done in $O(\log n)$ steps. The claim follows. □

During the execution of an *Enqueue* operation there are no loops or function calls aside from a call to the function *Propagate*(id). And during a *Dequeue* operation, a process executes at most k instances of *Propagate*(id). The following corollary ensues.

Corollary 3.2.21.1. *A process executes $O(\log n)$ steps during the execution of an Enqueue operation and $O(k \log n)$ steps during the execution of a Dequeue operation.*

3.3 Set-Linearizable Wait-free Queue Algorithm with Multiplicity

In this section, we rely on the approach of relaxing the semantics of the FIFO queue to propose a wait-free implementation where both the *Enqueue* and *Dequeue* operations have a worst-case step complexity of $O(\log n)$.

Specifically, we consider the *set-sequential* specification of shared objects formally introduced in Section 1.2.3 and the weakened consistency condition of *set-linearizability* [35]. Simply put, the set-sequential specification of an object allows for multiple operations to be executed simultaneously even in a sequential setting. And we say that an execution E of a concurrent object is set-linearizable if there exists an equivalent set-sequential execution S that contains all the complete operations of E and possibly some pending operations such that if an operation op is before another operation op' in E then op is also before op' in S. Figure 3.7 illustrates the difference between the linearization and set-linearization of the same execution.

While the approach to relaxation usually relies on either considering a weakened consistency condition or relaxing the sequential specification of the object, we consider a combination of the two. Namely, we consider the *multiplicity* relaxation [14] which allows for multiple concurrent *Dequeue* operations to return the same element, and we prove that that the implementation of such a FIFO queue is set-linearizable. It is necessary to consider the set-sequential specification of the queue because it is impossible to define a sequential execution of the relaxed FIFO queue with multiplicity that does not violate the sequential specification, namely two non concurrent *Dequeue* operations cannot return the same element.

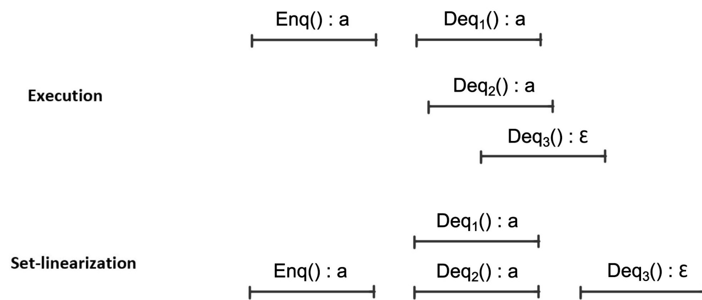


Figure 3.7: Example of a set-linearizable execution of the relaxed queue with multiplicity.

3.3.1 Algorithm Pseudocode and Description

Only the algorithm of the *Dequeue* operation is different from the Algorithm in Section 3.2. In the implementation of the relaxed queue, we do not require the unicity of the sequence numbers of the *Dequeue* operations. From this point on, we denote `Exact-Queue` the implementation of the FIFO queue in Algorithm 6-7 in Section 3.2, and `Relaxed-Queue` the implementation of the relaxed queue based on the `Exact-Queue` with the changes described in Algorithm 8.

Algorithm 8: Relaxed-Queue: implementation of the wait-free queue with multiplicity (Dequeue pseudo-code for process p).

```
1  Function Dequeue()
2      num ← deqCounter.MaxRead()
3      if deqOps[num].Read() ≠ (⊥, ⊥) then
4          deqCounter.MaxWrite(num + 1)
5          num ← num + 1
6      if num ≥ 1 then
7          UpdateTree(num − 1)
8      FinishDeq(num)
9      (h, id) ← deqOps[num].Read()
10     if id = ⊥ then
11         return ϵ
12     else
13         (ret, −) ← items[id][h]
14         return ret
```

We use a max register object for $deqCounter$ instead of the previously used $Fetch\&Inc$. Multiple concurrent *Dequeue* operations retrieve the same sequence number *num* from $deqCounter$ as long as $deqOps[num]$ remains unchanged. A *Dequeue* operation takes the sequence number $num+1$ only after the *Dequeue* operations with the sequence number *num* are completed (i.e. $deqOps[num] \neq (\perp, \perp)$). Thus, we relinquish the need for a helping mechanism for slow *Dequeue* operations since such an operation would have to be completed by another operation with the same sequence number before the next sequence number is assigned.

If a process retrieves the value *num* from *deqcounter* at the beginning of a *Dequeue* then its sequence number *seq* is in $\{num, num+1\}$ depending the value of $deqops[num]$ it reads. If $deqOps[num]$ has been written, the operation increments $deqCounter$ using the *MaxWrite* primitive, and takes the sequence number $num+1$, otherwise its sequence number is *num*. Similarly to Algorithm 6, the operation then executes the necessary steps to write $deqOps[seq]$ where $seq \in \{num, num+1\}$ is the sequence number of the operation. Meaning that the process executes $UpdateTree(seq-1)$ if the *Dequeue* operation with the sequence number $seq-1$ exists, to ensure that the root of the tree has an accurate value. Then, the process executes $FinishDeq(seq)$, after which $deqOps[seq]$ is set to a value different than its initial value. If $DeqOps[seq] = (i, p)$ the *Dequeue* operation returns $items[p][i].val$, otherwise it returns ϵ. Several *Dequeue* operations may have the same sequence number, and thus return the same value. The design of the algorithm ensures that two *Dequeue* operations can have the same sequence number only if they are concurrent. In the following, we consider the implementation of the relaxed FIFO queue with `multiplicity` and give the detailed proof for set-linearizability as well as the property of wait-freedom and worst-case step complexity of $O(\log n)$ for both *Enqueue* and *Dequeue* operations.

3.3.2 Algorithm Properties

Let E be an execution of `Relaxed-Queue`. The sequence number of a *Dequeue* operation corresponds to the value of *num* during the execution of line 9 of Algorithm 8. The sequence number of a *Dequeue* operations is no longer necessarily unique because multiple instances can retrieve the same sequence number *num* from *deqCounter*.

Lemma 3.3.1. *A partial order between Dequeue operations is provided by their sequence number. This order respects the real-time order.*

Proof. Let deq_1 and deq_2 be two *Dequeue* operations by process p_1 and p_2 respectively. Let seq_1 be the sequence number of deq_1 and seq_2 be the sequence number of deq_2. We prove that if deq_1 precedes deq_2 in real-time order, then $seq_1 < seq_2$.

deq_1 completes before deq_2 is invoked, thus p_1 executes the function $FinishDeq(num)$, after which a value has necessarily been written to $deqOps[num]$ where num is the sequence number of deq_1. Therefore, if deq_2 retrieves the same sequence number as deq_1 at line 2 of Algorithm 8, the test at line 3 would fail and the process would increment the value of *deqCounter* and *num* (lines 4 and 5). The claim follows.

□

Lemma 3.3.2. *Let deq_1 and deq_2 be two Dequeue operations. If deq_1 and deq_2 have the same sequence number, then they return the same value.*

Proof. Let j be the sequence number of both deq_1 and deq_2. Both operations return an element by reading the value stored in $deqOps[j]$ at line 9 of Algorithm 8. The claim follows. □

Observation 3.3.3. *Let deq be a Dequeue operation with the sequence number i, and let op be an operation that ends before deq is invoked. We have op ends before $deqOps[i]$ is written.*

Lemma 3.3.4. *Let C be a configuration of E. $\forall h > 0$ and $\forall i \geq 1$, if a Dequeue operation with the sequence number $i+h$ exists and it is complete at C, then a Dequeue operation with the sequence number i is complete at C.*

Proof. Consider the first configuration C where there is a complete *Dequeue* operation with the sequence number $i+h$, i.e.; $deqOps[i+h] \neq (\perp, \perp)$. Assume by contradiction that $deqOps[i]$ has its initial value at C.

Before a *Dequeue* operation can have the sequence number $i+h$, the condition in line 3 of Algorithm 8 needs to be verified for each sequence number in the range $[i, i+h[$. Meaning that before reaching the configuration C, a *Dequeue* operation had successfully executed the instance $FinishDeq(i)$ that writes $deqOps[i]$. There's a contradiction. □

As $deqOps[num]$ is updated only during the execution of the function $FinishDeq(num)$; the following observation is a consequence of Lemma 3.3.4.

Observation 3.3.5. *Before the first execution of $FinishDeq(i+h)$, $FinishDeq(i)$ has been executed.*

Each *Enqueue* operation *op* has a unique timestamp composed of an integer obtained by reading the Max register *enqCounter* during the execution of line 10, and the *id* of the process that executed the operation *op*.

Observation 3.3.6. *For each p, the timestamps of the elements written in the sub-array items[p] are monotonically increasing in accordance with their index in the array. In other terms, we have* $items[p][i].ts < items[p][i+1].ts$*.*

At any given configuration, the sub-queue of process p is the sub-array of $items[p]$ in the range $items[p][head[p].MaxRead()], ..., items[p][tail[p]-1]$. The following three Lemmas are the same exact properties as Lemmas 3.2.6, 3.2.8 and 3.2.9 since the pseudo-code involved is unchanged.

Lemma 3.3.7. *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked. Let* (st_1, id_1) *be the timestamp of* enq_1 *and* (st_2, id_2) *be the timestamp of* enq_2*. We have* $st_1 < st_2$*.*

Lemma 3.3.8. *Let enq denote the i-th Enqueue operation by a process p. Let* $ts = (st, p)$ *be the timestamp of enq. Let s be any node in the tree T in the path from the p-th leaf to the root of the tree. At any configuration C after enq ends and such that* $deqOps[j] \neq (i, p)$ *for each* $j \geq 0$*, we have that the timestamp stored at s is smaller than or equal to ts at C.*

Lemma 3.3.9. *Let enq be an Enqueue operation with the timestamp ts that enqueued* $items[p][i]$*. If* (i, p) *was written to* $deqOps[j]$ *by a process q, then the execution of line 25 of Algorithm 7 to read ts by q was executed after the invocation of enq.*

We say that the i-th *Enqueue* operation by a process p matches the *Dequeue* operation *deq* with sequence number j, if *deq* writes $deqOps[j] = (i, p)$ at some point in the execution. And we say that a *Dequeue* operation *deq* with sequence number j matches the i-th *Enqueue* operation by process p, if it returns $items[p][i]$.

Lemma 3.3.10. *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked. If* enq_2 *has a matching Dequeue operation* deq_2*, then* enq_1 *also has a matching Dequeue operation* deq_1*.*

Proof. By contradiction, we suppose that deq_2 exists and deq_1 does not. We denote ts_1 and ts_2 the timestamps associated with enq_1 and enq_2 respectively and num_2 the sequence number of deq_2. From Lemma 3.3.7, $ts_1 < ts_2$ because enq_1 ends before enq_2 begins.

And since enq_1 does not have a matching *Dequeue*, there is no $j \geq 0$ such that $deqOps[j] = (i, p)$ where $items[i][p]$ is enqueued by enq_1. Therefore, from Lemma 3.3.8, for any node s in the path in T from the p-th leaf to the root, the timestamp stored at s is smaller than or equal to ts_1 after enq_1 ends. In particular, for the root of the tree, the timestamp stored is smaller or equal to ts_1. From Lemma 3.3.9, the step of line 25 of Algorithm 7 to read the root of the tree before writing $deqOps[num_2]$ is executed after the invocation of enq_2 which is after the invocation of enq_1. Meaning that during this step, the timestamp at the root was smaller or equal to ts_1 contradicting the fact that $ts_1 < ts_2$. □

Lemma 3.3.11. *Let* enq_1 *and* enq_2 *be two Enqueue operations such that* enq_1 *ends before* enq_2 *is invoked and let* deq_1 *and* deq_2 *be the matching Dequeue operations to* enq_1 *and* enq_2 *respectively. We have that* deq_1 *has a lower sequence number than* deq_2*.*

Proof. We denote num_1 and num_2 the sequence numbers of deq_1 and deq_2 respectively, and ts_1 and ts_2 the timestamps of enq_1 and enq_2 respectively. By contradiction, we suppose that $num_1 > num_2$ ($num_1 \neq num_2$ from Lemma 3.3.2). Since enq_1 ends before enq_2 begins we have that $ts_1 < ts_2$ (Lemma 3.3.7).

And since $deqOps[i]$ are written in an increasing order of i according to Lemma 3.3.4, we have that $deqOps[num_2]$ is written before $deqOps[num_1]$. However, from Lemma 3.3.8, as long as $deqOps[num_1]$ has its initial value, then the timestamp stored at the root is smaller than or equal to ts_1. At the execution of line 25 of Algorithm 7 to compute the final value of $deqOps[num_2]$, the root has a timestamp smaller or equal to ts_1; contradicting the fact that $ts_1 < ts_2$. $\square$

Lemma 3.3.12. *Let deq be a Dequeue operation and let enq be an Enqueue operation that ends before deq is complete. Let C be a configuration of E where enq does not have a matching Dequeue operation deq' or deq' is not complete at C. If deq is complete at C, then deq does not return ϵ.*

Proof. By contradiction, we suppose that deq returns ϵ. Let i denote the sequence number of deq and ts denote the timestamp of enq. We also denote deq_i the operation that writes $deqOps[i]$.

Since deq returns ϵ, deq reads the value $(\epsilon, -1)$ in $deqOps[i]$ at line 9 of Algorithm 8. Therefore, during the execution of $FinishDeq(i)$, deq_i reads $(\epsilon, -1)$ at the root of the tree (line 27 of Algorithm 7). However, By Lemma 3.3.8, the timestamp at the root of the tree after the end of enq is smaller than or equal to ts. Since enq ends before deq starts, it specifically ends before deq is complete. Meaning that during the execution of line 25 of Algorithm 7 during the instance $FinishDeq(i)$ that writes $deqOps[i]$ during deq_i, the timestamp at the root of the tree was smaller than or equal to ts. We reach a contradiction because $(\epsilon, -1)$ is larger than any timestamp $(h, -)$ $\forall h \in \mathbb{N}$. $\square$

3.3.3 Set-linearizability

Let E denote a given execution of *Relaxed-Queue*. We classify every *Dequeue*() operation deq that appears in E to exactly one of the following types :

1. deq does not execute line 2 of Algorithm 8, or deq executes it but then verifies the condition in line 8 of Algorithm 8 and never executes the step at line 5.

2. deq has a sequence number j and `deqOps[j]` has the initial value $(\perp, \perp)$ in E.

3. deq has a sequence number j and `deqOps[j]` $\neq (\perp, \perp)$ in E.

We remove from E, any *Dequeue*() operation of type 1 and 2. We denote DEQ the set of *Dequeue*() operations of type 3. Let deq be any incomplete *Dequeue*() operation in DEQ and let j be its sequence number. We complete deq by returning the value v if `deqOps[j]` $= (i, id)$ in E and `items[id][i]` $= (v, -)$. Otherwise, we complete deq by returning the empty queue value ϵ. We consider the set DEQ_i of all instances of *Dequeue* that have the same sequence number i. Let deq_i be the operation in DEQ_i that writes $deqOps[i]$ during the call to $FinishDeq(i)$ at line 8 of Algorithm 8. Since $deqOps[i]$ is a CAS object, deq_i is unique. We denote DEQ' the set of all deq_i operations

for $i \geq 0$, i.e. $DEQ' = \{deq_i, \forall i \geq 0\}$. The operations in DEQ' are totally ordered according to their sequence number.

We remove every *Enqueue*() operation that does not execute line 13 of Algorithm 6 in E. We denote ENQ the set of *Enqueue*() operations that appear in E and that we do not remove. Every *Enqueue*() operation enq in ENQ is uniquely identified by a pair (i, id) meaning that enq is the i-th *Enqueue*() operation performed by the process id. We associate the *Dequeue*() operation in DEQ with sequence number i with the *Enqueue*() operation (j, id) such that `deqOps[i]` $= (j, id)$.

Let ENQ_d denote the *Enqueue*() operations in ENQ that have an associated *Dequeue*() operation in DEQ'. We associate each *Enqueue*() operations in ENQ_d with the sequence number of the corresponding *Dequeue*(). Thus, *Enqueue*() operations in ENQ_d are totally ordered according to the given sequence number.

We construct the set-linearization SL of the operations in E as follow:

1. First we insert the *Enqueue*() operations in ENQ_d one by one and according to their total order, denoted $enq_{i_1}, enq_{i_2} \ldots$ and so on. Notice that enq_{i_h} is the *Enqueue*() operation associated with the *Dequeue*() operation having the sequence number i_h in DEQ'. Assuming that $enq_{i_{h+1}}$ exists, we have $i_h < i_{h+1}$; and all the *Dequeue*() operations having a sequence number $i \in [i_h + 1, i_{h+1} - 1]$ return the value ϵ.

2. Then, we insert the *Dequeue*() operations in DEQ' one by one according to their the sequence number. For any sequence number k, If deq_k returns ϵ it is inserted immediately after deq_{k-1} if it exists, or at the beginning of SL otherwise. In the case where deq_k does not return ϵ, it is inserted immediately after the furthest point in SL following: (i) the previous deq_{k-1}, (ii) the matching *Enqueue* operation enq_{i_l} with $i_l = k$, and (iii) the last *Enqueue* operation that ends before the invocation of any *Dequeue* operation with the sequence number k (i.e. DEQ_k).

3. Let enq denote an *Enqueue* operation from the remaining *Enqueue*() operations with no matching *Dequeue* operations (i.e. $ENQ \setminus ENQ_d$). We insert enq after the last operation in ENQ_d and before the first *Dequeue*() operation deq_i in SL such that, there exists a *Dequeue* operation deq'_i in DEQ_i that starts after enq ends (or at the end of SL if such *Dequeue*() does not exist). If multiple operations from $ENQ \setminus ENQ_d$ are inserted at the same point, then they are ordered according to their real-time order.

4. For $i \geq 0$, we insert all *Dequeue* operations in $DEQ_i \setminus \{deq_i\}$ at the same point as deq_i.

For two operations op_1 and op_2, we denote $op_1 <_{SL} op_2$ when op_1 precedes op_2 in the set-linearization SL.

Lemma 3.3.13. *Let op_1 and op_2 be two Enqueue operations in E such that op_1 ends before op_2 is invoked. op_1 precedes op_2 in SL.*

Proof. First, consider the case where both operations do not have matching *Dequeue*() operations. From set-linearization rule 3, an *Enqueue* operation that does not have a

matching *Dequeue* operation is inserted before the first *Dequeue* operation deq_i in SL such that there exists an operation deq'_i in DEQ_i that starts after enq ends; or at the end of L if such *Dequeue* operation does not exist. If op_1 is inserted at the end of SL, then op_2 is also inserted at the end of SL after op_1, because op_2 starts after op_1 ends and there is no *Dequeue* operation that starts after op_1 ends. We suppose that there exists a *Dequeue* operation deq_i such that op_1 is inserted immediately before deq_i. If op_2 is inserted at the end of SL, the claim is trivial. So let deq_j be a *Dequeue* operation such that op_2 is inserted immediately before deq_j. We have $op_1 <_{ro} op_2 <_{ro} deq'_j$. Meaning that $deq_j = deq_i$ or $deq_i <_{SL} deq_j$, because both operations deq'_i and deq'_j start after op_1 ends, and deq'_i is the first such operation in SL. Therefore, $op_1 <_{SL} op_2$ according to their real time execution order following set-linearization rule 3.

Next, if op_1 has a matching *Dequeue*() operation but op_2 does not, we have that op_2 is inserted after the last *Enqueue*() operation that has a matching *Dequeue*() operation in SL. The case where op_1 does not have a matching *Dequeue*() operation but op_2 does, is impossible according to Lemma 3.3.10. We suppose that both op_1 and op_2 have matching *Dequeue*() operations, named respectively deq_1 and deq_2. From Lemma 3.3.11, we have that deq_1 has a smaller sequence number than deq_2. Therefore, from set-linearization rule 1, op_1 is before op_2 in SL. □

Lemma 3.3.14. *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. If enq has a matching Dequeue operation deq' with the sequence number i then $j < i$.*

Proof. By contradiction we suppose that $j \geq i$. We consider the configuration C where deq completes. According to Lemma 3.3.4, deq' also has been completed at C. Meaning that $deqOps[i] \neq (\bot, \bot)$ at C. However, from the hypothesis, enq is not invoked until after deq finishes. Contradicting the fact that deq' is the matching *Dequeue* operation of enq. □

Lemma 3.3.15. *Let deq be a Dequeue operation with the sequence number j and let enq be an Enqueue operation invoked after deq returns. We suppose that enq has a matching Dequeue operation deq' with the sequence number i. We have that any Dequeue operation with a sequence number $l < j$ is before enq in SL.*

Proof. By contradiction, we suppose that there exists *Dequeue* operations with sequence numbers strictly smaller than j that are after enq in SL, and let deq_l be the first of these operations in SL.

We suppose that deq_l returns ϵ. Since deq_l is the first *Dequeue* operation with a sequence number smaller than j that is inserted after enq. We have that $deq_{l-1} <_{SL} enq$. From set-linearization rule 2, deq_l is inserted immediately after deq_{l-1} (if it exists). Therefore, $deq_l <_{SL} enq$. There is a contradiction.

We suppose that deq_l does not return ϵ. Let enq_l be the matching *Enqueue* operation to deq_l. From Lemma 3.3.14, we have that $j < i$. Therefore, $l < j < i$. Thus, $enq_l <_{SL} enq$. Furthermore, we have $deq_{l-1} <_{SL} enq$ because deq_l is the first operation with a sequence number smaller than j inserted after enq in SL. Therefore, from set-linearization rule 2, there exists a *Dequeue* operation deq'_l such that $enq <_{ro} deq'_l$. Consequently, $deq_j <_{ro} enq <_{ro} deq'_l$. Contradicting the fact that $l < j$ (Lemma 3.3.1). □

Theorem 3.3.16. *Let op_1 and op_2 be two operations in E such that op_1 ends before op_2 is invoked. op_1 precedes op_2 in SL.*

Proof. Four cases have to be studied according to the type of operations.

1. op_1 and op_2 are two *Dequeue*() operations. Since op_1 ends before op_2 begins, the sequence number i_1 of op_1 is strictly smaller than the sequence number i_2 of op_2 (Lemma 3.3.1). From set-linearization rule 2 and rule 4, we have op_1 is before op_2 in SL.

2. The case where op_1 and op_2 are *Enqueue*() operations is proved by Lemma 3.3.13.

3. op_1 is an *Enqueue* operation and op_2 is a *Dequeue*() operation. Let i denote the sequence number of op_2. First, consider the case that op_2 does not return ϵ. In the case where $op_1 \in ENQ_d$, from set-linearization rule 2 and 4, op_2 is inserted after the last *Enqueue* operation that ends before every *Dequeue* operation in DEQ_i starts. Therefore, op_2 is inserted after op_1 in SL. In the case where $op_1 \notin ENQ_d$, from set-linearization 3, it is inserted before the first *Dequeue* operation deq_i such that there exists deq'_i in DEQ_i that starts after op_1 ends; or at the end of SL if such *Dequeue* does not exist. Thus op_1 is inserted before op_2.

   Next, consider the case where op_2 returns ϵ. Every *Dequeue* operation with the sequence number i returns ϵ and are inserted at the same point in SL (set-linearization rule 4). Let deq_i denote the *Dequeue* operation that writes $deqOps[i]$. We have that op_1 ends before deq_i is complete (Observation 3.3.3). By Lemma 3.3.12, op_1 has a matching *Dequeue* operation deq, and deq is complete before deq_i is complete. And since deq is complete before deq_i is complete, we have that $j < i$ where j is the sequence number of deq (Observation 3.3.5). Therefore, from set-linearization rule 2, deq is inserted before op_2. Thus, from set-linearization rule 1, $op_1 <_{SL} deq <_{SL} deq_i$. And from set-linearization rule 4, deq_i and op_2 are insterted at the same point; i.e. $op_1 <_{SL} op_2$.

4. Finally, we suppose that op_1 is a *Dequeue*() operation and that op_2 is an *Enqueue*() operation. Let j denote the sequence number of op_1. If op_2 does not have a matching *Dequeue* operation, from set-linearization rule 3, it is inserted before the first *Dequeue* operation deq_k in SL such that there exists a *Dequeue* operation deq'_k in DEQ_k that starts after op_2 ends or at the end of SL if such operation does not exist. By definition, all the operations in DEQ_j are concurrent. Hence, there is no *Dequeue* operation in DEQ_j that starts after op_2 ends because such operation cannot be in contention with op_1 which ends before op_2 starts. Therefore, if deq_k exists it is after op_1 in SL. Thus, op_2 is inserted after op_1 in SL.

   Next, consider that op_2 has a matching *Dequeue* operation deq with the sequence number i. If op_1 returns ϵ, from the set-linearization rule 2, we have op_1 is inserted immediately after deq_{j-1} the *Dequeue* operation with the previous sequence number (or the beginning of SL if it does not exist). And from Lemma 3.3.15, we have that deq_l is inserted before op_2 for any $l < j$. In particular, we have that deq_{j-1} is inserted before op_2. Therefore, op_1 is inserted before op_2.
   We suppose that op_1 does not return ϵ. From set-linearization rule 2, op_1 is inserted after (i) deq_{j-1}, (ii) the matching *Enqueue* operation enq_j and after (iii)

the last *Enqueue* operation enq' that ends before any *Dequeue* operation with the sequence number j starts. We show that op_2 is inserted after all these three operations. From Lemma 3.3.15, we have that deq_{j-1} is inserted before op_2 (i). From Lemma 3.3.14, we have that $j < i$ meaning that enq_j is linearized before op_2 according to the total order of the sequence numbers of their matching *Dequeue* operations (ii). And since op_1 ends before op_2 starts, $enq' <_{ro} op_2$. Therefore, $enq' <_{SL} op_2$ because we have shown that the set-linearization of the *Enqueue* operations respects the real time execution order (iii). The claim follows.

□

3.3.4 FIFO Queue Specification

In this section, we show that the *Dequeue* operations in a set-linearization SL of an execution of the *Relaxed-Queue* follow the FIFO order.

Lemma 3.3.17. *Let deq be a Dequeue operation that returns $v \neq \epsilon$. There exists an Enqueue(v) denoted enq that such that enq is before deq in SL, and there is no Dequeue operation deq' that also returns v such that deq' is not inserted at the same point as deq in SL.*

Proof. First, we prove that enq exists. Since deq returns $v \neq \epsilon$, it has read a value (j, p) in $deqOps[i]$ where i is the sequence number of deq (line 9 of Algorithm 8). Meaning that $items[p][j] = v$ and the *Enqueue* operation that enqueued v denoted enq, is the j-th instance of *Enqueue* by process p. From set-linearization rule 2, the matching *Dequeue* operation to enq is inserted after enq. Therefore, deq is either the matching operation to enq or has the same sequence number, and for both cases, deq is inserted after enq.

Let deq' be an operation that also returns v. Since deq' reads $deqOps[i]$, it has the same sequence number i as deq. From set-linearization rule 4, it is inserted at the same point as deq. □

Lemma 3.3.18. *Let enq_1 and enq_2 be two Enqueue operations such that $enq_1 <_{SL} enq_2$. If enq_2 has a matching Dequeue deq_2, then enq_1 has a matching Dequeue deq_1 and $deq_1 <_{SL} deq_2$.*

Proof. By contradiction, we suppose that enq_1 does not have a matching *Dequeue* operation. From set-linearization rule 3, enq_1 is inserted after all *Enqueue* operations in ENQ_d. Especially, enq_1 is inserted after enq_2. There is a contradiction. And from set-linearization rule 1, enq_1 and enq_2 are inserted according to the order of the sequence numbers of their matching *Dequeue* operations. The claim follows. □

From the two previous Lemmas 3.3.17-3.3.18, we have the following theorem.

Theorem 3.3.19. *Let deq be a Dequeue operation in SL. If deq does not return ϵ, then it returns the element enqueued by the first Enqueue operation in SL that does not have a matching Dequeue operation inserted before deq.*

Lemma 3.3.20. *Let deq_ϵ be a Dequeue operation that returns ϵ. And let enq be an Enqueue operation inserted before deq_ϵ in SL. We have that enq has a matching Dequeue operation deq that is also inserted before deq_ϵ in SL.*

Proof. First, we show that *enq* has a matching *Dequeue* operation *deq*. By contradiction, we suppose that *enq* is in $ENQ \setminus ENQ_d$. From set-linearization rule 3, *enq* is inserted before the first *Dequeue* operation deq_l in SL such that there exists an operation deq'_l that starts after *enq* ends; or at the end of SL if deq'_l does not exist. The case where *enq* is inserted at the end of SL is trivial because it contradicts the fact that *enq* is inserted before deq_ϵ. So deq'_l exists. By lemma 3.3.12 deq'_l does not return ϵ. Since $enq <_{SL} deq_\epsilon$, we have $deq'_l <_{SL} deq_\epsilon$ Hence, deq_ϵ has a greater sequence number than deq'_l from set-linearization rule 2. Thus, deq_ϵ is complete after deq'_l is complete (Lemma 3.3.4). We conclude by lemma 3.3.12, that deq_ϵ does not return ϵ. There is a contradiction. Thus, *enq* has a matching *Dequeue* operation denoted *deq*.

In the following, we establish that *deq* is inserted before deq_ϵ. Let i denote the sequence number of deq_ϵ and let j be the sequence number of *deq*. We denote deq_j the operation that writes $deqOps[j]$. By contradiction, we assume that $i < j$ (i.e. *deq* is inserted after deq_ϵ). Let deq_k be the first *Dequeue* operation inserted after *enq* with k its sequence number. Such an operation exists as $enq <_{SL} deq_\epsilon$. We have $k \leq i$, according to the set-linearization rule 2. Assume that deq_k returns ϵ. If $k = 0$ then no operation is inserted before deq_k; in this case, there is a contradiction. Otherwise ($k \geq 1$), there is no *Enqueue* operation inserted after deq_{k-1} and before deq_k because deq_k is inserted immediately after deq_{k-1} (set-linearization rule 2). This contradicts the fact that deq_k is the first *Dequeue* operation inserted after *enq*. Hence deq_k does not return ϵ. We conclude that $k < i$.

deq_k does not match *enq* as we assume that *deq* is inserted after deq_ϵ. From set-linearization rule 2, deq_k can only be inserted after *enq* because *enq* terminates before the invocation of an operation deq'_k with the same sequence number k. Since $k < i$, deq'_k is complete before deq_ϵ is complete (Lemma 3.3.4). Therefore, deq_ϵ is complete afte *enq* ends. Thus, by Lemma 3.3.12, deq_ϵ cannot return ϵ if $j > i$. There is a contradiction. □

3.3.5 Step Complexity

In this section, we establish that the *Enqueue* and *Dequeue* operations implemented in the *Relaxed-Queue* both have a worst-case step complexity of $O(\log n)$.

Lemma 3.3.21. *A process executes $O(\log n)$ steps during the execution of either an Enqueue operation or Dequeue operation.*

Proof. Lemmas 3.2.21 and 3.2.21.1 hold for the *Relaxed-Queue*. Therefore, the claim follows for *Enqueue* operations.

Let *deq* denote an instance of the *Dequeue* operation implemented in Algorithm 8. The number of steps executed during *deq* is dependent on the cost of the *UpdateTree* function (Line 7 of Algorithm 8), in which a call to *Propagate* can be executed. From Lemma 3.2.21, the number of steps executed during an instance of *Propagate* is $O(\log n)$. The claim follows. □

3.4 Discussion

We have presented a wait-free implementation of a k-multiple dequeuer n-multiple enqueuer FIFO queue. The worst case step complexity of the *Enqueue* operation is

in $O(\log n)$ and the *Dequeue* operation is in $O(k \log n)$. Meaning, that as long as the number k of dequeuer processes is constant, our implementation has logarithmic step complexity, which improves on the previous upper bound of $O(\sqrt{n})$. While we focused on theoretical evaluations of step complexity, it could also be of interest to compare the algorithm empirically to other FIFO implementations to gauge its applicative relevance.

Any queue implementation has a limitation regarding space complexity because of the requirement to store all the enqueued elements that have not been dequeued. Simply by considering an execution where a process only executes *Enqueue* operations, we can show a lower bound on space complexity in the number of elements present in the queue. Besides this limitation, there also seems to be a trade-off between step and space complexity in the implementations that appear in the literature. For instance, David [16] implements a single enqueuer queue with a *constant* step complexity but with infinite space complexity. But then, it is argued in [16], that it is possible to bound the space complexity of their implementation to the detriment of the step complexity that would reach $O(n)$.

Some implementations propose memory reclaiming schemes in which data that is no longer useful is discarded (i.e. dequeued elements). In [46], Yang et al. propose such a scheme based on the *epoch-based reclamation* in [21] to manage the memory of non-blocking lists. The performance of the wait-free queue implemented in [46] is measured empirically, and it is shown that the implementation manages to outperform other prior queue implementations regardless of the overhead generated by the memory usage optimization. We do not consider the issue of optimizing the space complexity in the scope of this work because of the intricacies that seem to correlate with balancing both the step and space complexities of a wait-free queue implementation, and we leave the question for future work.

Then, to the best of our knowledge, we presented the first relaxed FIFO queue with logarithmic step complexity where every process can perform both *Enqueue*(v) and *Dequeue*$()$ operations. It remains an open question whether it is possible to implement an exact wait-free linearizable FIFO queue with worst-case logarithmic step complexity without restriction on the number of enqueuers and dequeuers or to implement a relaxed FIFO queue in constant or near-constant step complexity.

Chapter 4

Conclusion

In this thesis, we study the possibility of improving the complexity of concurrent object implementations by relaxing their sequential specification. In particular, we focused on three common objects, the counter, max register, and FIFO queue.

We studied both upper and lower bounds of these relaxed objects to have a clear understanding, as much as possible, of the extent the relaxations can improve the implementation of a shared object and bring forth any limitations to this approach.

First, we study how allowing wait-free linearizable implementations of the counter and max register objects to return approximate values, rather than accurate ones, may improve their step complexity.

We consider the k-multiplicative-accurate max register and the k-multiplicative-accurate counter, where read operations are allowed a margin of error of a multiplicative factor of k. We give a wait-free linearizable k-multiplicative-accurate counter implementation for $k \geq n$ with constant amortized step complexity where n is the number of processes.

We also show that by bounding the execution, we are able to implement the k-multiplicative-accurate counter for $k \geq \sqrt{n}$ in a wait-free linearizable manner and with a worst-case step complexity of $O(min(\log(\log(m+1)), n))$ where m represents the bound on the number of CounterIncrement operations during an execution. Both implementations offer an exponential improvement on the complexities of their best exact counterparts in the state of the art.

Then, we study the lower bounds of the complexity of the k-multiplicative-accurate counter and max register in both their bounded and unbounded variations. We obtain the result that when the approximation parameter k does not depend on the number of processes, relaxing counter semantics by allowing inaccuracy of a multiplicative factor cannot asymptotically reduce the amortized step complexity of unbounded counters by more than a logarithmic factor. We also prove that our bounded k-multiplicative-accurate max register is optimal and matches the lower bound.

When it comes to the FIFO queue, we investigate whether it is possible to implement a logarithmic worst-case step complexity wait-free implementation that does not suffer from concurrency constraints. Therefore, we present a wait-free FIFO queue implementation that supports n enqueuers and k dequeuers where the worst-case step complexity of an Enqueue operation is in $O(\log n)$ and where the complexity of the Dequeue operation depends on the level of concurrency during the execution and is $O(k \log n)$ in the worst-case scenario.

We then rely on the relaxation of the FIFO queue semantics to show that allowing concurrent Dequeue operations to retrieve the same element results in an implementation with O(log n) worst-case step complexity for both the Enqueue and Dequeue operations.

Perspectives and prospects

There remains a few open problems around the results we presented that can be explored.

In the case of the k-multiplicative-accurate counter, depending on the parameter k, we do not know how the relaxation affects the implementation when $k \in]\sqrt{n/2}, n[$. Although our implementation of the unbounded relaxed counter can achieve constant amortized step complexity for $k \geq \sqrt{n}$ when the executions are long enough, a small gap still remains for the possible values of the approximation parameter k.

On a more high-level aspect, we have presented many cases where the relaxation of shared objects achieves better theoretical complexity results than exact objects. However, it is often the case that the relaxations are closely dependent on the nature of the object and do not necessarily translate into a large set of objects. It could be interesting to attempt to classify different types of relaxations to understand how they correlate with each other as well as how they relate to the different classes of weakened consistency conditions.

Bibliography

[1] Yehuda Afek, Eli Gafni, and Adam Morrison. "Common2 Extended to Stacks and Unbounded Concurrency". In: *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*. PODC '06. Denver, Colorado, USA: Association for Computing Machinery, 2006, pp. 218–227. ISBN: 1595933840. DOI: 10.1145/1146381.1146415. URL: https://doi.org/10.1145/1146381.1146415.

[2] Yehuda Afek, Guy Korland, and Eitan Yanovsky. "Quasi-Linearizability: Relaxed Consistency for Improved Concurrency". In: *14th International Conference on Principles of Distributed Systems (OPODIS)*. Vol. 6490. Lecture Notes in Computer Science. Springer, 2010, pp. 395–410.

[3] Amitanand Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. "On the Availability of Non-Strict Quorum Systems". In: *Proceedings of the 19th International Conference on Distributed Computing*. DISC'05. Cracow, Poland: Springer-Verlag, 2005, pp. 48–62. ISBN: 3540291636. DOI: 10.1007/11561927_6. URL: https://doi.org/10.1007/11561927_6.

[4] *Approx count distinct (transact-sql)*. https://bit.ly/2Gyxgwa.

[5] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. "Polylogarithmic concurrent data structures from monotone circuits". In: *J. ACM* 59.1 (2012), 2:1–2:24.

[6] James Aspnes, Maurice Herlihy, and Nir Shavit. "Counting Networks and Multi-Processor Coordination". In: *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*. STOC '91. New Orleans, Louisiana, USA: Association for Computing Machinery, 1991, pp. 348–358. ISBN: 0897913973. DOI: 10.1145/103418.103421. URL: https://doi.org/10.1145/103418.103421.

[7] James Aspnes et al. "Lower Bounds for Restricted-Use Objects". In: *SIAM J. Comput.* 45.3 (2016), pp. 734–762.

[8] Hagit Attiya, Armando Castaneda, and Danny Hendler. "Nontrivial and Universal Helping for Wait-Free Queues and Stacks". In: *19th International Conference on Principles of Distributed Systems (OPODIS 2015)*. Ed. by Emmanuelle Anceaume, Christian Cachin, and Maria Potop-Butucaru. Vol. 46. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 1–16. ISBN: 978-3-939897-98-9. DOI: 10.4230/LIPIcs.OPODIS.2015.31. URL: http://drops.dagstuhl.de/opus/volltexte/2016/6620.

[9] Hagit Attiya and Arie Fouren. "Adaptive and efficient algorithms for lattice agreement and renaming". In: *SIAM Journal on Computing* 31.2 (2001), pp. 642–664.

[10] Hagit Attiya and Danny Hendler. "Time and Space Lower Bounds for Implementations Using k-CAS". In: *IEEE Trans. Parallel Distrib. Syst.* 21.2 (2010), pp. 162–173.

[11] Hagit Attiya and Jennifer L. Welch. "Sequential Consistency versus Linearizability". In: *ACM Trans. Comput. Syst.* 12.2 (May 1994), pp. 91–122. ISSN: 0734-2071. DOI: 10.1145/176575.176576. URL: https://doi.org/10.1145/176575.176576.

[12] Mirza Ahad Baig et al. "Long-Lived Counters with Polylogarithmic Amortized Step Complexity". In: *33rd International Symposium on Distributed Computing, (DISC)*. Vol. 146. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 3:1–3:16.

[13] *Bigquery: Hyperloglog++ functions in standard sql.* https://bit.ly/2V1gDfA.

[14] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. *Relaxed Queues and Stacks from Read/Write Operations*. 2020. arXiv: 2005.05427 [cs.DC].

[15] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. "Unifying Concurrent Objects and Distributed Tasks: Interval-Linearizability". In: *J. ACM* 65.6 (Nov. 2018). ISSN: 0004-5411. DOI: 10.1145/3266457. URL: https://doi.org/10.1145/3266457.

[16] Matei David. "A Single-Enqueuer Wait-Free Queue Implementation". In: vol. 3274. Oct. 2004, pp. 132–143. ISBN: 978-3-540-23306-0. DOI: 10.1007/978-3-540-30186-8_10.

[17] David Eisenstat. "Two-enqueuer queue in Common2". In: (June 2008).

[18] Faith Ellen et al. "A Complexity-Based Hierarchy for Multiprocessor Synchronization: [Extended Abstract]". In: *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. PODC '16. Chicago, Illinois, USA: Association for Computing Machinery, 2016, pp. 289–298. ISBN: 9781450339643. DOI: 10.1145/2933057.2933113. URL: https://doi.org/10.1145/2933057.2933113.

[19] Panagiota Fatourou and Nikolaos D. Kallimanis. "Highly-Efficient Wait-Free Synchronization". In: *Theor. Comp. Sys.* 55.3 (Oct. 2014), pp. 475–520. ISSN: 1432-4350. DOI: 10.1007/s00224-013-9491-y. URL: https://doi.org/10.1007/s00224-013-9491-y.

[20] Andreas Haas et al. "Local Linearizability for Concurrent Container-Type Data Structures". In: *CONCUR*. 2016.

[21] Timothy L. Harris. "A Pragmatic Implementation of Non-Blocking Linked-Lists". In: *Proceedings of the 15th International Conference on Distributed Computing*. DISC '01. Berlin, Heidelberg: Springer-Verlag, 2001, pp. 300–314. ISBN: 3540426051.

[22] Danny Hendler and Nir Shavit. "Operation-Valency and the Cost of Coordination". In: *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing*. PODC '03. Boston, Massachusetts: Association for Computing Machinery, 2003, pp. 84–91. ISBN: 1581137087. DOI: 10.1145/872035.872047. URL: https://doi.org/10.1145/872035.872047.

[23] Danny Hendler et al. "Upper and Lower Bounds for Deterministic Approximate Objects". In: *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. IEEE, 2021, pp. 438–448. DOI: 10.1109/ICDCS51616.2021.00049. URL: https://doi.org/10.1109/ICDCS51616.2021.00049.

[24] Thomas A. Henzinger et al. "Quantitative relaxation of concurrent data structures". In: *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2013, pp. 317–328.

[25] Maurice Herlihy. "Wait-free synchronization". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13.1 (1991), pp. 124–149.

[26] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[27] Prasad Jayanti. "On the Robustness of Herlihy's Hierarchy". In: *Proceedings of the Twelfth Annual ACM Symposium on Principles of Distributed Computing*. PODC '93. Ithaca, New York, USA: Association for Computing Machinery, 1993, pp. 145–157. ISBN: 0897916131. DOI: 10.1145/164051.164070. URL: https://doi.org/10.1145/164051.164070.

[28] Prasad Jayanti and Srdjan Petrovic. "Logarithmic-Time Single Deleter, Multiple Inserter Wait-Free Queues and Stacks". In: *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science*. FSTTCS '05. Hyderabad, India: Springer-Verlag, 2005, pp. 408–419. ISBN: 3540304959. DOI: 10.1007/11590156_33. URL: https://doi.org/10.1007/11590156_33.

[29] Prasad Jayanti, King Tan, and Sam Toueg. "Time and Space Lower Bounds for Nonblocking Implementations". In: *SIAM J. Comput.* 30.2 (Apr. 2000), pp. 438–456. ISSN: 0097-5397.

[30] Pankaj Khanchandani and Roger Wattenhofer. "On the Importance of Synchronization Primitives with Low Consensus Numbers". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking*. ICDCN '18. Varanasi, India: Association for Computing Machinery, 2018. ISBN: 9781450363723. DOI: 10.1145/3154273.3154306. URL: https://doi.org/10.1145/3154273.3154306.

[31] Christoph M. Kirsch, Michael Lippautz, and Hannes Payer. "Fast and Scalable, Lock-Free k-FIFO Queues". In: *Parallel Computing Technologies*. Ed. by Victor Malyshkin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 208–223. ISBN: 978-3-642-39958-9.

[32] Alex Kogan and Erez Petrank. "Wait-Free Queues with Multiple Enqueuers and Dequeuers". In: *SIGPLAN Not.* 46.8 (Feb. 2011), pp. 223–234. ISSN: 0362-1340. DOI: 10.1145/2038037.1941585. URL: https://doi.org/10.1145/2038037.1941585.

[33] Zongpeng Li. "Non-blocking implementations of Queues in asynchronous distributed shared-memory systems". In: (Jan. 2001).

[34] Adam Morrison and Yehuda Afek. "Fast Concurrent Queues for X86 Processors". In: *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '13. Shenzhen, China: Association for Computing Machinery, 2013, pp. 103–112. ISBN: 9781450319225. DOI: 10.1145/2442516.2442527. URL: https://doi.org/10.1145/2442516.2442527.

[35] Gil Neiger. "Set-Linearizability". In: *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '94. Los Angeles, California, USA: Association for Computing Machinery, 1994, p. 396. ISBN: 0897916549. DOI: 10.1145/197917.198176. URL: https://doi.org/10.1145/197917.198176.

[36] Matthieu Perrin et al. *On Composition and Implementation of Sequential Consistency*. 2016. DOI: 10.48550/ARXIV.1607.06258. URL: https://arxiv.org/abs/1607.06258.

[37] *Quick distinct count in oracle database 12cr1 (12.1.0.2)*. https://bit.ly/2EldEJD.

[38] Hamza Rihani, Peter Sanders, and Roman Dementiev. "MultiQueues: Simple Relaxed Concurrent Priority Queues". In: *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '15. Portland, Oregon, USA: Association for Computing Machinery, 2015, pp. 80–82. ISBN: 9781450335881. DOI: 10.1145/2755573.2755616. URL: https://doi.org/10.1145/2755573.2755616.

[39] Arik Rinberg and Idit Keidar. "Brief Announcement: Intermediate Value Linearizability: A Quantitative Correctness Criterion". In: *Proceedings of the 39th Symposium on Principles of Distributed Computing*. PODC '20. Virtual Event, Italy: Association for Computing Machinery, 2020, pp. 221–223. ISBN: 9781450375825. DOI: 10.1145/3382734.3405712. URL: https://doi.org/10.1145/3382734.3405712.

[40] Adones Rukundo, Aras Atalar, and Philippas Tsigas. "Monotonically Relaxing Concurrent Data-Structure Semantics for Increasing Performance: An Efficient 2D Design Framework". In: *33rd International Symposium on Distributed Computing, DISC*. Vol. 146. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, 31:1–31:15.

[41] Nir Shavit and Gadi Taubenfeld. "The Computability of Relaxed Data Structures: Queues and Stacks as Examples". In: *Distrib. Comput.* 29.5 (Oct. 2016), pp. 395–407. ISSN: 0178-2770. DOI: 10.1007/s00446-016-0272-0. URL: https://doi.org/10.1007/s00446-016-0272-0.

[42] Edward Talmage and Jennifer L. Welch. "Anomalies and Similarities among Consensus Numbers of Variously-Relaxed Queues". In: *Computing* 101.9 (Sept. 2019), pp. 1349–1368. ISSN: 0010-485X. DOI: 10.1007/s00607-018-0661-2. URL: https://doi.org/10.1007/s00607-018-0661-2.

[43] Edward Talmage and Jennifer L. Welch. "Improving Average Performance by Relaxing Distributed Data Structures". In: *28th International Symposium on Distributed Computing (DISC)*. Vol. 8784. Lecture Notes in Computer Science. Springer, 2014, pp. 421–438.

[44] Edward Talmage and Jennifer L. Welch. “Relaxed Data Types as Consistency Conditions”. In: *Algorithms* 11.5 (2018).

[45] Martin Wimmer et al. “The Lock-Free k-LSM Relaxed Priority Queue”. In: *SIGPLAN Not.* 50.8 (Jan. 2015), pp. 277–278. ISSN: 0362-1340. DOI: 10.1145/2858788.2688547. URL: https://doi.org/10.1145/2858788.2688547.

[46] Chaoran Yang and John Mellor-Crummey. “A Wait-Free Queue as Fast as Fetch-and-Add”. In: *SIGPLAN Not.* 51.8 (Feb. 2016). ISSN: 0362-1340. DOI: 10.1145/3016078.2851168. URL: https://doi.org/10.1145/3016078.2851168.