VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

SPECIALIZED PROJECT

STUDYING AND DEVELOPING NONBLOCKING DISTRIBUTED MPSC QUEUES

Major: Computer Science

THESIS COMMITTEE: 6 - ĐỒ ÁN CHUYÊN NGÀNH
SUPERVISORS: DR. DIỆP THANH ĐĂNG
ASSOC. PROF. DR. THOẠI NAM

—ooo—

STUDENT: ĐỖ NGUYỄN AN HUY - 2110193

HCMC, 06/2025

Disclaimers

I affirm that this specialized project is the product of my original research and experimentation. Any references, resources, results which this project is based on or a derivative work of have been given due citations and properly listed in the references section. All original contents presented are the culmination of my dedication and perserverance under the close guidance of my supervisors, Mr. Thoại Nam and Mr. Diệp Thanh Đăng, from the Faculty of Computer Science and Engineering, Ho Chi Minh City University of Technology. I take full responsibility for the accuracy and authenticity of this document. Any misinformation, copyright infringement or plagiarism shall be faced with serious punishment.

Acknowledgements

This thesis is the culmination of joint efforts coming from not only myself, but also my professors, my family, my friends and other teachers of Ho Chi Minh University of Technology.

I want to first acknowledge my university, Ho Chi Minh University of Tecnology. Throughout my four years of pursuing education here, I have built a strong theoretical foundation and earned various practical experiences. These all lend themselves well to the completion of this thesis. However, I have to say that the one person that plays the cornerstone in my acedemic foundation must be Mr. Diệp Thanh Đăng. Any of my works and achievements can only be built properly upon this cornerstone that he forever laid in my foundation. He has been the constant supervisor and reliable consultant through this capstone project, right from its inception, throughout its nurturing until the very end. The project couldn't have reached this stage of maturity without Mr. Đăng. Therefore, he has the rights to share whatever yields this project has to offer. I know it's getting informal, but if you ever read this, I want you to know that you surely have become one of the most important person in my life. You are the right blend of intellect, wisdom, organization, empathy, open-mindedness that I always seek in an academic companion, which surely secures you a special place in my heart. Whatever the outcome of what's about to happen, whoever I become in the future, I won't ever forget you and our magical moments when we're brewing up this project. I want to credit you for anything I achieve in front of just anyone, including myself. I have always craved for a companionship like Friedrich Engels and Karl Marx, maybe I've come quite close to it.

I want to thank Mr. Thoại Nam for providing all the facilities and infrastructure that this project necessitates. He essentially funded all of our idea-brewing grounds, where magic happens.

I also want to give my family the sincerest thanks for their emotional and financial support, without which I couldn't have whole-heartedly followed my research till the end.

Last but not least important, I want to thank my closest friends for their informal but ever-constant check-ups to make sure I didn't miss the timeline for this specialized project, which I usually don't have the mental capacity for.

Contents

Chapter I Introduction 9
1.1 Motivation 9
1.2 Objective 11
1.3 Scope 12
1.4 Research question 12
1.5 Thesis overview 13
1.6 Structure 13
Chapter II Background 15
2.1 Irregular applications 15
2.1.1 Actor model as an irregular application 15
2.1.2 Fan-out/Fan-in pattern as an irregular application 16
2.2 MPSC queue 17
2.3 Correctness condition of concurrent algorithms 18
2.4 Progress guarantee of concurrent algorithms 19
2.4.1 Blocking algorithms 19
2.4.2 Non-blocking algorithms 20
2.4.2.1 Lock-free algorithms 20
2.4.2.2 Wait-free algorithms 21
2.5 Popular atomic instructions in designing non-blocking algorithms 22
2.5.1 Fetch-and-add (FAA) 22
2.5.2 Compare-and-swap (CAS) 22
2.5.3 Load-link/Store-conditional (LL/SC) 23
2.6 Common issues when designing non-blocking algorithms 24
2.6.1 ABA problem 24
2.6.2 Safe memory reclamation problem 25
2.7 MPI-3 - A popular distributed programming library interface specifcation 26
2.7.1 MPI-3 RMA 26
2.7.2 MPI-RMA communication operations 26
2.7.3 MPI-RMA synchronization 27
2.8 Pure MPI - A porting approach of shared memory algorithms to distributed algorithms 28
Chapter III Related works 31
3.1 Non-blocking shared-memory MPSC queues 31
3.1.1 LTQueue 31
3.1.2 DQueue 33
3.1.3 WRLQueue 35
3.1.4 Jiffy 36
3.1.5 Remarks 37
3.2 Distributed MPSC queues 37
Chapter IV Distributed MPSC queues 40

4.1 Distributed one-sided-communication primitives in our distributed algorithm specification . . . 41
4.2 A simple baseline distributed SPSC . . . 43
4.3 dLTQueue - Straightforward LTQueue adapted for distributed environment . 46
4.3.1 Overview . . . 47
4.3.2 Data structure . . . 48
4.3.3 Algorithm . . . 50
4.4 Slotqueue - dLTQueue-inspired distributed MPSC queue with all constant-time operations . . . 56
4.4.1 Overview . . . 56
4.4.2 Data structure . . . 57
4.4.3 Algorithm . . . 59
Chapter V Preliminary results . . . 63
5.1 Benchmarking metrics . . . 63
5.1.1 Throughput . . . 63
5.1.2 Latency . . . 63
5.2 Benchmarking baselines . . . 64
5.3 Microbenchmark program . . . 64
5.4 Benchmarking setup . . . 64
5.5 Benchmarking results . . . 65
Chapter VI Conclusion & Future works . . . 67
Appendix A Theoretical aspects . . . 68
A.1 Terminology . . . 68
A.2 Preliminaries . . . 68
A.2.1 Linearizability . . . 68
A.2.1.1 Linearizable SPSC . . . 69
A.2.1.2 Linearizable MPSC queue . . . 70
A.2.2 ABA-safety . . . 70
A.2.3 Theoretical performance . . . 71
A.3 Theoretical proofs of the distributed SPSC . . . 71
A.3.1 Correctness . . . 71
A.3.1.1 ABA problem . . . 71
A.3.1.2 Memory reclamation . . . 71
A.3.1.3 Linearizability . . . 71
A.3.2 Progress guarantee . . . 74
A.3.3 Theoretical performance . . . 74
A.4 Theoretical proofs of dLTQueue . . . 75
A.4.1 Proof-specific notations . . . 75
A.4.2 Correctness . . . 77
A.4.2.1 ABA problem . . . 77
A.4.2.2 Memory reclamation . . . 78
A.4.2.3 Linearizability . . . 78
A.4.3 Progress guarantee . . . 91

A.4.4 Theoretical performance 91
A.5 Theoretical proofs of Slotqueue 91
A.5.1 Proof-specific notations 92
A.5.2 Correctness 93
A.5.2.1 ABA problem 93
A.5.2.2 Memory reclamation 96
A.5.2.3 Linearizability 97
A.5.3 Progress guarantee 103
A.5.4 Theoretical performance 103
References 104

List of Tables

Table 1 Specification of `MPI_Win_lock_all` and `MPI_Win_unlock_all`. 29
Table 2 Characteristic summary of existing shared memory MPSC queues. The cell marked with (*) indicates that our evaluation contradicts with the authors' claims. 31
Table 3 Characteristic summary of existing distributed MPSC queues.
R stands for remote operations and L stands for local operations.
(*) [1] claims that it's lock-free. 38
Table 4 Characteristic summary of our proposed distributed MPSC queues.
(1) n is the number of enqueuers.
(2) R stands for **remote operation** and L stands for **local operation**.
(*) The underlying SPSC is assumed to be our simple distributed SPSC in Section 4.2. 41
Table 5 Future works for the next semester 67
Table 6 Theoretical performance summary of our simple distributed SPSC. R means remote operations and L means local operations. 75
Table 7 Theoretical performance summary of dLTQueue. R means remote operations and L means local operations. 91
Table 8 Theoretical performance summary of Slotqueue. R means remote operations and L means local operations. 103

List of Images

Figure 1 Some programming patterns involving the MPSC queue data structure. . 10
Figure 2 An overview of this thesis. 13
Figure 3 Actor model visualization. 15
Figure 4 Fan-out/Fan-in pattern visualization. 16
Figure 5 Linerization points of method 1, method 2, method 3, method 4 happens at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially. 19
Figure 6 Blocking algorithm: When a process is suspended, it can potentially block other processes from making further progress. 20
Figure 7 Lock-free algorithm: All the live processes together always finish in a finite amount of steps. 21
Figure 8 Wait-free algorithm: Any live process always finishes in a finite amount of steps. 21
Figure 9 ABA problem in a linked-list stack. 24
Figure 10 Unsafe memory reclamation in a LIFO stack. 26
Figure 11 An illustration of passive target communication. Dashed arrows represent synchronization (source: [2]). 28
Figure 12 An illustration of our synchronization approach in MPI RMA. 30
Figure 13 LTQueue's structure. 32
Figure 14 Intuition on how timestamp-refreshing works. 33
Figure 15 DQueue's structure. 34
Figure 16 WRLQueue's structure. 35
Figure 17 WRLQueue's dequeue operation 36
Figure 18 Jiffy's structure. 36
Figure 19 AMQueue's structure. 39
Figure 20 dLTQueue's structure. 47
Figure 21 Basic structure of Slotqueue. 57
Figure 22 Microbenchmark results for enqueue operation. 65
Figure 23 Microbenchmark results for dequeue operation. 65
Figure 24 Microbenchmark results for total throughput. 66
Figure 25 dLTQueue's structure. 76
Figure 26 Basic structure of Slotqueue. 92

Chapter I Introduction

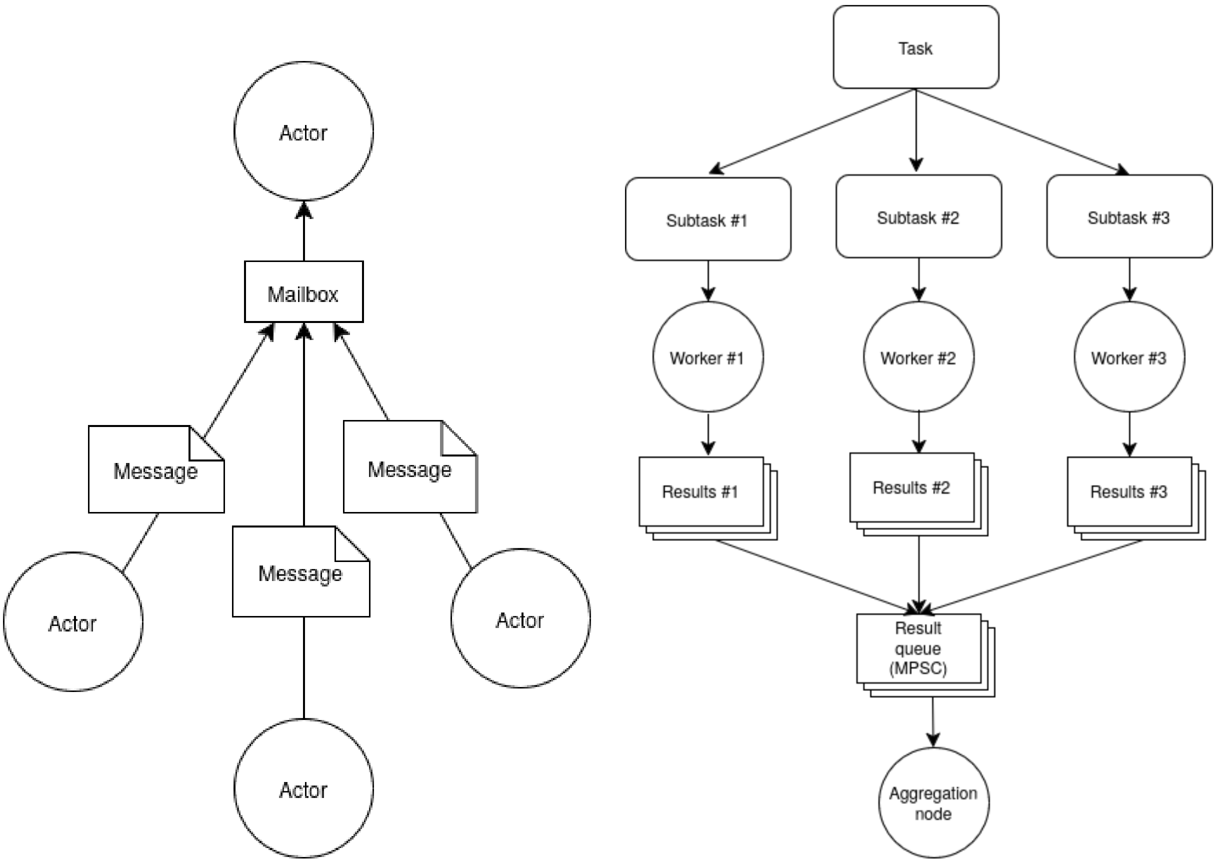
This chapter details the motivation for our research topic: "Studying and developing nonblocking distributed MPSC queues", based on which we set out the objectives and scope of this study. To summarize, we then come to the formulation of our research question and give a high-level overview of the thesis. We end this chapter with a brief description of the structure of the rest of this document.

1.1 Motivation

The demand for computation power has been increasing relentlessly. Increasingly complex computation problems arise and accordingly more computation power is required to solve them. Much engineering effort has been put forth towards obtaining more computation power. A popular topic in this regard is distributed computing: The combined power of clusters of commodity hardware can surpass that of a single powerful machine. To fully take advantage of the potential of distributed computing, specialized distributed algorithms and data structures need to be devised. Hence, there exists a variety of programming environments and frameworks that directly support the execution and development of distribute algorithms and data structures, one of which is the Message Passing Interface (MPI).

Traditionally, distributed algorithms and data structures use the usual Send/Receive message passing interface to communicate and synchronize between cluster nodes. Meanwhile, in the shared memory literature, atomic instructions are the preferred methods for communication and synchronization. This is due to the historical differences between the architectural support and programming models utilized in these two areas. For a class of problems known as regular applications, the use of the traditional Send/Receive interface suffices. However, this interface poses a challenge for irregular applications (Section 2.1). Fortunately, since the introduction of specialized networking hardware such as RDMA and the improved support of the remote memory access (RMA) programming model in MPI-3, this challenge has been alleviated: irregular applications can now be expressed more conveniently with an API that's similar to atomic operations in shared memory programming. This also implies that shared-memory algorithms and data structures can also be ported to distributed environments in a more straightforward manner. Since the design and development of shared-memory algorithms and data structures have been extensively studied, this has opened up a lot new research such as [3] on applying the principles of the shared memory literature to distributed computing.

Concurrent multi-producer single-consumer (MPSC) queue is one of those data structures that have seen many applications in shared-memory environments and plays the central role in many programming patterns, such as the actor model and the fan-out/fan-in pattern, as shown in Figure 1.



(a) Actor model. (b) Fan-out/Fan-in pattern.

Figure 1: Some programming patterns involving the MPSC queue data structure.

In the actor model, each process or compute node is represented as an actor. Each actor has a mailbox, which exhibits MPSC queue property: Other actors can send messages to the mailbox and the owner actor extracts messages and performs computation based on these messages. The fan-out/fan-in pattern involves splitting a task into multiple subtasks to workers, then the workers queue back the result to a result queue owned by another worker, who dequeues out the results to perform further processing, such as aggregation. These patterns can be potentially useful if they can be expressed efficiently in distributed environments. However, we have found dicussions of distributed MPSC queue algorithms in the current literature to be very scarce and scattered and as far as we know, none has really focused on designing a general-purpose distributed MPSC queue. The closest we found is the Berkeley Container Library (BCL) [4] that provides many general-purpose distributed data structures including a multi-producer multi-consumer (MPMC) queue and multi-producer/multi-consumer (MP/MC) queue, but sadly, no data structure for the specialized MPSC queue, while [1] discusses the design of a distributed multi-producer single-consumer (MPSC) queue specifically designed to support a pattern of message exchange. This presents an inhibition to programmers that want to either directly use the distributed MPSC queues or express programming patterns that inherently exhibit MPSC queue behaviors, they either have to work around the requirement or remodel their problems in another way. If a distributed MPSC queue is also provided as part of a library, this can in turn encourage many distributed applications and programming patterns that utilize the MPSC queues.

A desirable distributed MPSC queue algorithms should possess two favorable characteristics (1) scalability, the ability of an algorithm to utilize the highly concurrent nature of distributed clusters (2) fault-tolerance, the ability of an algorithm to continue running despite the failure of some compute nodes. Scalability is important for any concurrent algorithms, as one would never want to add more compute nodes just for performance to drop. Fault-tolerance, on the other hand, is especially more important in distributed computing, as failures can happen more frequently, such as network failures, node failures, etc. Fault-tolerance is concerned with a class of properties arisen in concurrent algorithms known as progress guarantee (Section 2.4). Non-blocking is a class of progress guarantee that ensures that the failure of one process does not cause the failure of the others.

Non-blocking MPSC queues and other FIFO variants, such as multi-producer multi-consumer (MPMC) queue, single-producer single-consumer (SPSC) queue, have been heavily studied in the shared memory literature, dating back from the 1980s-1990s [5], [6], [7] and more recently [8], [9]. It comes as no surprise that non-blocking algorithms in this domain are highly developed and optimized for performance and scalability. However, most research about distributed algorithms and data structures in general completely disregard the available state-of-the-art algorithms in the shared memory literature. Because shared-memory algorithms can now be straightforwardly ported to distributed context using this programming model, this presents an opportunity to make use of the highly accumulated research in the shared memory literature, which if adapted and mapped properly to the distributed context, may produce comparable results to algorithms exclusively devised within the distributed computing domain. Therefore, we decide to take this novel route to developing new non-blocking MPSC queue algorithms: Utilizing shared-memory programming techniques, adapting potential lock-free shared-memory MSPCs to design fault-tolerant and performant distributed MPSC queue algorithms. If this approach proves to be effective, a huge intellectual reuse of the shared-memory literature into the distributed domain is possible. Consequently, there may be no need to develop distributed MPSC queue algorithms from the ground up.

1.2 Objective

Based on what we have listed out in the previous section, we aim to:

- Investigate the principles underpinning the design of fault-tolerant and performant shared-memory algorithms.
- Investigate state-of-the-art shared-memory MPSC queue algorithms as case studies to support our design of distributed MPSC queue algorithms.
- Investigate existing distributed MPSC algorithms to serve as a comparison baseline.
- Model and design distributed MPSC queue algorithms using techniques from the shared-memory literature.

- Utilize the shared-memory programming model to evaluate various theoretical aspects of distributed MPSC queue algorithms: correctness and progress guarantee.
- Model the theoretical performance of distributed MPSC queue algorithms that are designed using techniques from the shared-memory literature.
- Collect empirical results on distributed MPSC queue algorithms and discuss important factors that affect these results.

1.3 Scope

The following narrows down what we're going to investigate in the shared-memory literature and which theoretical and empirical aspects we're interested in our distributed algorithms:

- Regarding the investigation of the design principles in the shared-memory literature, we focus on fault-tolerant and performant concurrent algorithm design using atomic operations and common problems that often arise in this area, namely, ABA problem and safe memory reclamation problem.
- Regarding the investigation of shared-memory MPSC queues currently in the literature, we focus on linearizable MPSC queues that follow strict FIFO semantics and support at least lock-free `enqueue` and `dequeue` operations.
- Regarding correctness, we concern ourselves with the linearizability correctness condition.
- Regarding fault-tolerance, we concern ourselves with the concept of progress guarantee, that is, the ability of the system to continue to make forward process despite the failure of one or more components of the system.
- Regarding algorithm prototyping, benchmarking and optimizations, we assume an MPI-3 setting.
- Regarding empirical results, we focus on performance-related metrics, e.g. throughput and latency.

1.4 Research question

Any research effort in this thesis revolves around this research question:

"How to utilize shared-memory programming principles to model and design distributed MPSC queue algorithms in a correct, fault-tolerant and performant manner?"

We further decompose this question into smaller subquestions:

1. How to model the correctness of a distributed MPSC queue algorithm?
2. Which factor contributes to the fault-tolerance and performance of a distributed MPSC queue algorithms?
3. Which shared-memory programming principle is relevant in modeling and designing distributed MPSC queue algorithms in a fault-tolerant and performant manner?
4. Which shared-memory programming principle needs to be modified to more effectively model and design distributed MPSC queue algorithms in a fault-tolerant and performant manner?

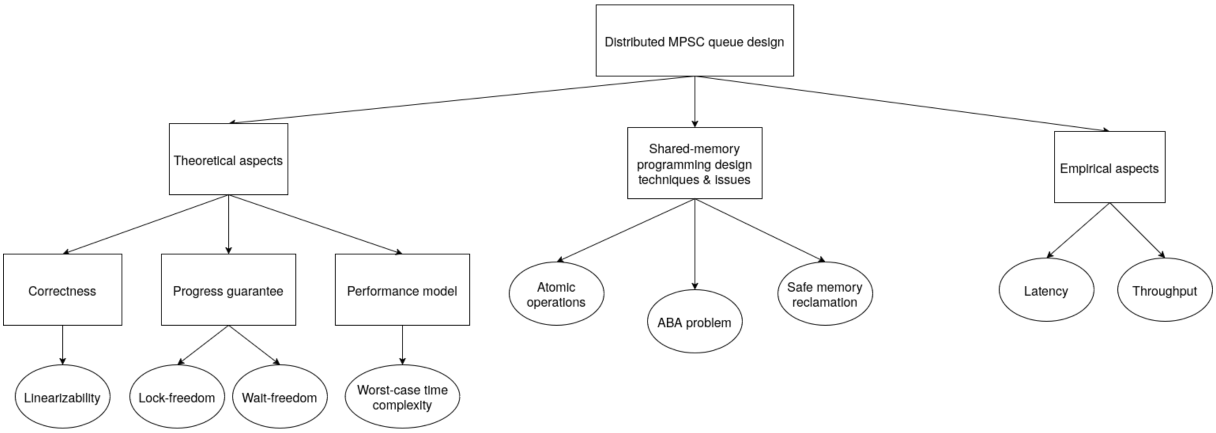


Figure 2: An overview of this thesis.

1.5 Thesis overview

An overview of this thesis is given in Figure 2.

This thesis explores the shared-memory programming model to design fault-tolerant and performant concurrent algorithms using atomic operations. Traditionally, in this aspect, two notorious problems often arise: ABA problem and safe memory reclamation. We investigate the traditional techniques used in the shared-memory literature to resolve these problems and appropriately adapt them to solve similar issues when designing fault-tolerant and performant distributed MPSC queues.

This thesis contributes two new distributed wait-free distributed MPSC queue algorithms. Theoretically, we're concerned with their correctness (linearizability), progress guarantee (lock-freedom and wait-freedom) which has an implication on their fault-tolerance and their theoretical performance, which is approximated by their number of remote operations and local operations.

This thesis concludes with an empirical analysis of our novel algorithms to see if their actual behavior matches our theoretical performance model, interprets these results and discusses its implication.

1.6 Structure

The rest of this report is structured as follows:

Chapter II discusses the theoretical foundation this thesis is based on. As mentioned, this thesis investigates the principles of shared-memory programming and the existing state-of-the-art shared-memory MPSC queues. We then explore the utilities offered by MPI-3 to implement distributed algorithms modeled by shared-memory programming techniques.

Chapter III surveys the shared-memory literature for state-of-the-art queue algorithms, specifically MPSC queues. We specifically focus on non-blocking shared-memory algorithms that have the potential to be adapted efficiently for distributed environment.

This chapter additionally surveys existing distributed MPSC algorithms to serve as a comparison baseline for our novel distributed MPSC queue algorithms.

Chapter IV introduces our novel distributed MPSC queue algorithms, designed using shared-memory programming techniques and inpsired by the selected shared-memory MPSC queue algorithms surveyed in Chapter III. It specifically presents our adaptation efforts of existing algorithms in the shared-memory literature to make their distributed implementations feasible and efficient.

Chapter V details our benchmarking metrics and elaborates our benchmarking setup. We aim to demonstrate some preliminary results on how well our novel MPSC queue algorithms, additionally compared to existing distributed MPSCs queues. Finally, we discuss important factors that affect the runtime properties of distributed MPSC queue algorithms, which have partly been explained by our theoretical analysis in Appendix A.

Chapter VI concludes what we have accomplished in this thesis and considers future possible improvements to our research.

Appendix A discusses various interesting theoretical aspects of our distributed MPSC queue algorithms in Chapter IV, specifically correctness (linearizability), progress guarantee (lock-freedom and wait-freedom), performance model. Our analysis of the algorithm's performance model helps back our empirical findings in Chapter V.

Chapter II Background

2.1 Irregular applications

Irregular applications are a class of programs particularly interesting in distributed computing. They are characterized by:

- Unpredictable memory access: Before the program is actually run, we cannot know which data it will need to access. We can only know that at run time.
- Data-dependent control flow: The decision of what to do next (such as which data to access next) is highly dependent on the values of the data already accessed, hence the unpredictable memory access property because we cannot statically analyze the program to know which data it will access. The control flow is inherently engraved in the data, which is not known until runtime.

Irregular applications are interesting because they demand special techniques to achieve high performance. One specific challenge is that this type of applications is hard to model in traditional MPI APIs using the Send/Receive interface. This is specifically because using this interface requires a programmer to have already anticipated communication within pairs of processes before runtime, which is difficult with irregular applications. The introduction of MPI remote memory access (RMA) in MPI-2 and its improvement in MPI-3 has significantly improved MPI's capability to express irregular applications comfortably. This will be explained further in Section 2.7.

2.1.1 Actor model as an irregular application

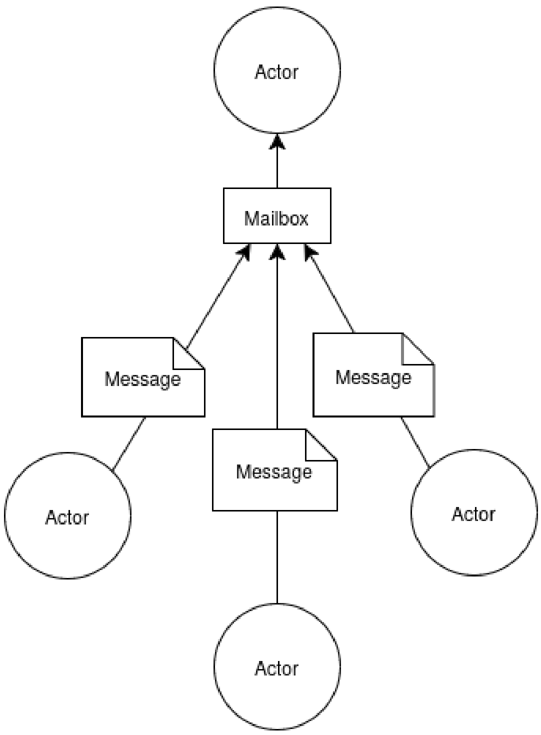


Figure 3: Actor model visualization.

Actor model in actuality is a type of irregular application supported by the concurrent MPSC queue data structure.

Each actor can be a process or a compute node in the cluster, carrying out a specific responsibility in the system. From time to time, there's a need for the actors to communicate with each other. For this purpose, the actor model offers a mailbox local to each actor. This mailbox exhibits MPSC queue behavior: Other actors can send messages to the mailbox to notify the owner actor and the owner actor at their leisure repeatedly extracts message from its mailbox. The actor model provides a simple programming model for concurrent processing.

The reasons why the actor model being an irregular application are straightforward to see:

- Unpredictable memory access: The cases in which one actor can anticipate which one of the other actors can send it a message are pretty rare and application-specific. As a general framework, in an actor model, the usual assumption is that any number of actors can try to communicate with an actor at some arbitrary time. By this nature, the communication pattern is unpredictable.
- Data-dependent control-flow: If an actor A sends a message to another actor B, and when B reads this message, B decides to send another message to another actor C. As we can see, the control-flow is highly engraved in the messages, or in other words, the messages drive the program flow, which can only be known at runtime.

2.1.2 Fan-out/Fan-in pattern as an irregular application

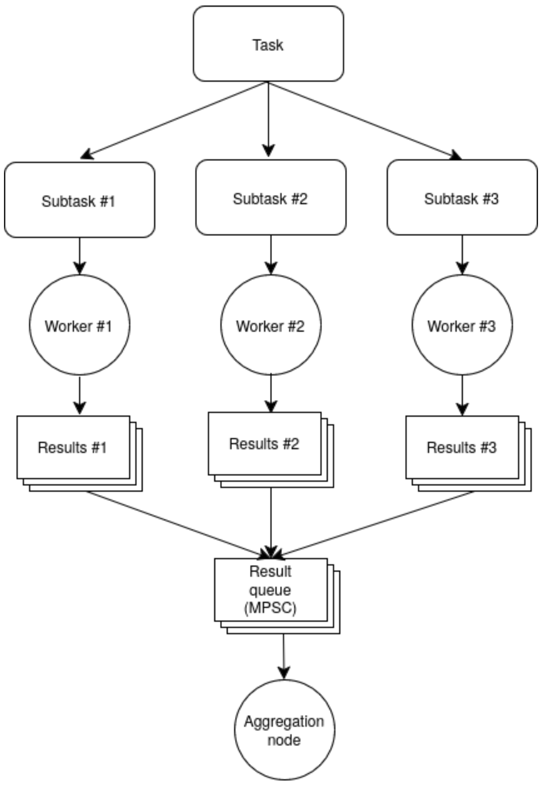


Figure 4: Fan-out/Fan-in pattern visualization.

The fan-out/fan-in pattern is another type of irregular application supported by the concurrent MPSC queue data structure.

In this pattern, there's a big task that can be splitted into subtasks to be executed concurrently on some work nodes. In the execution process, each worker produces a result set, each enqueued back to a result queue located on an aggregation node. The aggregation node can then dequeue from this result queue to perform further processing. Clearly, this result queue exhibits MPSC behavior.

The fan-out/fan-in pattern exhibits less irregularity than the actor model, however. Usually, the worker nodes and the aggregation node are known in advance. The aggregation node can anticipate Send calls from the worker nodes. Still, there's a degree of irregularity that this pattern exhibit: How can the aggregation node know how many Send calls a worker nodes will issue? This is highly driven by the task and the data involved in this task, hence, we have the data-dependent control-flow property. One can still statically calculate or predict how many Send calls a worker node will issue. Nevertheless, this is problem-specific. Therefore, the memory access pattern is somewhat unpredictable. Notice that if supported by a concurrent MPSC queue data structure, the fan-out/fan-in pattern is free from this burden of organizing the right amount of Send/Receive calls. Thus, combining with the MPSC queue, the fan-out/fan-in pattern becomes more general and easier to program.

We have seen the role MPSC queues play in supporting irregular applications. It's important to understand what really comprises an MPSC queue data structure.

2.2 MPSC queue

Multi-producer, single-consumer (MPSC) queue is a specialized concurrent first-in first-out (FIFO) data structure. A FIFO is a container data structure where items can be inserted into or taken out of, with the constraint that the items that are inserted earlier are taken out of earlier. Hence, it's also known as the queue data structure. The process that performs item insertion into the FIFO is called the producer and the process that performs items deletion (and retrieval) is called the consumer.

In concurrent queues, multiple producers and consumers can run concurrently. One class of concurrent FIFOs is the MPSC queue, where one consumer may run in parallel with multiple producers.

The reasons we're interested in MPSC queues instead of the more general multi-producer, multi-consumer (MPMC) queue data structures are that (1) high-performance and high-scalability MPSC queues are much simpler to design than MPMCs while (2) MPSC queues are powerful enough to solve certain problems, as demonstrated in Section 2.1. The MPSC queue in actuality is an irregular application in and out of itself:

- Unpredictable memory access: As a general data structure, the MPSC queue allows any process to enqueue and dequeue from at any time. By nature, its memory access patern is unpredictable.
- Data-dependent control-flow: The consumer's behavior is entirely dependent on whether and which data is available in the MPSC queue. The execution paths of MPSC queues can vary, based on the queue contention i.e. some processes may

backoff or retry some failed operations, this scenario often arise in lock-free data structures.

As an implication, some irregular applications can actually “push” the “irregularity burden” to the distributed MPSC queue, which is already designed for high-performance and fault tolerance. This provides a comfortable level of abstraction for programmers that need to deal with irregular applications.

2.3 Correctness condition of concurrent algorithms

Correctness of concurrent algorithms is hard to defined, regarding the semantics of concurrent data structures like MPSC queues. One effort to formalize the correctness of concurrent data structures is the definition of **linearizability**. A method call on the FIFO can be visualized as an interval spanning two points in time. The starting point is called the **invocation event** and the ending point is called the **response event**. **Linearizability** informally states that each method call should appear to take effect instantaneously at some moment between its invocation event and response event [10]. The moment the method call takes effect is termed the **linearization point**. Specifically, suppose the followings:

- We have n concurrent method calls m_1, m_2, ..., m_n.
- Each method call m_i starts with the **invocation event** happening at timestamp s_i and ends with the **response event** happening at timestamp e_i. We have $s_i < e_i$ for all $1 \leq i \leq n$.
- Each method call m_i has the **linearization point** happening at timestamp l_i, so that $s_i \leq l_i \leq e_i$.

Then, linerizability means that if we have $l_1 < l_2 < ... < l_n$, the effect of these n concurrent method calls m_1, m_2, ..., m_n must be equivalent to calling m_1, m_2, ..., m_n **sequentially**, one after the other in that order.

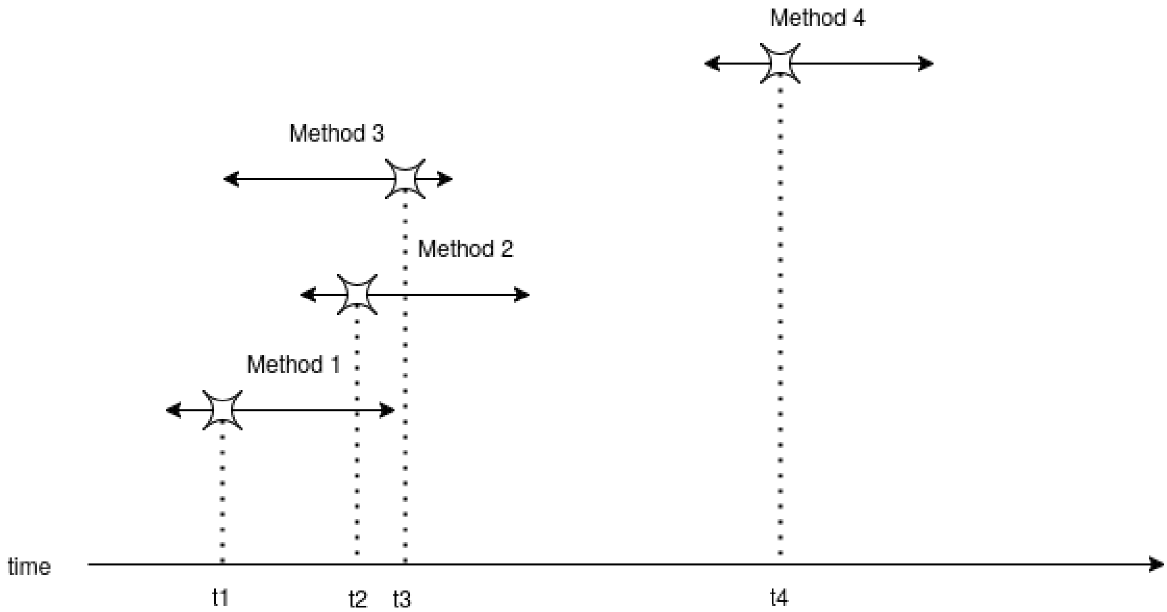


Figure 5: Linerization points of method 1, method 2, method 3, method 4 happens at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially.

Linearizability is widely used as a correctness condition because of (1) its composability (if every component in the system is linearizable, the whole system is linearizable [11]), which promotes modularity and ease of proof (2) its compatibility with human intuition, i.e. linearizability respects real-time order [11]. Naturally, we choose linearizability to be the only correctness condition for our algorithms.

2.4 Progress guarantee of concurrent algorithms

Progress guarantee is a criteria that only arises in the context of concurrent algorithms. Informally, it's the degree of hinderance one process imposes on another process from completing its task. In the context of sequential algorithm, this is irrelevant because there's only ever one process. Progress guarantee has an implication on an algorithm's performance and fault-tolerance, especially in adverse situations, as we will explain in the following sections.

2.4.1 Blocking algorithms

Many concurrent algorithms are based on locks to create mutual exclusion, in which only some processes that have acquired the locks are able to act, while the others have to wait. While lock-based algorithms are simple to read, write and verify, these algorithms are said to be **blocking**: One slow process may slow down the other faster processes, for example, if the slow process successfully acquires a lock and then the operating system (OS) decides to suspends it to schedule another one, this means until the process is awaken, the other processes that contend for the lock cannot continue.

Blocking is the weakest progress guarantee one algorithm can offer, it allows one process to impose arbitrary impedance to any other processes, as shown in Figure 6.

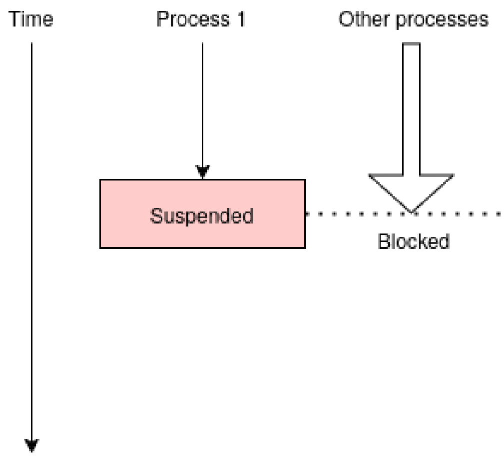


Figure 6: Blocking algorithm: When a process is suspended, it can potentially block other processes from making further progress.

Blocking algorithms introduces many problems such as:

- Deadlock: There's a circular lock-wait dependencies among the processes, effectively prevent any processes from making progress.
- Convoy effect: One long process holding the lock will block other shorter processes contending for the lock.
- Priority inversion: A higher-priority process effectively has very low priority because it has to wait for another low priority process.

Furthermore, if a process that holds the lock dies, this will render the whole program unable to make any progress. This consideration holds even more weight in distributed computing because of a lot more failure modes, such as network failures, node falures, etc.

Therefore, while blocking algorithms, especially those using locks, are easy to write, they do not provide **progress guarantee** because **deadlock** or **livelock** can occur and its use of mutual exclusion is unnecessarily restrictive. Forturnately, there are other classes of algorithms which offer stronger progress guarantees.

2.4.2 Non-blocking algorithms

An algorithm is said to be **non-blocking** if a failure or slow-down in one process cannot cause the failure or slow-down in another process. Lock-free and wait-free algorithms are two especially interesting subclasses of non-blocking algorithms. Unlike blocking algorithms, they provide stronger degrees of progress guarantees.

2.4.2.1 Lock-free algorithms

Lock-free algorithms provide the following guarantee: Even if some processes are suspended, the remaining processes are ensured to make global progress and complete in bounded time. In other words, a process cannot cause hinderance to the global progress of the program. This property is invaluable in distributed computing, one dead or suspended process will not block the whole program, providing fault-tolerance. Designing lock-free algorithms requires careful use of atomic instructions, such as Fetch-and-add (FAA), Compare-and-swap (CAS), etc which will be explained in Section 2.5.

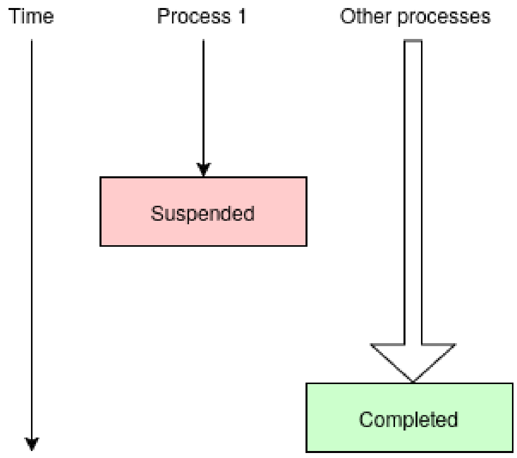


Figure 7: Lock-free algorithm: All the live processes together always finish in a finite amount of steps.

2.4.2.2 Wait-free algorithms

Wait-freedom offers the strongest degree of progress guarantee. It mandates that no process can cause constant hinderance to any running process. While lock-freedom ensures that at least one of the alive processes will make progress, wait-freedom guarantees that any alive processes will finish in finite number of steps. Wait-freedom can be desirable because it prevents starvation. Lock-freedom still allows the possibility of one process having to wait for another indefinitely, as long as some still makes progress.

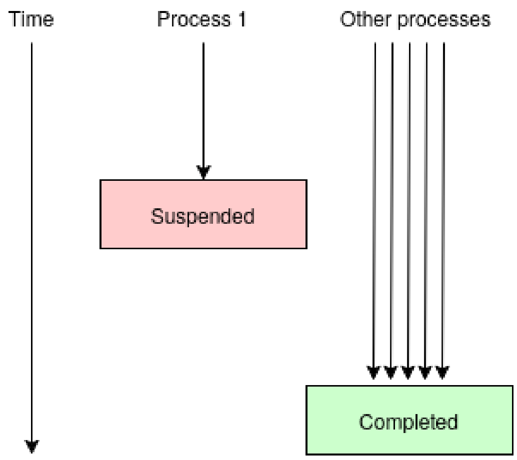


Figure 8: Wait-free algorithm: Any live process always finishes in a finite amount of steps.

2.5 Popular atomic instructions in designing non-blocking algorithms

In non-blocking algorithms, finer-grained synchronization primitives than simple locks are required, which manifest themselves as atomic instructions. Therefore, it's necessary to get familiar with the semantics of these atomic instructions and common programming patterns associated with them.

2.5.1 Fetch-and-add (FAA)

Fetch-and-add (FAA) is a simple atomic instruction with the following semantics: It atomically increments a value at a memory location x by a and returns the previous value just before the increment. Informally, FAA's effect is equivalent to the function in Procedure 1, assuming that the function is executed atomically.

Procedure 1: `int fetch_and_add(int* x, int a)`

```
old_value = *x
*x = *x + a
return old_value
```

Fetch-and-add can be used to create simple distributed counters.

2.5.2 Compare-and-swap (CAS)

Compare-and-swap (CAS) is probably the most popular atomic operation instruction. The reason for its popularity is (1) CAS is a **universal atomic instruction** with the **concensus number** of ∞, which means it's the most powerful atomic instruction [12] (2) CAS is implemented in most hardware (3) some concurrent lock-free data structures such as MPSC queues are more easily expressed using a powerful atomic instruction such as CAS.

The semantics of CAS is as follows. Given the instruction `CAS(memory location, old value, new value)`, atomically compares the value at `memory location` to see if it equals `old value`; if so, sets the value at `memory location` to `new value` and returns true; otherwise, leaves the value at `memory location` unchanged and returns false. Informally, its effect is equivalent to the function in Procedure 2.

Procedure 2: `bool compare_and_swap(int* x, int old_val, int new_val)`

```
if (*x == old_val)
    *x = new_val
    return true
return false
```

Compare-and-swap is very powerful and consequently, pervasive in concurrent algorithms and data structure.

Non-blocking concurrent algorithms often utilize CAS as follows. The steps 1-3 are retried until success.

1. Read the current value `old value = read(memory location)`.
2. Compute `new value` from `old value` by manipulating some resources associated with `old value` and allocating new resources for `new value`.
3. Call `CAS(memory location, old value, new value)`. If that succeeds, the new resources for `new value` remain valid because it was computed using valid resources associated with `old value`, which has not been modified since the last read. Otherwise, free up the resources we have allocated for `new value` because `old value` is no longer there, so its associated resources are not valid.

This scheme is, however, susceptible to the ABA problem, which will be discussed in Section 2.6.1.

2.5.3 Load-link/Store-conditional (LL/SC)

Load-link/Store-conditional is actually a pair of atomic instructions for synchronization.

Semantically, load-link returns a value currently located at a memory location x while store-conditional sets the memory location x to a value v if there's no other writes to x since the last load-link call, otherwise, the store-conditional call would fail.

Intuitively, LL/SC provides an easier synchronization primitive than CAS: LL/SC ensures that a store-conditional can only succeed if there's no access to a memory location, while CAS can still succeed in this case if the value at the memory location does not change. Due to this property, LL/SC is not vulnerable to the ABA problem (see Section 2.6.1). However, CAS is in fact as powerful as LL/SC, condersing that they can implement each other [12].

Practically, store-conditional can still fail even if there's no writes to the same memory location since the last load-link call. This is called a spurious failure. For example, consider the following generic sequence of events:

1. Thread X calls load-link on x and loads out v.
2. Thread X computes a new value v'.
3. Some *exceptional event* happens (discussed below). Assume that no other threads access x during this time.
4. Thread X calls store-conditional to store v' to x. It *should succeed* but *fails* anyways.

Exceptional events that can cause the store-conditional to fail spuriously include:

- Cache line flushing: If the cache line that caches the memory location x is written back to memory, logically, the memory location x has been accessed and therefore, the store-conditional fails.

- Context switch: If thread X is swapped out by the OS, cache lines may be invalidated and flushed out, which consequently leads to the first scenario.

LL/SC even though as powerful as CAS, is not as widespread as CAS, in fact, as of MPI-3, only CAS is supported.

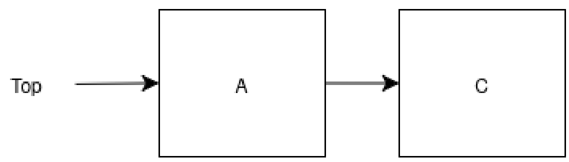
2.6 Common issues when designing non-blocking algorithms

2.6.1 ABA problem

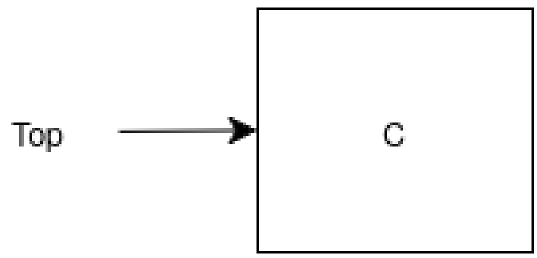
ABA problem is a notorious problem associated with the compare-and-swap atomic instruction. Because CAS is so widely used in non-blocking algorithms, ABA problem almost has to always be accounted for.

As a reminder, here's how CAS is often utilized in non-blocking concurrent algorithms: The steps 1-3 are retried until success.

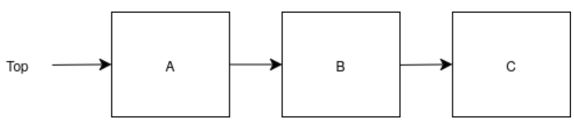
1. Read the current value `old value = read(memory location)`.
2. Compute `new value` from `old value` by manipulating some resources associated with `old value` and allocating new resources for `new value`.
3. Call `CAS(memory location, old value, new value)`. If that succeeds, the new resources for `new value` remain valid because it was computed using valid resources associated with `old value`, which has not been modified since the last read. Otherwise, free up the resources we have allocated for `new value` because `old value` is no longer there, so its associated resources are not valid.



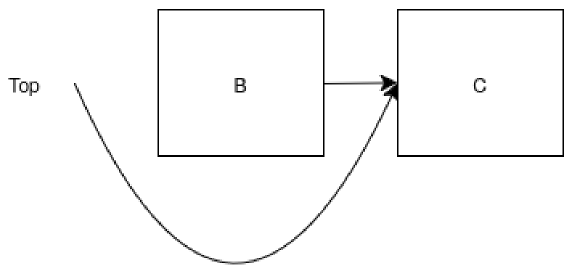
(a) Process X wants to pop a value, it observes Top = `A` and Top → next = `C` then suspends.



(b) Another process pops the value `A` and sets Top to `C`.



(c) Another process pushes two values `B` and `A` and sets Top to `A`.



(d) Process X successfully performs the pop by calling `CAS(&Top, A, C)`. `Top` no longer points to the top of the stack.

Figure 9: ABA problem in a linked-list stack.

As hinted, this scheme is susceptible to the notorious ABA problem. The following scenario illustrate and example of ABA problem:

1. Process 1 reads the current value of `memory location` and reads out `A`.
2. Process 1 manipulates resources associated with `A`, and allocates resources based on these resources.
3. Process 1 suspends.
4. Process 2 reads the current value of `memory location` and reads out `A`.
5. Process 2 `CAS(memory location, A, B)` so that resources associated with `A` are no longer valid.
6. Process 3 `CAS(memory location, B, A)` and allocates new resources associated with `A`.
7. Process 1 continues and `CAS(memory location, A, new value)` relying on the fact that the old resources associated with `A` are still valid while in fact they aren't.

ABA problem arises fundamentally because most algorithms assume a memory location is not accessed if its value is unchanged.

A specific case of ABA problem is given in Figure 9.

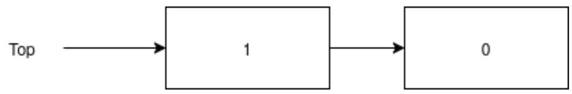
To safe-guard against ABA problem, one must ensure that between the time a process reads out a value from a shared memory location and the time it calls CAS on that location, there's no possibility another process has CAS-ed the memory location to the same value.

A simple scheme that's widely used practically and also in this thesis is the **unique timestamp** scheme. This scheme's idea is simple: for each shared memory location that is affected by CAS operations, we reserve some bits of this memory location for a monotonic counter. Each time a CAS operation is carried out, this counter is incremented. Theoretically, ABA problem would never happen because combining with this counter, the value of this memory location is always unique, due to the counter never repeats itself. However, practically, the counter can overflow and wrap-around to the same value and ABA problem would happen in this case. Therefore, the counter's range must be large enough so that this scenario can't virtually happen. Empirically, a counter of 32-bit should be enough. The drawback of this approach is that we have wasted 32 meaningful bits to avoid ABA problem.

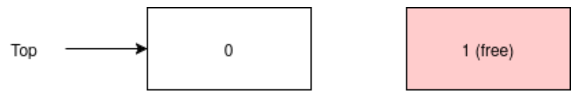
2.6.2 Safe memory reclamation problem

The problem of safe memory reclamation often arises in concurrent algorithms that dynamically allocate memory. In such algorithms, dynamically-allocated memory must be freed at some point. However, there's a good chance that while a process is freeing memory, other processes contending for the same memory are keeping a reference to that memory. Therefore, deallocated memory can potentially be accessed, which is erroneous.

An example of unsafe memory reclamation is given in Figure 10.



(a) Process X about to push a value onto the stack, already reading the top pointer but suspended.



(b) The top node is popped, the reference X holds is no longer valid. When X resumes, a freed memory location will be accessed.

Figure 10: Unsafe memory reclamation in a LIFO stack.

Solutions to this problem must ensure that memory is only freed when no other processes are holding references to it. In garbage-collected programming environments, this problem can be conveniently pushed to the garbage collector. In non-garbage-collected programming environments, however, custom schemes must be utilized.

2.7 MPI-3 - A popular distributed programming library interface specifcation

MPI stands for message passing interface, which is a **message-passing library interface specification**. Design goals of MPI includes high availability across platforms, efficient communication, thread-safety, reliable and convenient communication interface while still allowing hardware-specific accelerated mechanisms to be exploited [2].

2.7.1 MPI-3 RMA

RMA in MPI RMA stands for remote memory access. As introduced in the first section of Section Chapter II, RMA APIs is introduced in MPI-2 and its capabilities are further extended in MPI-3 to conveniently express irregular applications. In general, RMA is intended to support applications with dynamically changing data access patterns where the data distribution is fixed or slowly changing [2]. This is very similar to the properties of irregular applications as discussed in Section 2.1. In such applications, one process, based on the data it needs, knowing the data distribution, can compute the nodes where the data is stored. However, because data access pattern is not known, each process cannot know whether any other processes will access its data. Using the traditional Send/Receive interface, both sides need to issue matching operations by distributing appropriate transfer parameters. This is not suitable, as previously explain, only the side that needs to access the data knows all the transfer parameters while the side that stores the data cannot anticipate this.

2.7.2 MPI-RMA communication operations

RMA only requires one side to specify all the transfer parameters and thus only that side to participate in data communication.

To utilize MPI RMA, each process needs to open a memory window to expose a segment of its memory to RMA communication operations such as remote writes (`MPI_PUT`), remote reads (`MPI_GET`) or remote accumulates (`MPI_ACCUMULATE`,

`MPI_GET_ACCUMULATE`, `MPI_FETCH_AND_OP`, `MPI_COMPARE_AND_SWAP`) [2]. These remote communication operations only requires one side to specify.

2.7.3 MPI-RMA synchronization

Besides communication of data from the sender to the receiver, one also needs to synchronize the sender with the receiver. That is, there must be a mechanism to ensure the completion of RMA communication calls or that any remote operations have taken effect. For this purpose, MPI RMA provides **active target synchronization** and **passive target synchronization**. In this document, we're particularly interested in **passive target synchronization** as this mode of synchronization does not require the target process of an RMA operation to explicitly issue a matching synchronization call with the origin process, easing the expression of irregular applications [13].

In **passive target synchronization**, any RMA communication calls must be within a pair of `MPI_Win_lock`/`MPI_Win_unlock` or `MPI_Win_lock_all`/`MPI_Win_unlock_all`. After the unlock call, those RMA communication calls are guaranteed to have taken effect. One can also force the completion of those RMA communication calls without the need for the call to unlock using flush calls such as `MPI_Win_flush` or `MPI_Win_flush_local`.

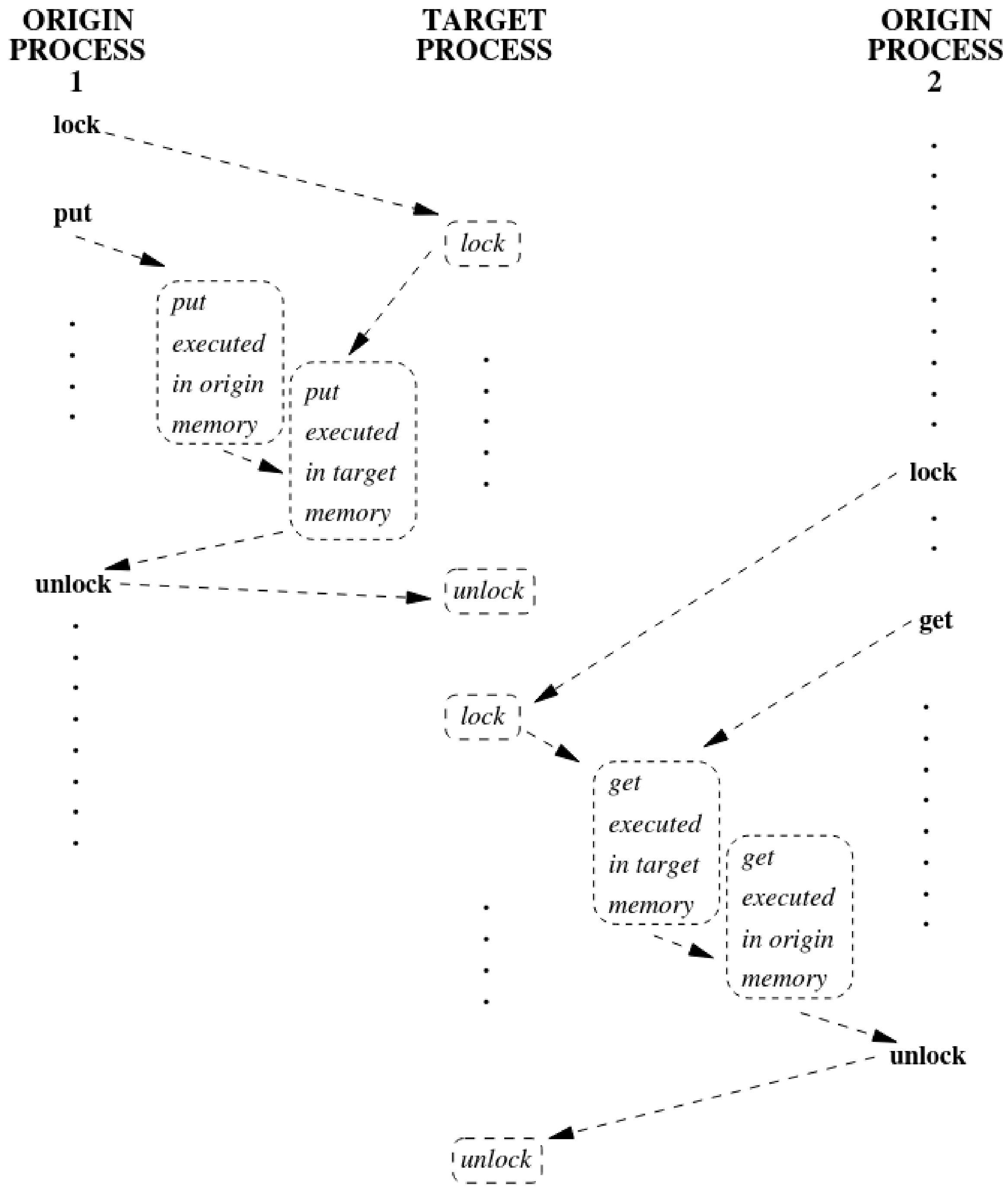


Figure 11: An illustration of passive target communication. Dashed arrows represent synchronization (source: [2]).

2.8 Pure MPI - A porting approach of shared memory algorithms to distributed algorithms

In pure MPI, we use MPI exclusively for communication and synchronization. With MPI RMA, the communication calls that we utilize are:

- Remote read: `MPI_Get`
- Remote write: `MPI_Put`
- Remote accumulation: `MPI_Accumulate`, `MPI_Get_accumulate`, `MPI_Fetch_and_op` and `MPI_Compare_and_swap`.

For lock-free synchronization, we choose to use **passive target synchronization** with `MPI_Win_lock_all/MPI_Win_unlock_all`.

In the MPI-3 specification [2], these functions are specified as in Table 1.

Operation	Usage
`MPI_Win_lock_all`	Starts and RMA access epoch to all processes in a memory window, with a lock type of `MPI_LOCK_SHARED`. The calling process can access the window memory on all processes in the memory window using RMA operations. This routine is not collective.
`MPI_Win_unlock_all`	Matches with an `MPI_Win_lock_all` to unlock a window previously locked by that `MPI_Win_lock_all`.

Table 1: Specification of `MPI_Win_lock_all` and `MPI_Win_unlock_all`.

The reason we choose this is 3-fold:

- Unlike **active target synchronization**, **passive target synchronization** does not require the process whose memory is being accessed by an MPI RMA communication call to participate in. This is in line with our intention to use MPI RMA to easily model irregular applications like MPSC queues.
- Unlike **active target synchronization**, `MPI_Win_lock_all` and `MPI_Win_unlock_all` do not need to wait for a matching synchronization call in the target process, and thus, is not delayed by the target process.
- Unlike **passive target synchronization** with `MPI_Win_lock/MPI_Win_unlock`, multiple calls of `MPI_Win_lock_all` can succeed concurrently, so one process needing to issue MPI RMA communication calls do not block others.

An example of our pure MPI approach with `MPI_Win_lock_all/MPI_Win_unlock_all`, inspired by [13], is illustrated in the following:

```
MPI_Win_lock_all(0, win);

MPI_Get(...); // Remote get
MPI_Put(...); // Remote put
MPI_Accumulate(..., MPI_REPLACE, ...); // Atomic put
MPI_Get_accumulate(..., MPI_NO_OP, ...); // Atomic get
MPI_Fetch_and_op(...); // Remote fetch-and-op
MPI_Compare_and_swap(...); // Remote compare and swap
...

MPI_Win_flush(...); // Make previous RMA operations take effects
MPI_Win_flush_local(...); // Make previous RMA operations take
effects locally
...

MPI_Win_unlock_all(win);
```

Listing 3: An example snippet showcasing our synchronization approach in MPI RMA.

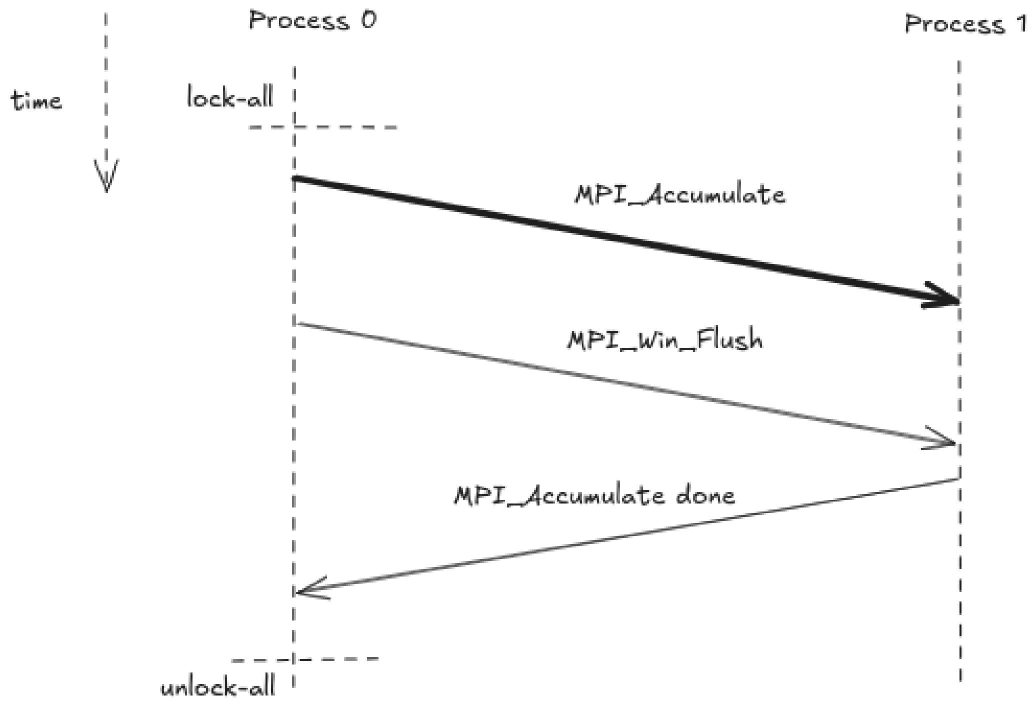


Figure 12: An illustration of our synchronization approach in MPI RMA.

Chapter III Related works

3.1 Non-blocking shared-memory MPSC queues

There exists numerous research into the design of non-blocking shared memory MPMCs and SPSCs. Interestingly, research into non-blocking MPSC queues are noticeably scarce. Although in principle, MPMC queues and SPSC queues can both be adapted for MPSC queues use cases, specialized MPSC queues can usually yield much more performance. In reality, we have only found 4 papers that are concerned with the direct support of lock-free MPSC queues: LTQueue [8], DQueue [14], WRLQueue [15] and Jiffy [9]. Table 2 summarizes the charateristics of these algorithms.

MPSC queues	**LTQueue** [8]	**DQueue** [14]	**WRLQueue** [15]	**Jiffy** [9]
ABA solution	Load-link/ Store-conditional	Incorrect custom scheme (*)	Custom scheme	Custom scheme
Safe memory reclamation	Custom scheme	Incorrect custom scheme (*)	Custom scheme	Custom scheme
Progress guarantee of dequeue	Wait-free	Wait-free	Blocking (*)	Wait-free
Progress guarantee of enqueue	Wait-free	Wait-free	Wait-free	Wait-free
Number of elements	Unbounded	Unbounded	Bounded	Unbounded

Table 2: Characteristic summary of existing shared memory MPSC queues. The cell marked with (*) indicates that our evaluation contradicts with the authors' claims.

3.1.1 LTQueue

To our knowledge, LTQueue [8] is the earliest paper that directly focuses on the design of a wait-free shared memory MPSC queue.

This algorithm is wait-free with $O(\log n)$ time complexity for both enqueues and dequeues, with n being the number of enqueuers due to a novel timestamp-update scheme and a tree-structure organization of timestamps.

The basic structure of LTQueue is given in Figure 13. In LTQueue, each enqueuer maintains an SPSC queue that only it and the dequeuer access. This SPSC queue must additionally support the `readFront` operation which returns the front element currently in the SPSC. The SPSC can be any implementations that conform to this interface. In the original paper, the SPSC is represented as a simple linked-list.

The rectangular nodes at the bottom in Figure 13 represents an enqueuer, whose SPSC contains items with 2 fields: `value` and `timestamp`. Every enqueuer has to timestamp its data before enqueueing. The timestamps can be obtained using a distributed counter shared by all the enqueuers.

The purpose of timestamping is to determine the order to dequeue the items from the local SPSCs. To efficiently maintain the timestamps and determine which SPSC to dequeue from first, a tree structure with a min-heap property is built upon the enqueuer nodes. The original algorithm leaves the exact representation of the tree open, for example, the arity of the tree, which is shown to be 2 in Figure 13. The circle-shaped nodes in this figure represents the nodes in this tree structure, which are shared by all processes. Each node stores the minimum timestamp along with the owner enqueuer's rank (an identifier given to a process) in the subtree rooted at that node. After every modification to the local SPSC, i.e. after an enqueue and a dequeue, the changes must be propagated up to the root node.

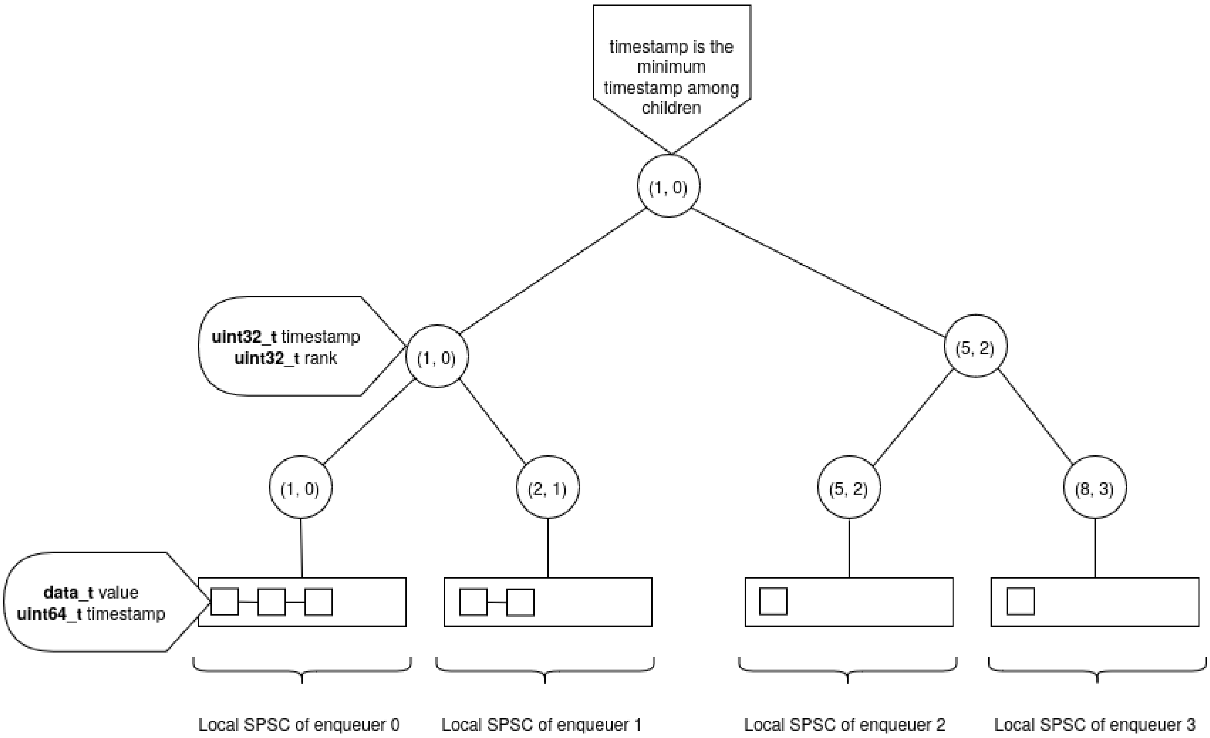


Figure 13: LTQueue's structure.

To dequeue, the dequeuer simply looks at the root node to determine the rank of the enqueuer to dequeue its SPSC.

The fundamental idea contributes to LTQueue's wait-freedom is the wait-free timestamp-propagation procedure. If there's a change to an enqueuer's SPSC, the timestamp of any nodes that lie on the path from the enqueuer to the root node are refreshed. The timestamp-refreshing procedure is simple:

1. Call load-link on the node's `(timestamp, rank)`.
2. Look at all the timestamps of the node's children and determine the minimum timestamp and its owner rank.
3. Call store-conditional to store the new minimum timestamp and the new owner rank to the current node.

Notice that due to contention, the timestamp-refreshing procedure can fail. In that case, the timestamp-propagation procedure simply retries the timestamp-refreshing procedure one more time. This second call, again, can fail. However, after this second call, the node's timestamp is guaranteed to be up-to-date. The intuition behind this is demonstrated in Figure 14. Furthermore, because every node is refreshed at most twice, the timestamp-refresh procedure should finish in a finite number of steps.

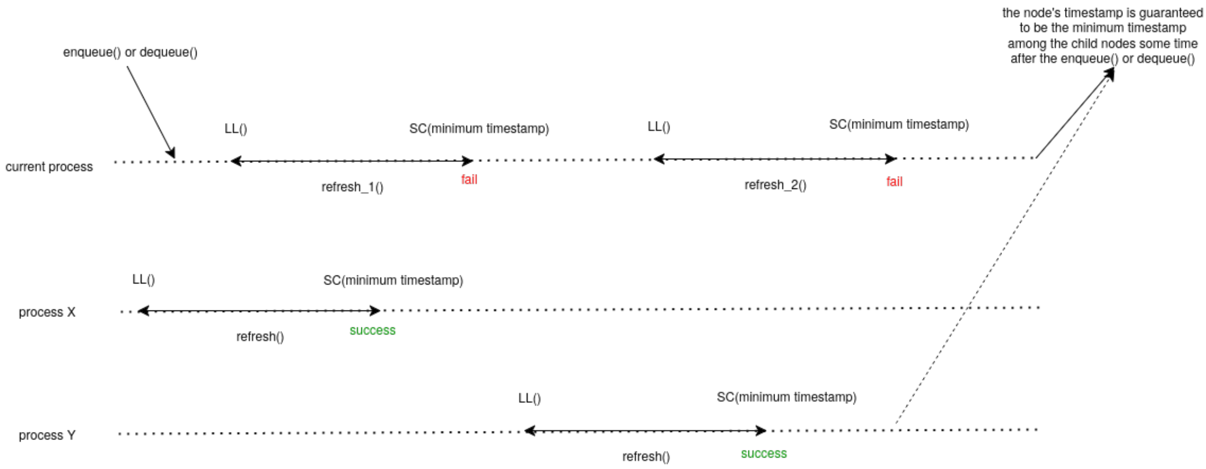


Figure 14: Intuition on how timestamp-refreshing works.

The LTQueue algorithm avoids ABA entirely by utilizing load-link/store-conditional. This represents a challenge to directly implementing this algorithm in distributed environment.

The memory reclamation responsibility is handled by the SPSC structure, which is pretty trivial with a custom scheme.

The design of each enqueuer maintaining a separate SPSC allows multiple enqueuer to successfully enqueues its data in parallel without stepping on the others' feet. This can potentially scale well to a large number of processes. However, scalability may be limite due to potentially growing contention during timestamp propagation. The performance of LTQueue in shared-memory environments may still have a lot of room for improvement, i.e. more cache-awareness design, avoiding unnecessary contention, etc. Nevertheless, their timestamp-refreshing scheme is interesting in and out of itself and can potentially inspire the design of new algorithms. In fact, LTQueue's idea is core to one of our optimized distributed MPSC queue algorithm, Slotqueue (Section 4.4).

3.1.2 DQueue

DQueue [14] focuses on optimizing performance, aiming to be cache-friendly and avoid expensive atomic instructions such as CAS.

The basic structure of DQueue is demonstrated in Figure 15.

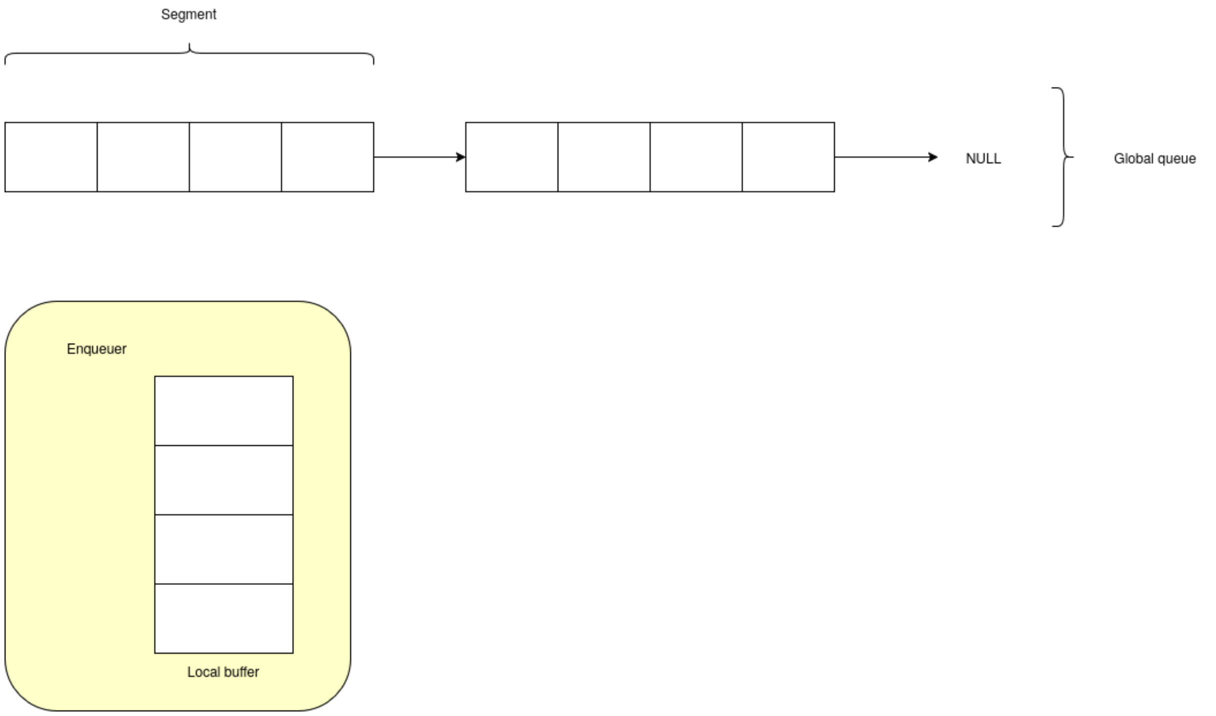


Figure 15: DQueue's structure.

The global queue where data is represented as a linked list of segments. A segment is simply a contiguous array of data items. This design allows for unbounded queue capacity while still allowing a fair degree of random access within a segment. This allows us to use indices to index elements in the queue, thus permitting the use of inexpensive FAA instructions to swing the head and tail indices.

Each enqueuer maintains a local buffer to batch enqueued items before flushing to the global queue. This helps prevent contention and play nice with the cache. To enqueue an item, an enqueuer simply FAA the head index to reserve a slot in the global queue, the obtained index is stored along with the data in the local buffer so that when flushing the local buffer, the enqueuer knows where to write the data into the global queue. Note that while flushing, an index may point to a not-yet-existent slot in the global queue. Therefore, new segments must be allocated on the fly and CAS-ed to the end of the queue.

The dequeuer dequeues the items by looking at the head index. If the queue is not empty but the slot at the head index is empty, the dequeuer utilize a helping mechanism by looking at all enqueuers to help them flush out the local buffer. After this, the head slot is guaranteed to be non-empty and the dequeuer can finally dequeues out this value.

The ABA problem is solved by relying on its safe memory reclamation scheme. In DQueue, CAS is only used to update the tail pointer to point to the newly allocated segment. Therefore, ABA problem in DQueue only involves internal manipulation of pointers to dynamically allocated memory. This means that if a proper memory reclamation scheme is used, ABA problem cannot occur.

DQueue relies on a dedicated garbage collection thread to reclaim segments that have been exhausted by the dequeuer. However, this should be a careful process as even though some segment has been exhausted, some enqueuers can still hold an index that references one these segments. DQueue implements this by reclaming all exhausted segments if there is no enqueuer holding an index referencing these segments. Unfortunately, we believe DQueue's scheme is unsafe. Specifically, as described, DQueue allows the garbage collection thread to reclaim non-adjacent segments in the global queue, without patching any of the next pointers. Any segment just before a reclaimed segment would point to a deallocated next segment. By definition, this segment was not reclaimed because it is referenced by an enqueuer. This means this enqueuer cannot traverse the next pointer chain to get to the end of the queue without accessing an already-deallocated segment.

If adapted to distributed environment, the flushing may be expensive, both from the point-of-view of the enqueuer and the dequeuer. If the dequeuer has to help every enqueuer to flush their local buffer, which should always result in at least one remote operation, the cost would be prohibitively high. Similarly, each flush requires the enqueuer to issue at least one remote operation, but this is at least acceptable as flushing is infrequent.

Still, we can still see that the pattern of maintaining a local buffer inside each enqueuer repeating throughout the literature, which we can definitely apply when designing distributed MPSC queues.

3.1.3 WRLQueue

WRLQueue [15] is a lock-free MPSC queue specifically designed for embedded real-time system. Its main purpose is to avoid excessive modification of storage space.

WRLQueue is simply a pair of buffers, one is worked on by multiple enqueuers and the other is worked on by the dequeuer. The structure of WRLQueue is shown in Figure 16.

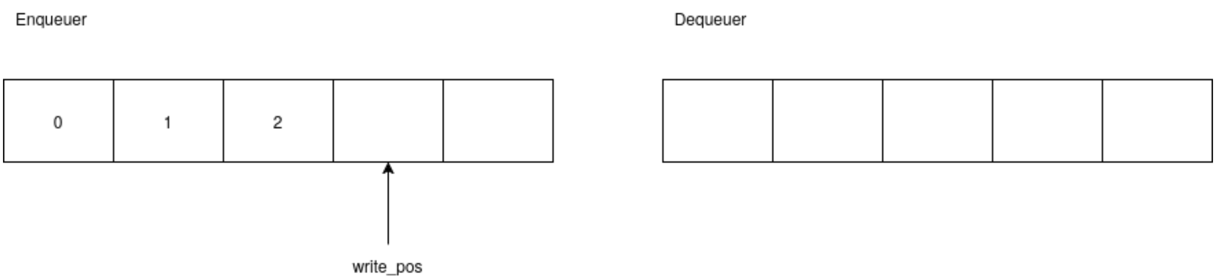


Figure 16: WRLQueue's structure.

The enqueuers batch their enqueues and write multiple elements onto the buffer at once. They use the usual scheme of FFA-ing the tail index (`write_pos` in Figure 16) to reserve their slots and write data items at their leisure.

The dequeuer upon invocation will swap its buffer with the enqueuer's buffers to dequeue from it, as in Figure 17. However, WRLQueue explicitly states that the dequeuer has to wait for all enqueue operations to complete in the other buffer before swapping. If an enqueue suspends or dies, the dequeuer will experience a slow-down, this clearly violates the property of non-blocking. Therefore, we believe that WRLQueue is blocking, concerning its dequeue operation.

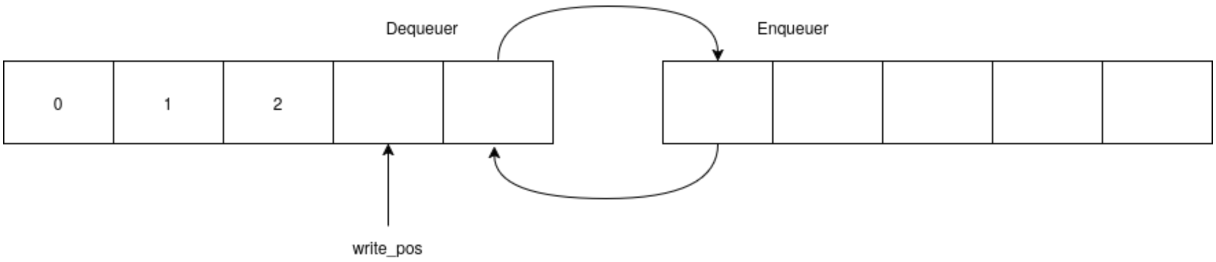


Figure 17: WRLQueue's dequeue operation

3.1.4 Jiffy

Jiffy [9] is a fast and memory-efficient wait-free MPSC queue by avoiding excessive allocation of memory.

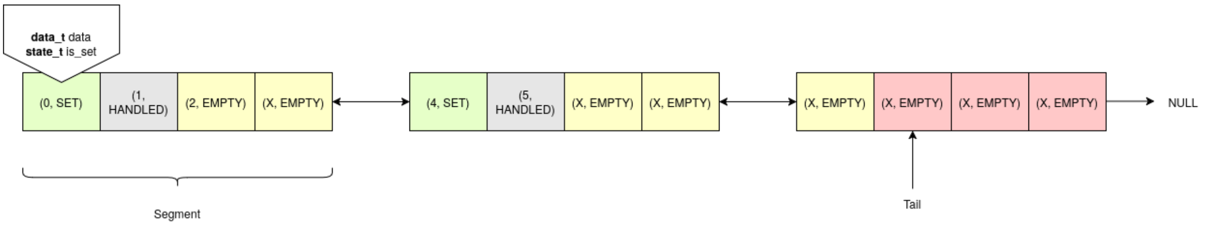


Figure 18: Jiffy's structure.

Like DQueue, Jiffy represents the queue as a doubly-linked list of segments as in Figure 18. This design again allows Jiffy to be unbounded while using head and tail indices to index elements. Each segment contains a pointer to a dynamically allocated array of slots, instead of directly storing the array. Each slot in the segment contains the data item and a state of that slot (`state_t` in the figure). There are 3 states: `SET`, `EMPTY` and `HANDLED`. Initially, all slots are `EMPTY`. Instead of keeping a global head index, there are per-segment Head indices pointing to the first non-`HANDLED` slot. However, there is still one global Tail index shared by all the processes.

To enqueue, each enqueuer would FAA the Tail to reserve a slot. If the slot isn't in the linked list yet, it tries to allocate new segments and CAS them at the end of the linked list until the slot is available. It then traverses to the desired segment by following the previous pointers starting from the last segment. It then writes the data and sets the slot's state to `SET`. Notice that `EMPTY` slots actually have two substates. If an `EMPTY` slot is before the Tail index, that slot is actually reserved by an enqueuer but has not been set yet, while the `EMPTY` slots after the Tail index are truly empty.

To dequeue, the dequeuer would start from the Head index of the first segment, scanning until it finds the first non-`HANDLED` slot before the end of the queue. If there's no such slot, the queue is empty and the dequeuer would return nothing. If this slot is `SET`, it simply reads the data item in this slot and sets it to `HANDLED`. If this slot is `EMPTY`, that means this slot has been reserved by an enqueuer that hasn't finished. In this case, the dequeuer performs a scan forward to find the first `SET` slot. If not found, the dequeuer returns nothing. Otherwise, it continues to repeatedly scan all slots between the first non-`HANDLED` and the last found `SET` slot until the first `SET` slot between in this interval is unchanged between 2 scans. Only then, the dequeuer would return the data item in this `SET` slot and mark it as `HANDLED`.

Similar to DQueue, CAS is only used when appending new segments at the end of the queue. Therefore, ABA problem only involves internal manipulation of pointers to dynamically-allocated memory. If a proper memory reclamation scheme is utilized.

Regarding memory reclamation, while the dequeuer is scanning the queue, it will reclaim any segments with only `HANDLED` slots. We can see there's potentially a pitfall similar to the one DQueue runs into here. To avoid this pitfall, Jiffy takes the following measures:

- When scanning the queue and the dequeuer sees that a segment contains only `HANDLED` slots, it only reclaims the dynamically-allocated array in the segment, which consumes the most memory, while still keeping the linked-list structure intact. Therefore, if any enqueuer is holding a reference to a segment before the partially-reclaimed segment, it can still traverse the next pointer chain safely.
- To fully reclaim a segment, when partially reclaim a segment, it is added to a garbage list. Note that the first segments that contain only `HANDLED` slots can be fully reclaimed right when the dequeuer performs the scan. When a segment is fully reclaimed, any segment in the garbage list that precedes this segment is also fully reclaimed.

3.1.5 Remarks

Out of the 4 investigated MPSC queue algorithms, we quickly eliminate DQueue and WRLQueue as a potential candidate for porting to distributed environment because they either do not provide a sufficient progress guarantee or protection against ABA problem and memory reclamation problem. Jiffy's idea of the dequeuer rescanning the global queue looking for a `SET` slot is quite useful and partly contributes to our idea of double scanning in Slotqueue (Section 4.4), which is our improvement over indefinite repeated scans as in Jiffy. For the time being, due to time constraints, LTQueue remains our primary inspiration and Jiffy will be adapted for distributed environments in the future.

3.2 Distributed MPSC queues

This section summarizes to the best of our knowledge existing MPSC queue algorithms, which is reflected in Section 3.2.

The only paper we have found so far that either mentions directly or indirectly the design of an MPSC queue is [1]. [1] introduces a hosted blocking (the original paper claims that it's lock-free) bounded distributed MPSC queue called active-message queue (AMQueue) that bares resemblance to WRLQueue in [15].

FIFO queues	**Active-message queue (AMQueue) [1]**
Progress guarantee of dequeue	Blocking (*)
Progress guarantee of enqueue	Wait-free
ABA solution	No compare-and-swap usage
Safe memory reclamation	Custom scheme
Number of elements	Bounded

Table 3: Characteristic summary of existing distributed MPSC queues.
R stands for remote operations and L stands for local operations.
(*) [1] claims that it's lock-free.

The structure of AMQueue is given in Figure 19. The MPSC is split into 2 queues, each maintains its own set of control variables:

- `WriterCnt`: The number of enqueuer currently writing in this queue.
- `Offset`: The index to the first empty entry in the queue.

Note that any shared data and control variables are hosted on the dequeuer.

To determine which queue to read and write, the `QueueNum` binary variable is used. If `QueueNum` is `0`, then the first queue is being actively written by enqueuers and the second queue is being reserved for the dequeuer, and otherwise.

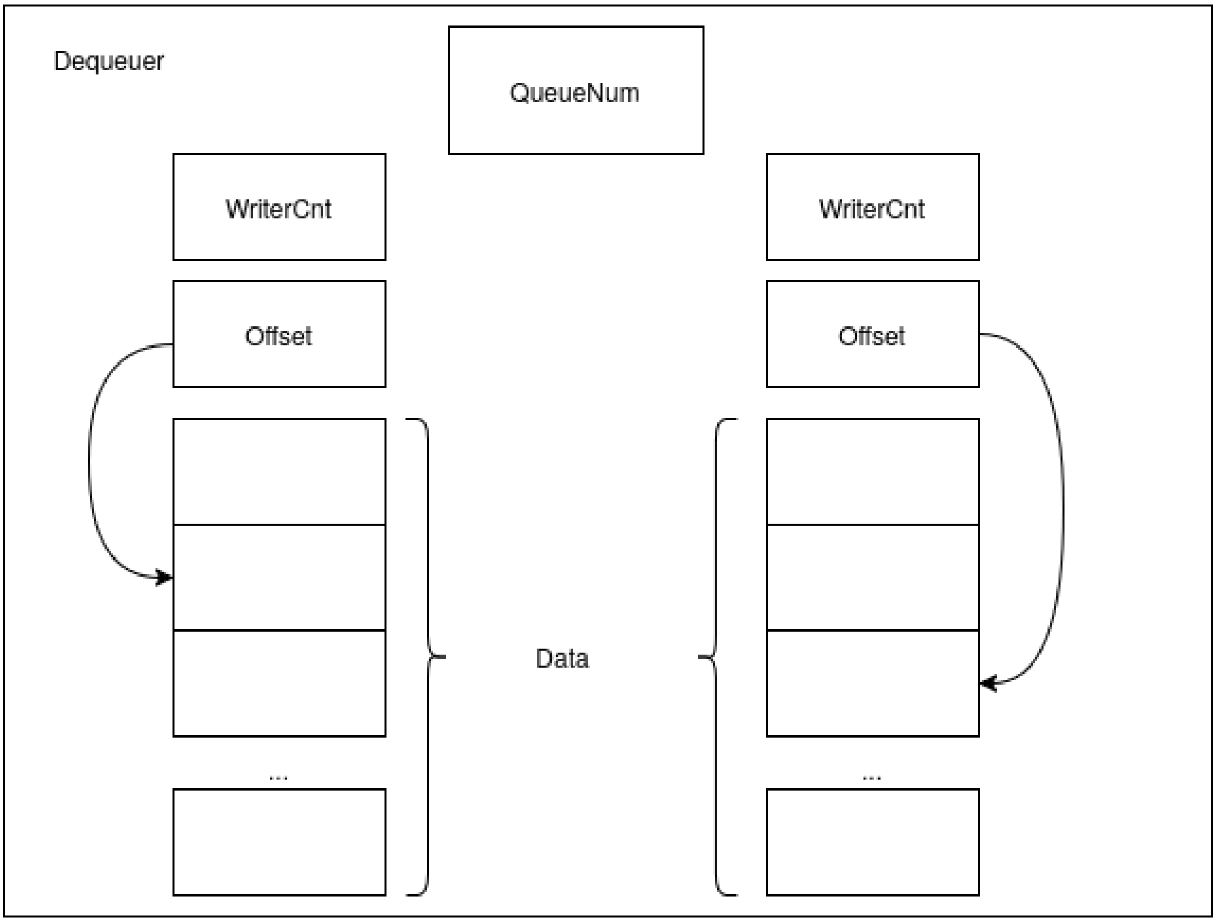


Figure 19: AMQueue's structure.

To enqueue, the enqueuer first reads the `QueueNum` variable to see which of the queue is active. The enqueuer then registers for that queue by atomically FAA-ing the corresponding `WriteCnt` variable. If the fetched value is negative though, the `QueueNum` queue is being swapped for dequeueing and the enqueuer has to decrement the `WriteCnt` variable and repeat the process until `WriteCnt` is positive. After a successful registration, the enqueuer then reserves an entry in the data array by FAA-ing the `Offset` variable. After that, the enqueuer can enqueue data at its leisure. Upon success, the enqueuer has to decrement `WriteCnt` before returning.

To dequeue, the dequeuer inverts `QueueNum` to direct future enqueuers to the other queue. The dequeuer then subtracts a sufficiently large number from `WriterCnt` to signal to other enqueuers that it has started processing. The dequeuer has to wait for all current enqueuers in the queue to finish by repeatedly checking the `WriterCnt` variable, hence the blocking property. After all enqueuers have finished, the dequeuer then batch-dequeues all data in the queue, resets the `Offset` and `WriterCnt` variables to `0`.

AMQueue will serve as a benchmarking baseline for our MPSC queues in Chapter V.

Chapter IV Distributed MPSC queues

Based on the MPSC queue algorithms we have surveyed in Chapter III, we propose two wait-free distributed MPSC queue algorithms:

- dLTQueue (Section 4.3) is a direct modification of the original LTQueue [8] without any usage of LL/SC, adapted for distributed environment.
- Slotqueue (Section 4.4) is inspired by the timestamp-refreshing idea of LTQueue [8] and repeated-rescan of Jiffy [9]. Although it still bears some resemblance to LTQueue, we believe it to be more optimized for distributed context.

In actuality, dLTQueue and Slotqueue are more than simple MPSC algorithms. They are “MPSC queue wrappers”, that is, given an SPSC queue implementation, they yield an MPSC implementation. There’s one additional constraint: The SPSC interface must support an additional `readFront` operation, which returns the first data item currently in the SPSC queue.

This fact has an important implication: when we’re talking about the characteristics (correctness, progress guarantee, performance model, ABA solution and safe memory reclamation scheme) of an MPSC queue wrapper, we’re talking about the correctness, progress guarantee, performance model, ABA solution and safe memory reclamation scheme of the wrapper that turns an SPSC queue to an MPSC queue:

- If the underlying SPSC queue is linearizable (which is composable), the resulting MPSC queue is linearizable.
- The resulting MPSC queue’s progress guarantee is the weaker guarantee between the wrapper’s and the underlying SPSC’s.
- If the underlying SPSC queue is safe against ABA problem and memory reclamation, the resulting MPSC queue is also safe against these problems.
- If the underlying SPSC queue is unbounded, the resulting MPSC queue is also unbounded.
- The theoretical performance of dLTQueue and Slotqueue has to be coupled with the theoretical performance of the underlying SPSC.

The characteristics of these MPSC queue wrappers are summarized in Table 4. For benchmarking purposes, we use a baseline distributed SPSC introduced in Section 4.2 in combination with the MPSC queue wrappers. The characteristics of the resulting MPSC queues are also shown in Table 4.

MPSC queues	dLTQueue	Slotqueue
Correctness	Linearizable	Linearizable
Progress guarantee of dequeue	Wait-free	Wait-free
Progress guarantee of enqueue	Wait-free	Wait-free
Dequeue time-complexity (*)	$4\log_2(n)R + 6\log_2(n)L$	$3R + 2nL$
Enqueue time-complexity (*)	$6\log_2(n)R + 4\log_2(n)L$	$4R + 3L$
ABA solution	Unique timestamp	No hazardous ABA problem
Safe memory reclamation	No dynamic memory allocation	No dynamic memory allocation
Number of element	Depending on the underlying SPSC	Depending on the underlying SPSC

Table 4: Characteristic summary of our proposed distributed MPSC queues.
(1) n is the number of enqueuers.
(2) R stands for **remote operation** and L stands for **local operation**.
(*) The underlying SPSC is assumed to be our simple distributed SPSC in Section 4.2.

In the following sections, we present first the one-sided-communication primitives that we assume will be available in our distributed algorithm specification and then our proposed distributed MPSC queue wrappers in detail. Any other discussions about theoretical aspects of these algorithms such as linearizability, progress guarantee, performance model are deferred to Appendix A.

In our description, we assume that each process in our program is assigned a unique number as an identifier, which is termed as its **rank**. The numbers are taken from the range of `[0, size - 1]`, with `size` being the number of processes in our program.

4.1 Distributed one-sided-communication primitives in our distributed algorithm specification

Although we use MPI-3 RMA to implement these algorithms, the algorithm specifications themselves are not inherently tied to the MPI-3 RMA interface. For clarity and convenience in specification, we define the following distributed primitives used in our pseudocode.

`remote<T>`

> A distributed shared variable of type T. The process that physically stores the variable in its local memory is referred to as the **host**. This represents data that can be accessed or modified remotely by other processes.

```
void aread_sync(remote<T> src, T* dest)
```

> Issue a synchronous read of the distributed variable `src` and stores its value into the local memory location pointed to by `dest`. The read is guaranteed to be completed when the function returns.

```
void aread_sync(remote<T*> src, int index, T* dest)
```

> Issue a synchronous read of the element at position `index` within the distributed array `src` (where `src` is a pointer to a remotely hosted array of type `T`) and stores the value into the local memory location pointed to by `dest`. The read is guaranteed to be completed when the function returns.

```
void awrite_sync(remote<T> dest, T* src)
```

> Issue a synchronous write of the value at the local memory location pointed to by `src` into the distributed variable `dest`. The write is guaranteed to be completed when the function returns.

```
void awrite_sync(remote<T*> dest, int index, T* src)
```

> Issue a synchronous write of the value at the local memory location pointed to by `src` into the element at position `index` within the distributed array `dest` (where `dest` is a pointer to a remotely hosted array of type `T`). The write is guaranteed to be completed when the function returns.

```
void aread_async(remote<T> src, T* dest)
```

> Issue an asynchronous read of the distributed variable `src` and initiate the transfer of its value into the local memory location pointed to by `dest`. The operation may not be completed when the function returns.

```
void aread_async(remote<T*> src, int index, T* dest)
```

> Issue an asynchronous read of the element at position `index` within the distributed array `src` (where `src` is a pointer to a remotely hosted array of type `T`) and initiate the transfer of its value into the local memory location pointed to by `dest`. The operation may not be completed when the function returns.

```
void awrite_async(remote<T> dest, T* src)
```

> Issue an asynchronous write of the value at the local memory location pointed to by `src` into the distributed variable `dest`. The operation may not be completed when the function returns.

`void awrite_async(remote<T*> dest, int index, T* src)`

> Issue an asynchronous write of the value at the local memory location pointed to by `src` into the element at position `index` within the distributed array `dest` (where `dest` is a pointer to a remotely hosted array of type `T`). The operation may not be completed when the function returns.

`void flush(remote<T> src)`

> Ensure that all read and write operations on the distributed variable `src` (or its associated array) issued before this function call are fully completed by the time the function returns.

`bool compare_and_swap_sync(remote<T> dest, T old_value, T new_value)`

> Issue a synchronous compare-and-swap operation on the distributed variable `dest`. The operation atomically compares the current value of `dest` with `old_value`. If they are equal, the value of `dest` is replaced with `new_value`; otherwise, no change is made. The operation is guaranteed to be completed when the function returns, ensuring that the update (if any) is visible to all processes. The type `T` must be a data type with a size of `1`, `2`, `4`, or `8` bytes.

`bool compare_and_swap_sync(remote<T*> dest, int index, T old_value, T new_value)`

> Issue a synchronous compare-and-swap operation on the element at position `index` within the distributed array `dest` (where `dest` is a pointer to a remotely hosted array of type `T`). The operation atomically compares the current value of the element at `dest[index]` with `old_value`. If they are equal, the element at `dest[index]` is replaced with new_value; otherwise, no change is made. The operation is guaranteed to be completed when the function returns, ensuring that the update (if any) is visible to all processes. The type `T` must be a data type with a size of `1`, `2`, `4`, or `8`.

`T fetch_and_add_sync(remote<T> dest, T inc)`

> Issue a synchronous fetch-and-add operation on the distributed variable `dest`. The operation atomically adds the value `inc` to the current value of `dest`, returning the original value of `dest` (before the addition) to the calling process. The update to `dest` is guaranteed to be completed and visible to all processes when the function returns. The type `T` must be an integral type with a size of `1`, `2`, `4`, or `8` bytes.

4.2 A simple baseline distributed SPSC

For prototyping, the two MPSC queue wrapper algorithms we propose here both utilize a baseline distributed SPSC data structure, which we will present first. For implementation simplicity, we present a bounded SPSC, effectively make our proposed algorithms support only a bounded number of elements. However, one can trivially substitute another distributed unbounded SPSC to make our proposed algorithms support an unbounded number of elements, as long as this SPSC supports the same interface as ours.

Placement-wise, all queue data in this SPSC is hosted on the enqueuer while the control variables i.e. `First` and `Last`, are hosted on the dequeuer.

Types

`data_t` = The type of data stored.

Shared variables

`First`: `remote<uint64_t>`

The index of the last undequeued entry.

Hosted at the dequeuer.

`Last`: `remote<uint64_t>`

The index of the last unenqueued entry.

Hosted at the dequeuer.

`Data`: `remote<data_t*>`

An array of `data_t` of some known capacity.

Hosted at the enqueuer.

Enqueuer-local variables

`Capacity`: A read-only value indicating the capacity of the SPSC.

`First_buf`: The cached value of `First`.

`Last_buf`: The cached value of `Last`.

Dequeuer-local variables

`Capacity`: A read-only value indicating the capacity of the SPSC.

`First_buf`: The cached value of `First`.

`Last_buf`: The cached value of `Last`.

Enqueuer initialization

Initialize `First` and `Last` to `0`.

Initialize `Capacity`.

Allocate array in `Data`.

Initialize `First_buf = Last_buf = 0`.

Dequeuer initialization

Initialize `Capacity`.

Initialize `First_buf = Last_buf = 0`.

The procedures of the enqueuer are given as follows.

Procedure 4: `bool spsc_enqueue(data_t v)`

```
1 new_last = Last_buf + 1
2 if (new_last - First_buf > Capacity)
3   aread_sync(First, &First_buf)
4   if (new_last - First_buf > Capacity)
5     return false
6 awrite_sync(Data, Last_buf % Capacity, &v)
7 awrite_sync(Last, &new_last)
8 Last_buf = new_last
9 return true
```

spsc_enqueue first computes the new Last value (Line 1). If the queue is full as indicating by the difference the new Last value and First_buf (Line 2), there can still be the possibility that some elements have been dequeued but First_buf hasn't been synced with First yet, therefore, we first refresh the value of First_buf by fetching from First (Line 3). If the queue is still full (Line 4), we signal failure (Line 5). Otherwise, we proceed to write the enqueued value to the entry at Last_buf % Capacity (Line 6), increment Last (Line 7), update the value of Last_buf (Line 8) and signal success (Line 9).

Procedure 5: `bool spsc_readFront`$_e$`(data_t* output)`

```
10 if (First_buf >= Last_buf)
11   return false
12 aread_sync(First, &First_buf)
13 if (First_buf >= Last_buf)
14   return false
15 aread_sync(Data, First_buf % Capacity, output)
16 return true
```

spsc_readFront$_e$ first checks if the SPSC is empty based on the difference between First_buf and Last_buf (Line 10). Note that if this check fails, we signal failure immediately (Line 11) without refetching either First or Last. This suffices because Last cannot be out-of-sync with Last_buf as we're the enqueuer and First can only increase since the last refresh of First_buf, therefore, if we refresh First and Last, the condition on Line 10 would return false anyways. If the SPSC is not empty, we refresh First and re-perform the empty check (Line 13 - Line 14). If the SPSC is again not empty, we read the queue entry at First_buf % Capacity into output (Line 15) and signal success (Line 16).

The procedures of the dequeuer are given as follows.

Procedure 6: `bool spsc_dequeue(data_t* output)`

```
17 new_first = First_buf + 1
18 if (new_first > Last_buf)
19     aread_sync(Last, &Last_buf)
20     if (new_first > Last_buf)
21         return false
22 aread_sync(Data, First_buf % Capacity, output)
23 awrite_sync(First, &new_first)
24 First_buf = new_first
25 return true
```

`spsc_dequeue` first computes the new `First` value (Line 17). If the queue is empty as indicating by the difference the new `First` value and `Last_buf` (Line 18), there can still be the possibility that some elements have been enqueued but `Last_buf` hasn't been synced with `Last` yet, therefore, we first refresh the value of `Last_buf` by fetching from `Last` (Line 19). If the queue is still empty (Line 20), we signal failure (Line 21). Otherwise, we proceed to read the top value at `First_buf % Capacity` (Line 22) into `output`, increment `First` (Line 23) - effectively dequeue the element, update the value of `First_buf` (Line 24) and signal success (Line 25).

Procedure 7: `bool spsc_readFront`$_d$`(data_t* output)`

```
26 if (First_buf >= Last_buf)
27     aread_sync(Last, &Last_buf)
28     if (First_buf >= Last_buf)
29         return false
30 aread_sync(Data, First_buf % Capacity, output)
31 return true
```

`spsc_readFront`$_d$ first checks if the SPSC is empty based on the difference between `First_buf` and `Last_buf` (Line 26). If this check fails, we refresh `Last_buf` (Line 27) and recheck (Line 28). If the recheck fails, signal failure (Line 29). If the SPSC is not empty, we read the queue entry at `First_buf % Capacity` into `output` (Line 30) and signal success (Line 31).

4.3 dLTQueue - Straightforward LTQueue adapted for distributed environment

This algorithm presents our most straightforward effort to port LTQueue [8] to distributed context. The main challenge is that LTQueue uses LL/SC as the universal atomic instruction and also an ABA solution, but LL/SC is not available in distributed

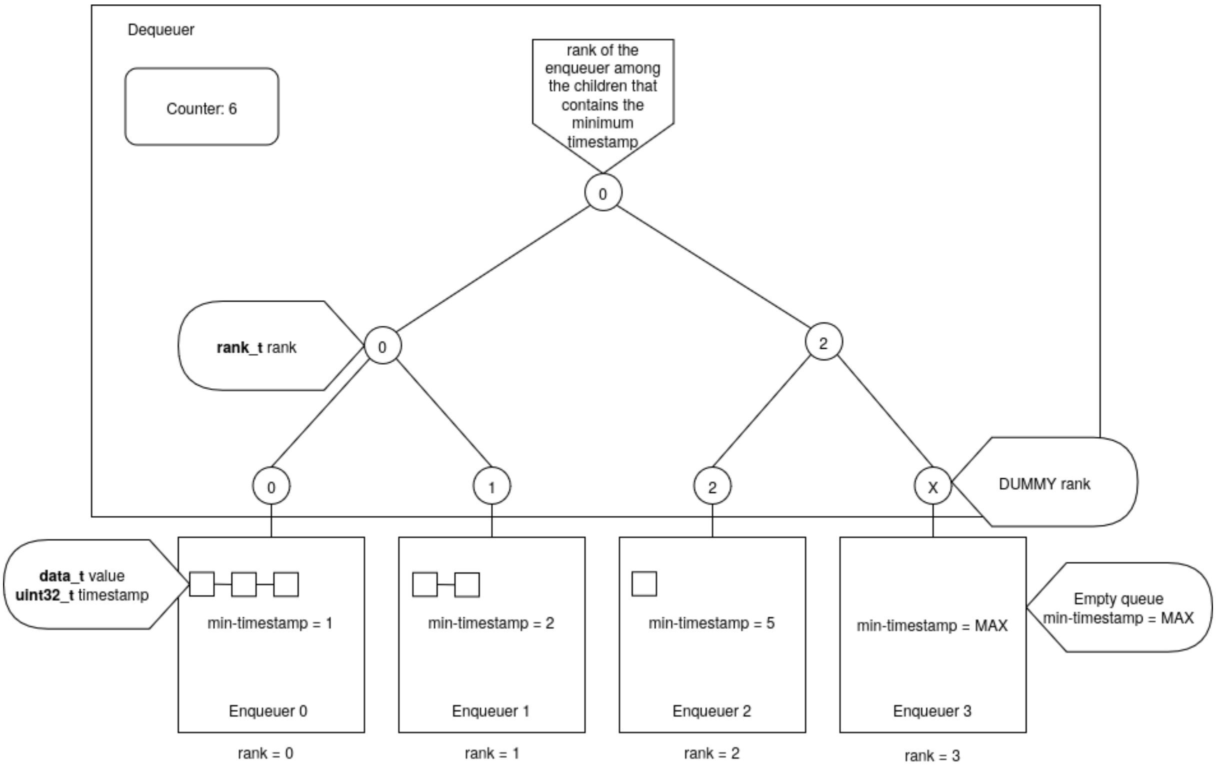


Figure 20: dLTQueue's structure.

programming environments. We have to replace any usage of LL/SC in the original LTQueue algorithm. We use compare-and-swap and the well-known monotonic timestamp scheme to guard against ABA problem.

4.3.1 Overview

The structure of our dLTQueue is shown as in Figure 20.

We differentiate between 2 types of nodes: **enqueuer nodes** (represented as the rectangular boxes at the bottom of Figure 20) and normal **tree nodes** (represented as the circular boxes in Figure 20).

Each enqueuer node corresponds to an enqueuer. Each time the local SPSC is enqueued with a value, the enqueuer timestamps the value using a distributed counter shared by all enqueuers. An enqueuer node stores the SPSC local to the corresponding enqueuer and a `min_timestamp` value which is the minimum timestamp inside the local SPSC.

Each tree node stores the rank of an enqueuer process. This rank corresponds to the enqueuer node with the minimum timestamp among the node's children's ranks. The tree node that's attached to an enqueuer node is called a **leaf node**, otherwise, it's called an **internal node**.

Note that if a local SPSC is empty, the `min_timestamp` variable of the corresponding enqueuer node is set to `MAX_TIMESTAMP` and the corresponding leaf node's rank is set to `DUMMY_RANK`.

Placement-wise:

- The **enqueuer nodes** are hosted at the corresponding **enqueuer**.

- All the **tree nodes** are hosted at the **dequeuer**.
- The distributed counter, which the enqueuers use to timestamp their enqueued value, is hosted at the **dequeuer**.

4.3.2 Data structure

Below is the types utilized in dLTQueue.

Types

`data_t` = The type of the data to be stored.

`spsc_t` = The type of the SPSC, this is assumed to be the distributed SPSC in Section 4.2.

`rank_t` = The type of the rank of an enqueuer process tagged with a unique timestamp (version) to avoid ABA problem.

struct
`value`: `uint32_t`
`version`: `uint32_t`
end

`timestamp_t` = The type of the timestamp tagged with a unique timestamp (version) to avoid ABA problem.

struct
`value`: `uint32_t`
`version`: `uint32_t`
end

`node_t` = The type of a tree node.

struct
`rank`: `rank_t`
end

The shared variables in our LTQueue version are as follows.

Note that we have described a very specific and simple way to organize the tree nodes in dLTQueue in a min-heap-like array structure hosted on the sole dequeuer. We will resume our description of the related tree-structure procedures `parent()` (Procedure 9), `children()` (Procedure 10), `leafNodeIndex()` (Procedure 11) with this representation in mind. However, our algorithm doesn't strictly require this representation and can be substituted with other more-optimized representations & distributed placements, as long as the similar tree-structure procedures are supported.

Shared variables

`Counter`: `remote<uint64_t>`
A distributed counter shared by the enqueuers. Hosted at the dequeuer.

`Tree_size`: `uint64_t`

A read-only variable storing the number of tree nodes present in the dLTQueue.

`Nodes`: `remote<node_t>`

An array with `Tree_size` entries storing all the tree nodes present in the dLTQueue shared by all processes.

Hosted at the dequeuer.

This array is organized in a similar manner as a min-heap: At index `0` is the root node. For every index $i > 0$, $\left\lfloor \frac{i-1}{2} \right\rfloor$ is the index of the parent of node i. For every index $i > 0$, $2i + 1$ and $2i + 2$ are the indices of the children of node i.

`Dequeuer_rank`: `uint32_t`

The rank of the dequeuer process. This is read-only.

`Timestamps`: A read-only **array** `[0..size - 2]` of `remote<timestamp_t>`, with `size` being the number of processes.

The entry at index i corresponds to the `Min_timestamp` distributed variable at the enqueuer with an order of i.

Similar to the fact that each process in our program is assigned a rank, each enqueuer process in our program is assigned an **order**. The following procedure computes an enqueuer's order based on its rank:

Procedure 8: `uint32_t enqueuerOrder(uint32_t enqueuer_rank)`

```
return enqueuer_rank > Dequeuer_rank ? enqueuer_rank - 1 : enqueuer_rank
```

This procedure is rather straightforward: Each enqueuer is assigned an order in the range `[0, size - 2]`, with `size` being the number of processes and the total ordering among the enqueuers based on their ranks is the same as the total ordering among the enqueuers based on their orders.

Enqueuer-local variables

`Enqueuer_count`: `uint64_t`

The number of enqueuers.

`Self_rank`: `uint32_t`

The rank of the current enqueuer process.

`Min_timestamp`: `remote<timestamp_t>`

`Spsc`: `spsc_t`

This SPSC is synchronized with the dequeuer.

Dequeuer-local variables

`Enqueuer_count`: `uint64_t`

The number of enqueuers.

`Spscs`: **array** of `spsc_t` with `Enqueuer_count` entries.

The entry at index i corresponds to the `Spsc` at the enqueuer with an order of i.

Initially, the enqueuers and the dequeuer are initialized as follows:

Enqueuer initialization

Initialize `Enqueuer_count`, `Self_rank` and `Dequeuer_rank`.

Initialize `Spsc` to the initial state.

Initialize `Min_timestamp` to `timestamp_t {MAX_TIMESTAMP, 0}`.

Dequeuer initialization

Initialize `Enqueuer_count`, `Self_rank` and `Dequeuer_rank`.

Initialize `Counter` to `0`.

Initialize `Tree_size` to `Enqueuer_count * 2`.

Initialize `Nodes` to an array with `Tree_size` entries. Each entry is initialized to `node_t {DUMMY_RANK}`.

Initialize `Spscs`, synchronizing each entry with the corresponding enqueuer.

Initialize `Timestamps`, synchronizing each entry with the corresponding enqueuer.

4.3.3 Algorithm

We first present the tree-structure utility procedures that are shared by both the enqueuer and the dequeuer:

Procedure 9: `uint32_t parent(uint32_t index)`

```
return (index - 1) / 2
```

`parent` returns the index of the parent tree node given the node with index `index`. These indices are based on the shared `Nodes` array. Based on how we organize the `Nodes` array, the index of the parent tree node of `index` is `(index - 1) / 2`.

Procedure 10: `vector<uint32_t> children(uint32_t index)`

```
3  left_child = index * 2 + 1
4  right_child = left_child + 1
5  res = vector<uint32_t>()
6  if (left_child >= Tree_size)
7  | return res
8  res.push(left_child)
9  if (right_child >= Tree_size)
10 | return res
11 res.push(right_child)
12 return res
```

Similarly, `children` returns all indices of the child tree nodes given the node with index `index`. These indices are based on the shared `Nodes` array. Based on how we organize the `Nodes` array, these indices can be either `index * 2 + 1` or `index * 2 + 2`.

Procedure 11: `uint32_t leafNodeIndex(uint32_t enqueuer_rank)`

```
13 return Tree_size + enqueuerOrder(enqueuer_rank)
```

`leafNodeIndex` returns the index of the leaf node that's logically attached to the enqueuer node with rank `enqueuer_rank` as in Figure 20.

The followings are the enqueuer procedures.

Procedure 12: `bool enqueue(data_t value)`

```
14 timestamp = fetch_and_add_sync(Counter, 1)
15 if (!spsc_enqueue(&Spsc, (value, timestamp)))
16 | return false
17 propagate_e()
18 return true
```

To enqueue a value, `enqueue` first obtains a count by `FAA` the distributed counter `Counter` (Line 14). Then, we enqueue the data tagged with the timestamp into the local SPSC (Line 15). Then, `enqueue` propagates the changes by invoking `propagate_e()` (Line 17) and returns `true`.

Procedure 13: `void propagate_e()`

```
19 if (!refreshTimestamp_e())
20     refreshTimestamp_e()
21 if (!refreshLeaf_e())
22     refreshLeaf_e()
23 current_node_index = leafNodeIndex(Self_rank)
24 repeat
25     current_node_index = parent(current_node_index)
26     if (!refresh_e(current_node_index))
27         refresh_e(current_node_index)
28 until current_node_index == 0
```

The `propagate_e` procedure is responsible for propagating SPSC updates up to the root node as a way to notify other processes of the newly enqueued item. It is split into 3 phases: Refreshing of `Min_timestamp` in the enqueuer node (Line 19 - Line 20), refreshing of the enqueuer's leaf node (Line 21 - Line 22), refreshing of internal nodes (Line 24 - Line 28). On Line 21 - Line 28, we refresh every tree node that lies between the enqueuer node and the root node.

Procedure 14: `bool refreshTimestamp_e()`

```
29 min_timestamp = timestamp_t {}
30 aread_sync(Min_timestamp, &min_timestamp)
31 {old-timestamp, old-version} = min_timestamp
32 front = (data_t {}, timestamp_t {})
33 is_empty = !spsc_readFront(Spsc, &front)
34 if (is_empty)
35     return compare_and_swap_sync(Min_timestamp,
       timestamp_t {old-timestamp, old-version},
       timestamp_t {MAX_TIMESTAMP, old-version + 1})
36 else
37     return compare_and_swap_sync(Min_timestamp,
       timestamp_t {old-timestamp, old-version},
       timestamp_t {front.timestamp, old-version + 1})
```

The `refreshTimestamp_e` procedure is responsible for updating the `Min_timestamp` of the enqueuer node. It simply looks at the front of the local SPSC (Line 33) and CAS `Min_timestamp` accordingly (Line 34 - Line 37).

Procedure 15: `bool refreshNode`$_e$`(uint32_t current_node_index)`

```
38 current_node = node_t {}
39 aread_sync(Nodes, current_node_index, &current_node)
40 {old-rank, old-version} = current_node.rank
41 min_rank = DUMMY_RANK
42 min_timestamp = MAX_TIMESTAMP
43 for child_node_index in children(current_node)
44     child_node = node_t {}
45     aread_sync(Nodes, child_node_index, &child_node)
46     {child_rank, child_version} = child_node
47     if (child_rank == DUMMY_RANK) continue
48     child_timestamp = timestamp_t {}
49     aread_sync(Timestamps[enqueuerOrder(child_rank)], &child_timestamp)
50     if (child_timestamp < min_timestamp)
51         min_timestamp = child_timestamp
52         min_rank = child_rank
   return compare_and_swap_sync(Nodes, current_node_index,
53 node_t {rank_t {old_rank, old_version}},
   node_t {rank_t {min_rank, old_version + 1}})
```

The `refreshNode`$_e$ procedure is responsible for updating the ranks of the internal nodes affected by the enqueue. It loops over the children of the current internal nodes (Line 43). For each child node, we read the rank stored in it (Line 46), if the rank is not `DUMMY_RANK`, we proceed to read the value of `Min_timestamp` of the enqueuer node with the corresponding rank (Line 49). At the end of the loop, we obtain the rank stored inside one of the child nodes that has the minimum timestamp stored in its enqueuer node (Line 51 - Line 52). We then try to CAS the rank inside the current internal node to this rank (Line 53).

Procedure 16: `bool refreshLeaf`$_e$`()`

```
54 leaf_node_index = leafNodeIndex(Self_rank)
55 leaf_node = node_t {}
56 aread_sync(Nodes, leaf_node_index, &leaf_node)
57 {old_rank, old_version} = leaf_node.rank
58 min_timestamp = timestamp_t {}
59 aread_sync(Min_timestamp, &min_timestamp)
60 timestamp = min_timestamp.timestamp
   return compare_and_swap_sync(Nodes, leaf_node_index,
61 node_t {rank_t {old-rank, old-version}},
   node_t {timestamp == MAX ? DUMMY_RANK : Self_rank, old_version + 1})
```

The `refreshLeaf`$_e$ procedure is responsible for updating the rank of the leaf node affected by the enqueue. It simply reads the value of `Min_timestamp` of the enqueuer node it's logically attached to (Line 59) and CAS the leaf node's rank accordingly (Line 61).

The followings are the dequeuer procedures.

Procedure 17: `bool dequeue(data_t* output)`

```
62 root_node = node_t {}
63 aread_sync(Nodes, 0, &root_node)
64 {rank, version} = root_node.rank
65 if (rank == DUMMY_RANK) return false
66 output_with_timestamp = (data_t {}, timestamp_t {})
67 if (!spsc_dequeue(&Spscs[enqueuerOrder(rank)]),
       &output_with_timestamp))
68   return false
69 *output = output_with_timestamp.data
70 propagate_d(rank)
71 return true
```

To dequeue a value, `dequeue` reads the rank stored inside the root node (Line 64). If the rank is `DUMMY_RANK`, the MPSC queue is treated as empty and failure is signaled (Line 65). Otherwise, we invoke `spsc_dequeue` on the SPSC of the enqueuer with the obtained rank (Line 67). We then extract out the real data and set it to `output` (Line 69). We finally propagate the dequeue from the enqueuer node that corresponds to the obtained rank (Line 70) and signal success (Line 71).

Procedure 18: `void propagate`$_d$`(uint32_t enqueuer_rank)`

```
if (!refreshTimestamp_d(enqueuer_rank))
  refreshTimestamp_d(enqueuer_rank)
if (!refreshLeaf_d(enqueuer_rank))
  refreshLeaf_d(enqueuer_rank)
current_node_index = leafNodeIndex(enqueuer_rank)
repeat
  current_node_index = parent(current_node_index)
  if (!refresh_d(current_node_index))
    refresh_d(current_node_index)
until current_node_index == 0
```

The `propagate`$_d$ procedure is similar to `propagate`$_e$, with appropriate changes to accommodate the dequeuer.

Procedure 19: `bool refreshTimestamp`$_d$`(uint32_t enqueuer_rank)`

```
enqueuer_order = enqueuerOrder(enqueuer_rank)
min_timestamp = timestamp_t {}
aread_sync(Timestamps, enqueuer_order, &min_timestamp)
{old-timestamp, old-version} = min_timestamp
front = (data_t {}, timestamp_t {})
is_empty = !spsc_readFront(&Spscs[enqueuer_order], &front)
if (is_empty)
    return compare_and_swap_sync(Timestamps, enqueuer_order,
    timestamp_t {old-timestamp, old-version},
    timestamp_t {MAX_TIMESTAMP, old-version + 1})
else
    return compare_and_swap_sync(Timestamps, enqueuer_order,
    timestamp_t {old-timestamp, old-version},
    timestamp_t {front.timestamp, old-version + 1})
```

The `refreshTimestamp`$_d$ procedure is similar to `refreshTimestamp`$_e$, with appropriate changes to accommodate the dequeuer.

Procedure 20: `bool refreshNode`$_d$`(uint32_t current_node_index)`

```
current_node = node_t {}
aread_sync(Nodes, current_node_index, &current_node)
{old-rank, old-version} = current_node.rank
min_rank = DUMMY_RANK
min_timestamp = MAX_TIMESTAMP
for child_node_index in children(current_node)
    child_node = node_t {}
    aread_sync(Nodes, child_node_index, &child_node)
    {child_rank, child_version} = child_node
    if (child_rank == DUMMY_RANK) continue
    child_timestamp = timestamp_t {}
    aread_sync(Timestamps[enqueuerOrder(child_rank)], &child_timestamp)
    if (child_timestamp < min_timestamp)
        min_timestamp = child_timestamp
        min_rank = child_rank
return compare_and_swap_sync(Nodes, current_node_index,
node_t {rank_t {old_rank, old_version}},
node_t {rank_t {min_rank, old_version + 1}})
```

The `refreshNode`$_d$ procedure is similar to `refreshNode`$_e$, with appropriate changes to accommodate the dequeuer.

Procedure 21: `bool refreshLeaf`$_d$`(uint32_t enqueuer_rank)`

```
leaf_node_index = leafNodeIndex(enqueuer_rank)
leaf_node = node_t {}
aread_sync(Nodes, leaf_node_index, &leaf_node)
{old_rank, old_version} = leaf_node.rank
min_timestamp = timestamp_t {}
aread_sync(Timestamps, enqueuerOrder(enqueuer_rank), &min_timestamp)
timestamp = min_timestamp.timestamp
return compare_and_swap_sync(Nodes, leaf_node_index,
node_t {rank_t {old-rank, old-version}},
node_t {timestamp == MAX ? DUMMY_RANK : Self_rank, old_version + 1})
```

The `refreshLeaf`$_d$ procedure is similar to `refreshLeaf`$_e$, with appropriate changes to accommodate the dequeuer.

4.4 Slotqueue - dLTQueue-inspired distributed MPSC queue with all constant-time operations

The straightforward dLTQueue algorithm we have ported in Section 4.3 pretty much preserves the original algorithm's characteristics, i.e. wait-freedom and time complexity of $\Theta(\log n)$ for `dequeue` and `enqueue` operations (which we will prove in Appendix A). We note that in shared-memory systems, this logarithmic growth is fine. However, in distributed systems, this increase in remote operations would present a bottleneck in enqueue and dequeue latency. Upon closer operation, this logarithmic growth is due to the propagation process because it has to traverse every level in the tree. Intuitively, this is the problem of we trying to maintain the tree structure. Therefore, to be more suitable for distributed context, we propose a new algorithm Slotqueue inspired by LTQueue, which uses a slightly different structure. The key point is that both `enqueue` and `dequeue` only perform a constant number of remote operations, at the cost of `dequeue` having to perform $\Theta(n)$ local operations, where n is the number of enqueuers. Because remote operations are much more expensive, this might be a worthy tradeoff.

4.4.1 Overview

The structure of Slotqueue is shown as in Figure 21.

Each enqueuer hosts a distributed SPSC as in dLTQueue (Section 4.3). The enqueuer when enqueues a value to its local SPSC will timestamp the value using a distributed counter hosted at the dequeuer.

Additionally, the dequeuer hosts an array whose entries each corresponds with an enqueuer. Each entry stores the minimum timestamp of the local SPSC of the corresponding enqueuer.

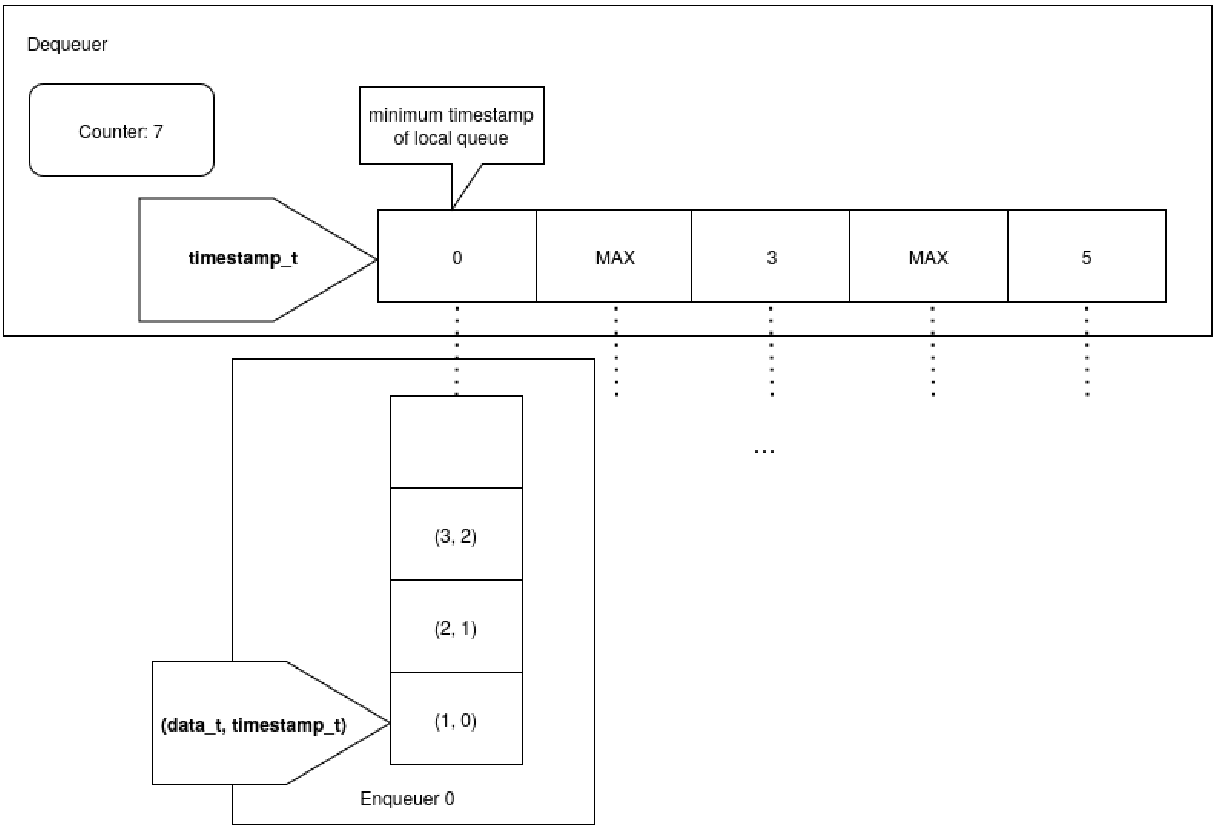


Figure 21: Basic structure of Slotqueue.

4.4.2 Data structure

We first introduce the types and shared variables utilized in Slotqueue.

Types

> `data_t` = The type of data stored.
>
> `timestamp_t` = `uint64_t`
>
> `spsc_t` = The type of the SPSC each enqueuer uses, this is assumed to be the distributed SPSC in Section 4.2.

Shared variables

> `Slots`: `remote<timestamp_t*>`
>
> > An array of `timestamp_t` with the number of entries equal to the number of enqueuers.
> >
> > Hosted at the dequeuer.
>
> `Counter`: `remote<uint64_t>`
>
> > A distributed counter.
> >
> > Hosted at the dequeuer.

Similar to the idea of assigning an order to each enqueuer in dLTQueue, the following procedure computes an enqueuer's order based on its rank:

Procedure 22: `uint64_t enqueuerOrder(uint64_t enqueuer_rank)`

```
return enqueuer_rank > Dequeuer_rank ? enqueuer_rank - 1 : enqueuer_rank
```

Again, each enqueuer is assigned an order in the range `[0, size - 2]`, with `size` being the number of processes and the total ordering among the enqueuers based on their ranks is the same as the total ordering among the enqueuers based on their orders.

Reversely, `enqueuerRank` computes an enqueuer's rank given its order.

Procedure 23: `uint64_t enqueuerRank(uint64_t enqueuer_order)`

```
return enqueuer_order >= Dequeuer_rank ? enqueuer_order + 1 : enqueuer_order
```

Enqueuer-local variables

- `Dequeuer_rank`: `uint64_t`
  - The rank of the dequeuer.
- `Enqueuer_count`: `uint64_t`
  - The number of enqueuers.
- `Self_rank`: `uint32_t`
  - The rank of the current enqueuer process.
- `Spsc`: `spsc_t`
  - This SPSC is synchronized with the dequeuer.

Dequeuer-local variables

- `Dequeuer_rank`: `uint64_t`
  - The rank of the dequeuer.
- `Enqueuer_count`: `uint64_t`
  - The number of enqueuers.
- `Spscs`: **array** of `spsc_t` with `Enqueuer_count` entries.
  - The entry at index i corresponds to the `Spsc` at the enqueuer with an order of i.

Initially, the enqueuer and the dequeuer are initialized as follows.

Enqueuer initialization

- Initialize `Dequeuer_rank`.
- Initialize `Enqueuer_count`.
- Initialize `Self_rank`.
- Initialize the local `Spsc` to its initial state.

Dequeuer initialization

- Initialize `Dequeuer_rank`.
- Initialize `Enqueuer_count`.
- Initialize `Counter` to 0.
- Initialize the `Slots` array with size equal to the number of enqueuers and every entry is initialized to `MAX_TIMESTAMP`.

> Initialize the `Spscs` array, the `i`-th entry corresponds to the `Spsc` variable of the enqueuer of order `i`.

4.4.3 Algorithm

The enqueuer operations are given as follows.

Procedure 24: `bool enqueue(data_t v)`

```
3 timestamp = fetch_and_add_sync(Counter)
4 if (!spsc_enqueue(&Spsc, (v, timestamp))) return false
5 if (!refreshEnqueue(timestamp))
6   refreshEnqueue(timestamp)
7 return true
```

To enqueue a value, `enqueue` first obtains a timestamp by FAA-ing the distributed counter (Line 3). It then tries to enqueue the value tagged with the timestamp (Line 4). At Line 5 - Line 6, the enqueuer tries to refresh its slot's timestamp.

Procedure 25: `bool refreshEnqueue(timestamp_t ts)`

```
8  enqueuer_order = enqueueOrder(Self_rank)
9  front = (data_t {}, timestamp_t {})
10 success = spsc_readFront(Spsc, &front)
11 new_timestamp = success ? front.timestamp : MAX_TIMESTAMP
12 if (new_timestamp != ts)
13   return true
14 old_timestamp = timestamp_t {}
15 aread_sync(&Slots, enqueuer_order, &old_timestamp)
16 success = spsc_readFront(Spsc, &front)
17 new_timestamp = success ? front.timestamp : MAX_TIMESTAMP
18 if (new_timestamp != ts)
19   return true
20 return compare_and_swap_sync(Slots, enqueuer_order,
       old_timestamp,
       new_timestamp)
```

`refreshEnqueue`'s responsibility is to refresh the timestamp stores in the enqueuer's slot to potentially notify the dequeuer of its newly-enqueued element. It first reads the current front element (Line 10). If the SPSC is empty, the new timestamp is set to `MAX_TIMESTAMP`, otherwise, the front element's timestamp (Line 11). If it finds that the front element's timestamp is different from the timestamp `ts` it returns `true` immediately (Line 12 - Line 13). Otherwise, it reads its slot's old timestamp (Line 15) and re-reads the current front element in the SPSC (Line 16) to update the new timestamp. Note that similar to Line 13, `refreshEnqueue` immediately succeeds if the new timestamp is different from the timestamp `ts` of the element it enqueues (Line 19). Otherwise, it tries to CAS its slot's timestamp with the new timestamp (Line 20).

The dequeuer operations are given as follows.

Procedure 26: `bool dequeue(data_t* output)`

```
21 rank = readMinimumRank()
22 if (rank == DUMMY_RANK)
23   return false
24 output_with_timestamp = (data_t {}, timestamp_t {})
25 if (!spsc_dequeue(Spsc, &output_with_timestamp))
26   return false
27 *output = output_with_timestamp.data
28 if (!refreshDequeue(rank))
29   refreshDequeue(rank)
30 return true
```

To dequeue a value, `dequeue` first reads the rank of the enqueuer whose slot currently stores the minimum timestamp (Line 21). If the obtained rank is `DUMMY_RANK`, failure is signaled (Line 22 - Line 23). Otherwise, it tries to dequeue the SPSC of the corresponding enqueuer (Line 25). It then tries to refresh the enqueuer's slot's timestamp to potentially notify the enqueuer of the dequeue (Line 28 - Line 29). It then signals success (Line 30).

Procedure 27: `uint64_t readMinimumRank()`

```
31 buffered_slots = timestamp_t[Enqueuer_count] {}
32 for index in 0..Enqueuer_count
33 | aread_sync(Slots, index, &bufferred_slots[index])
34 if every entry in bufferred_slots is MAX_TIMESTAMP
35 | return DUMMY_RANK
36 let rank be the index of the first slot that contains the minimum timestamp
   among bufferred_slots
37 for index in 0..rank
38 | aread_sync(Slots, index, &bufferred_slots[index])
39 min_timestamp = MAX_TIMESTAMP
40 for index in 0..rank
41 | timestamp = buffered_slots[index]
42 | if (min_timestamp < timestamp)
43 | | min_rank = enqueuerRank(index)
44 | | min_timestamp = timestamp
45 return min_rank
```

`readMinimumRank`'s main responsibility is to return the rank of the enqueuer from which we can safely dequeue next. It first creates a local buffer to store the value read from `Slots` (Line 31). It then performs 2 scans of `Slots` and read every entry into `buffered_slots` (Line 32 - Line 38). If the first scan finds only `MAX_TIMESTAMP`s, `DUMMY_RANK` is returned (Line 35). From there, based on `bufferred_slots`, it returns the rank of the enqueuer whose bufferred slot stores the minimum timestamp (Line 40 - Line 45).

Procedure 28: `refreshDequeue(rank: int)` **returns** `bool`

```
48 enqueuer_order = enqueuerOrder(rank)
49 old_timestamp = timestamp_t {}
50 aread_sync(&Slots, enqueuer_order, &old_timestamp)
51 front = (data_t {}, timestamp_t {})
52 success = spsc_readFront(Spscs[enqueuer_order], &front)
53 new_timestamp = success ? front.timestamp : MAX_TIMESTAMP
54 return compare_and_swap_sync(Slots, enqueuer_order,
       old_timestamp,
       new_timestamp)
```

`refreshDequeue`'s responsibility is to refresh the timestamp of the just-dequeued enqueuer to notify the enqueuer of the dequeue. It first reads the old timestamp of the slot (Line 50) and the front element (Line 52). If the SPSC is empty, the new timestamp is set to `MAX_TIMESTAMP`, otherwise, it's the front element's timestamp (Line 53). It finally tries to CAS the slot with the new timestamp (Line 54).

Chapter V Preliminary results

This section introduces our benchmarking process, including our setup, environment, interested metrics and our microbenchmark program. Most importantly, we showcase the premilinaries results on how well our novel algorithms perform, especially Slotqueue. We conclude this section with a discussion about the implications of these results.

Currently, performance-related properties are of our main focus.

5.1 Benchmarking metrics

This section provides an overview of the metrics we're interested in our algorithms. Performance-wise, latency and throughput are the two most popular metrics. These metrics revolve around the concept of "task". In our context, a task is a single method call of an MPSC queue algorithm, e.g enqueue and dequeue. Note that in our discussion, any two tasks are independent. Roughly speaking, two tasks are independent if one does not need to depend on the output of another for it to finish or there doesn't exist a bigger task that needs to depend on the output of the tasks. This rules out pipeline parallelism, where a task needs to wait for the output of a preceding task, and data parallelism, where a big task is split into and needs to wait for the outputs of multiple smaller tasks.

5.1.1 Throughput

Throughput is number of operations finished in a unit of time. Its unit is often given as ops/s (operations per second), ops/ms (operations per milliseconds) or ops/us (operations per microsecond). Intuitively, throughput is closest to our notion of "performance": The higher the throughput, the more tasks are done in a unit of time and thus, the higher the performance. The implication is that our ultimate goal is to optimize the throughput metric of our algorithms.

5.1.2 Latency

Latency is the time it takes for a single task to complete. Its unit is often given as s/op (seconds per operation), ms/op (milliseconds per operation) or us/op (microseconds per operation).

Intuitively, to optimize latency, one should minimize the number of execution steps required by a task. Therefore, it's obvious that optimizing for latency is much clearer than optimizing for throughput.

In concurrent algorithms, multiple tasks are executed by multiple processes. The key observation is that, if we fix the number of processes, the lower the average latency of a task, the larger the number of tasks that can be completed by a process, which implies a higher throughput. Therefore, a good latency often (but not always) implies a good throughput.

From the two points above, we can see that latency is a more intuitive metric to optimize for, while being quite indicative of the algorithm's performance.

One question is how to optimize for latency? As we have discussed, we should minimize the number of execution steps. A key observation is that when the number of processes grows, contention should also grow, thus, causing the number of steps taken by a task to grow and thus, the average latency to deteriorate. Note that if we manage to keep the average latency of a task fixed while also increasing the number of processes, we gain higher throughput due to higher concurrency. The actionable insight is that if we minimize contention in our algorithms, our algorithm should scale with the number of processes.

Following this discussion, we should aim to discover and optimize out highly contended areas in our algorithms if we want to make them scale well to a large number of nodes/processes.

5.2 Benchmarking baselines

We use three MPSC queue algorithms as benchmarking baselines:
- dLTQueue + our custom SPSC: Our most optimized version of LTQueue while still keeping the core algorithm in tact.
- Slotqueue + our custom SPSC: Our modification to dLTQueue to obtain a more optimized distributed version of LTQueue.
- AMQueue [1]: A hosted bounded MPSC queue algorithm, already detailed in Section 3.2.

5.3 Microbenchmark program

Our microbenchmark is as follows:
- All processes share a single MPSC, one of the processes is a dequeuer, and the rest are enqueuers.
- The enqueuers enqueue a total of 10^4 elements.
- The dequeuer dequeue out 10^4 elements.
- For MPSC, the MPSC is warmed up before the dequeuer starts.

We measure the latency and throughput of the enqueue and dequeue operation. This microbenchmark is repeated 5 times for each algorithm and we take the mean of the results.

5.4 Benchmarking setup

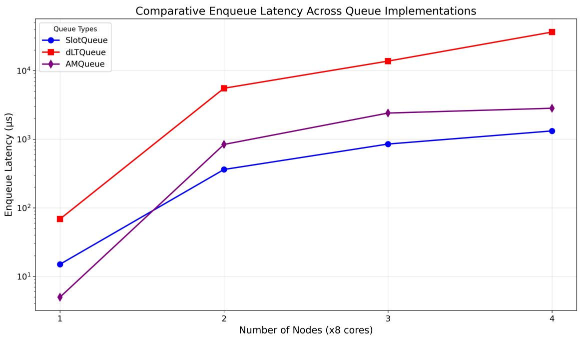
The experiments are carried out on a four-node cluster resided in HPC Lab at Ho Chi Minh University of Technology. Each node is an Intel Xeon CPU e5-2680 v3 with has 8 cores and 16 GB RAM. The interconnect used is Ethernet and so does not support true one-sided communication.

The operating system used is Ubuntu 22.04.5. The MPI implementation used is MPICH version 4.0, released on January 21st, 2022.

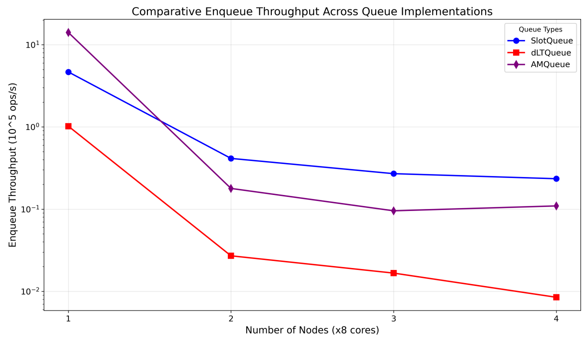
We run the producer-consumer microbenchmark on 1 to 4 nodes to measure both the latency and performance of our MPSC algorithms.

5.5 Benchmarking results

Figure 22, Figure 23 and Figure 24 showcase our benchmarking results, with the y-axis drawin in log scale.

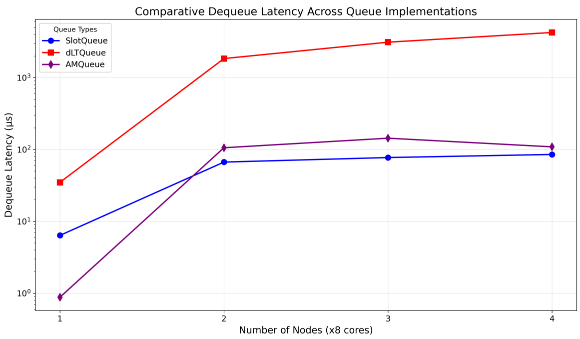


(a) Enqueue latency benchmark results.

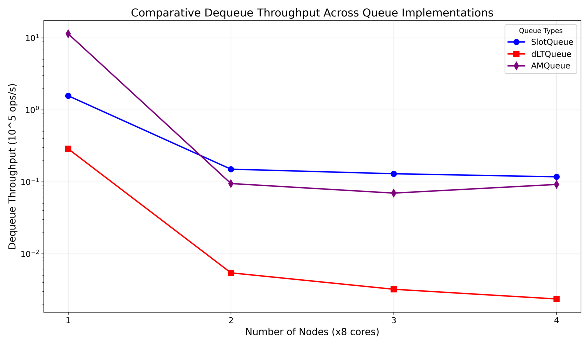


(b) Enqueue throughput benchmark results

Figure 22: Microbenchmark results for enqueue operation.



(a) Dequeue latency benchmark results.



(b) Dequeue throughput benchmark results

Figure 23: Microbenchmark results for dequeue operation.

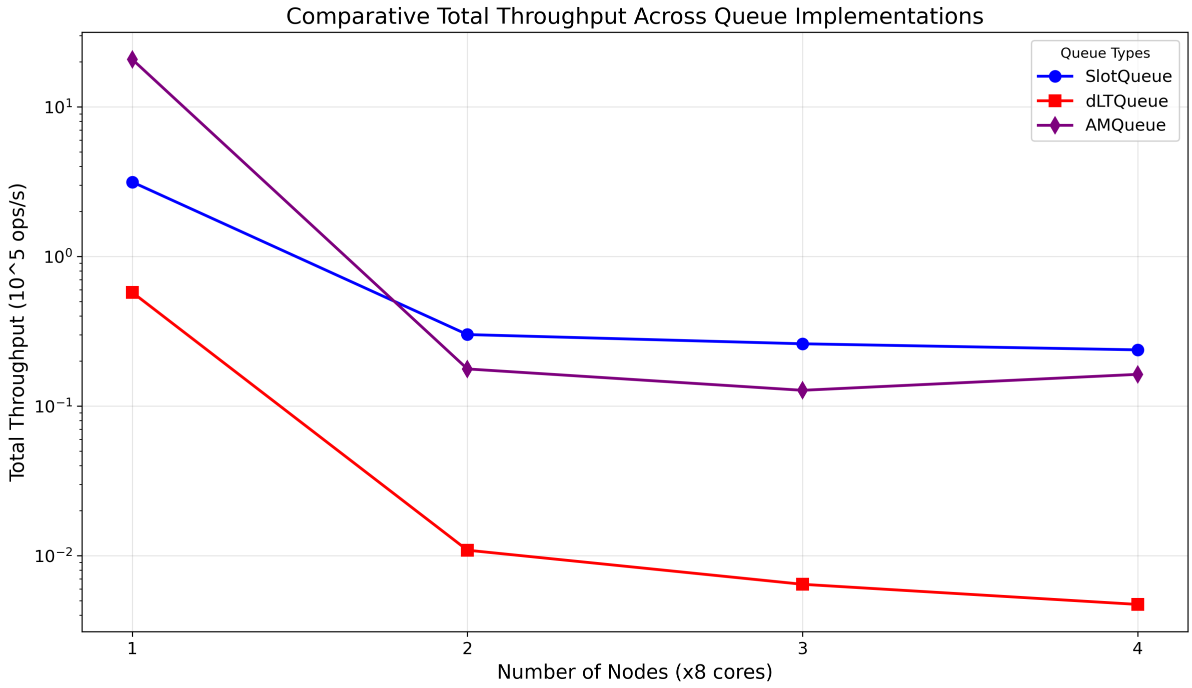


Figure 24: Microbenchmark results for total throughput.

The most evident thing is that Figure 24 and Figure 23b are almost identical. This backs our claim that in an MPSC queue, the performance is bottlenecked in the dequeuer.

For enqueue latency and throughput, dLTQueue performs far better than dLTQueue while being slightly better than AMQueue. This is in line with our theoretical projection in Table 4. One concerning trend is that Slotqueue's enqueue throughput seems to degrade with the number of nodes, which signals a potential scalability problem. This is problematic further in that our theoretical model suggests that the cost of enqueue is always fixed. This is to be investigated further in the future.

For dequeue latency and throughput, Slotqueue and AMQueue can quite match each other, while being better than dLTQueue. This is expected, agreeing with our projection of dequeue wrapping overhead in Table 4. Furthermore, Slotqueue is conceived as a more dequeuer-optimized version of dLTQueue. Based on this empirical result, it's reasonable to believe this is to be the case. Unlike enqueue, dequeue latency of Slotqueue seems to be quite stable, increasing very slowly. Because the dequeuer is the bottleneck of an MPSC, this is a good sign for the scalability of Slotqueue.

In conclusion, based on Figure 24, Slotqueue seems to perform better than dLTQueue and AMQueue in terms of both enqueue and dequeue operations, latency-wise and throughput-wise. The overhead of logarithmic-order number of remote operations in dLTQueue seems to be costly, adversely affecting its performance when the number of nodes increases. Additionally, compared to AMQueue, dLTQueue and Slotqueue also have the advantage of fault-tolerance, which due to the blocking nature of AMQueue, cannot be promised.

Chapter VI Conclusion & Future works

In this thesis, we have looked into the principles of shared-memory programming e.g. the use of atomic operations, to model and design distributed MPSC queue algorithms. We specifically investigate the existing MPSC queue algorithms in the shared memory literature and adapt them for distributed environments using our model. Following this, we have proposed two new distributed MPSC queue algorithms: dLTQueue and Slotqueue. We have proven various interested theoretical aspects of these algorithms, namely, correctness, fault-tolerance and performance. To reflect on what we have obtained theoretically, we have conducted some benchmarks on how queues behave, using another algorithm known as active-message queue (AMQueue) from [1]. We have discussed some anomalies discovered via the combined application of theory and epiricism. This lays the foundation for our next steps, which is listed Table 5.

Weeks	**Work**
1-3	• Adapt Jiffy to distributed environment. • Discover optimization opportunities with dLTQueue and Slotqueue.
4-6	• Perform benchmarks on RDMA cluster and investigate the performance degradation problem.
7-9	• Incorporate MPI-3's new support for shared-memory windows and C++11 atomic operations to optimize intra-node communication.
10-12	• Perform more thorough benchmarks and discover more benchmarking baselines for our MPSC queues.
13-15	• Finalize our results and provide insights from our research.

Table 5: Future works for the next semester

Appendix A Theoretical aspects

This section discusses the correctness and progress guarantee properties of the distributed MPSC queue algorithms introduced in Chapter IV. We also provide a theoretical performance model of these algorithms to predict how well they scale to multiple nodes.

A.1 Terminology

In this section, we introduce some terminology that we will use throughout our proofs.

Definition A.1.1 In an SPSC/MPSC queue, an enqueue operation e is said to **match** a dequeue operation d if d returns the value that e enqueues. Similarly, d is said to **match** e. In this case, both e and d are said to be **matched**.

Definition A.1.2 In an SPSC/MPSC queue, an enqueue operation e is said to be **unmatched** if no dequeue operation **matches** it.

Definition A.1.3 In an SPSC/MPSC queue, a dequeue operation d is said to be **unmatched** if no enqueue operation **matches** it, in other word, d returns `false`.

A.2 Preliminaries

In this section, we formalize the notion of correct concurrent algorithms and harmless ABA problem. We will base our proofs on these formalisms to prove their correctness. We also provide a simple way to theoretically model our queues' performance.

A.2.1 Linearizability

Linearizability is a criteria for evaluating a concurrent algorithm's correctness. This is the model we use to prove our algorithm's correctness. Our formalization of linearizability is equivalent to that of [10] by Herlihy and Shavit. However, there are some differences in our terminology.

For a concurrent object `S`, we can call some methods on `S` concurrently. A method call on the object `S` is said to have an **invocation event** when it starts and a **response event** when it ends.

Definition A.2.1.1 An **invocation event** is a triple $(S, t, args)$, where S is the object the method is invoked on, t is the timestamp of when the event happens and $args$ is the arguments passed to the method call.

Definition A.2.1.2 A **response event** is a triple (S, t, res), where S is the object the method is invoked on, t is the timestamp of when the event happens and res is the results of the method call.

Definition A.2.1.3 A **method call** is a tuple of (i, r) where i is an invocation event and r is a response event or the special value $\perp$ indicating that its response event hasn't happened yet. A well-formed **method call** should have a reponse event with a larger timestamp than its invocation event or the response event hasn't happened yet.

Definition A.2.1.4 A **method call** is **pending** if its invocation event is $\perp$.

Definition A.2.1.5 A **history** is a set of well-formed **method calls**.

Definition A.2.1.6 An extension of **history** H is a **history** H' such that any pending method call is given a response event.

We can define a **strict partial order** on the set of well-formed method calls:

Definition A.2.1.7 $\rightarrow$ is a relation on the set of well-formed method calls. With two method calls X and Y, we have $X \rightarrow Y \Leftrightarrow X$'s response event is not $\perp$ and its response timestamp is not greater than Y's invocation timestamp.

Definition A.2.1.8 Given a **history** H, $\rightarrow_H$ is a relation on H such that for two method calls X and Y in H, $X \rightarrow_H Y \Leftrightarrow X \rightarrow Y$.

Definition A.2.1.9 A **sequential history** H is a **history** such that $\rightarrow_H$ is a total order on H.

Now that we have formalized the way to describe the order of events via **histories**, we can now formalize the mechanism to determine if a **history** is valid. The easier case is for a **sequential history**.

Definition A.2.1.10 For a concurrent object S, a **sequential specification** of S is a function that either returns `true` (valid) or `false` (invalid) for a **sequential history** H.

The harder case is handled via the notion of **linearizable**.

Definition A.2.1.11 A history H on a concurrent object S is **linearizable** if it has an extension H' and there exists a *sequential history* H_S such that:

1. The **sequential specification** of S accepts H_S.
2. There exists a one-to-one mapping M of a method call $(i, r) \in H'$ to a method call $(i_S, r_S) \in H_S$ with the properties that:
   - i must be the same as i_S except for the timestamp.
   - r must be the same r_S except for the timestamp or r.
3. For any two method calls X and Y in H',
   $X \rightarrow_{H'} Y \Rightarrow M(X) \rightarrow_{H_S} M(Y)$.

We consider a history to be valid if it's linearizable.

A.2.1.1 Linearizable SPSC

Our SPSC supports 3 methods:

- `enqueue` which accepts an input parameter and returns a boolean.
- `dequeue` which accepts an output parameter and returns a boolean.
- `readFront` which accepts an output parameter and returns a boolean.

Definition A.2.1.1.12 An SPSC is **linearizable** if and only if any history produced from the SPSC that does not have overlapping dequeue method calls and overlapping enqueue method calls is *linearizable* according to the following *sequential specification*:

- An enqueue can only be matched by one dequeue.
- A dequeue can only be matched by one enqueue.
- The order of item dequeues is the same as the order of item enqueues.
- An enqueue can only be matched by a later dequeue.
- A dequeue returns `false` when the queue is empty.
- A dequeue returns `true` and matches an enqueue when the queue is not empty.
- An enqueue returns `false` when the queue is full.
- An enqueue would return `true` when the queue is not full and the number of elements should increase by one.
- A read-front would return `false` when the queue is empty.
- A read-front would return `true` and the first element in the queue is read out.

A.2.1.2 Linearizable MPSC queue

An MPSC queue supports 2 methods:

- `enqueue` which accepts an input parameter and returns a boolean.
- `dequeue` which accepts an output parameter and returns a boolean.

Definition A.2.1.2.13 An MPSC queue is **linearizable** if and only if any history produced from the MPSC queue that does not have overlapping dequeue method calls is *linearizable* according to the following *sequential specification*:

- An enqueue can only be matched by one dequeue.
- A dequeue can only be matched by one enqueue.
- The order of item dequeues is the same as the order of item enqueues.
- An enqueue can only be matched by a later dequeue.
- A dequeue returns `false` when the queue is empty.
- A dequeue returns `true` and matches an enqueue when the queue is not empty
- An enqueue that returns `true` will be matched if there are enough dequeues after that.
- An enqueue that returns `false` will never be matched.

A.2.2 ABA-safety

Not every ABA problem is unsafe. We formalize in this section which ABA problem is safe and which is not.

Definition A.2.2.14 A **modification instruction** on a variable `v` is an atomic instruction that may change the value of `v` e.g. a store or a CAS.

Definition A.2.2.15 A **successful modification instruction** on a variable `v` is an atomic instruction that changes the value of `v` e.g. a store or a successful CAS.

Definition A.2.2.16 A **CAS-sequence** on a variable `v` is a sequence of instructions of a method m such that:

- The first instruction is a load $v_0 = \texttt{load}(v)$.
- The last instruction is a `CAS(&`v`,`v_0`,`v_1`)`.
- There's no modification instruction on `v` between the first and the last instruction.

Definition A.2.2.17 A **successful CAS-sequence** on a variable `v` is a **CAS-sequence** on `v` that ends with a successful CAS.

Definition A.2.2.18 Consider a method m on a concurrent object S. m is said to be **ABA-safe** if and only if for any history of method calls produced from S, we can reorder any successful CAS-sequences inside an invocation of m in the following fashion:

- If a successful CAS-sequence is part of an invocation of m, after reordering, it must still be part of that invocation.
- If a successful CAS-sequence by an invocation of m precedes another by that invocation, after reordering, this ordering is still respected.
- Any successful CAS-sequence by an invocation of m after reordering must not overlap with a successful modification instruction on the same variable.
- After reordering, all method calls' response events on the concurrent object S stay the same.

A.2.3 Theoretical performance

We use a simple performance model, projecting the performance of a distributed algorithm by calculating or approximating how many remote operations and local operations are made in the usual case. An example of our model has been shown in the dequeue and enqueue time-complexity rows of Table 4.

A.3 Theoretical proofs of the distributed SPSC

In this section, we focus on the correctness and progress guarantee of the simple distributed SPSC established in Section 4.2.

A.3.1 Correctness

This section establishes the correctness of our distributed SPSC.

A.3.1.1 ABA problem

There's no CAS instruction in our simple distributed SPSC, so there's no potential for ABA problem.

A.3.1.2 Memory reclamation

There's no dynamic memory allocation and deallocation in our simple distributed SPSC, so it is memory-safe.

A.3.1.3 Linearizability

We prove that our simple distributed SPSC is linearizable.

Theorem A.3.1.3.1 *(Linearizability of the simple distributed SPSC)* The distributed SPSC given in Section 4.2 is linearizable.

Proof We claim that the following are the linearization points of our SPSC's methods:

- The linearization point of an `spsc_enqueue` call (Procedure 4) that returns `false` is Line 3.
- The linearization point of an `spsc_enqueue` call (Procedure 4) that returns `true` is Line 7.
- The linearization point of an `spsc_dequeue` call (Procedure 6) that returns `false` is Line 19.
- The linearization point of an `spsc_dequeue` call (Procedure 6) that returns `true` is Line 23.
- The linearization point of $\texttt{spsc\_readFront}_\texttt{e}$ call (Procedure 5) that returns `false` is Line 11 or Line 12 if Line 11 is passed.
- The linearization point of $\texttt{spsc\_readFront}_\texttt{e}$ call (Procedure 5) that returns `true` is Line 12.
- The linearization point of $\texttt{spsc\_readFront}_\texttt{d}$ call (Procedure 7) that returns `false` is Line 27.
- The linearization point of $\texttt{spsc\_readFront}_\texttt{d}$ call (Procedure 7) that returns `true` is right after Line 27 or right before Line 30 if Line 27 is never executed.

We define a total ordering $<$ on the set of completed method calls based on these linearization points: If the linearization point of a method call A is before the linearization point of a method call B, then $A < B$.

If the distributed SPSC is linearizable, $<$ would define a equivalent valid sequential execution order for our SPSC method calls.

A valid sequential execution of SPSC method calls would possess the following characteristics.

An enqueue can only be matched by one dequeue: Each time an `spsc_dequeue` is executed, it advances the `First` index. Because only one dequeue can happen at a time, it's guaranteed that each dequeue proceeds with one unique `First` index. Two dequeues can only dequeue out the same entry in the SPSC's array if their `First` indices are congruent modulo `Capacity`. However, by then, this entry must have been overwritten. Therefore, an enqueue can only be dequeued at most once.

A dequeue can only be matched by one enqueue: This is trivial, as based on how Procedure 6 is defined, a dequeue can only dequeue out at most one value.

The order of item dequeues is the same as the order of item enqueues: To put more precisely, if there are 2 `spsc_enqueues` e_1, e_2 such that $e_1 < e_2$, then either e_2 is unmatched or e_1 matches d_1 and e_2 matches d_2 such that $d_1 < d_2$. If e_2 is unmatched, the statement holds. Suppose e_2 matches d_2. Because $e_1 < e_2$, based on how Procedure 4 is defined, e_1 corresponds to a value i_1 of `Last` and e_2 corresponds to a value i_2 of `Last` such that $i_1 < i_2$. Based on how Procedure 6 is defined, each time a dequeue happens successfully, `First` would be incremented. Therefore, for e_2 to be matched, e_1 must be matched first because `First` must surpass i_1 before getting to i_2. In other words, e_1 matches d_1 such that $d_1 < d_2$.

An enqueue can only be matched by a later dequeue: To put more precisely, if an `spsc_enqueue` e matches an `spsc_dequeue` d, then $e < d$. If e hasn't executed its linearization point at Line 7, there's no way d's Line 22 can see e's value. Therefore, d's linearization point at Line 23 must be after e's linearization point at Line 7. Therefore, $e < d$.

A dequeue would return `false` when the queue is empty: To put more precisely, for an `spsc_dequeue` d, if by d's linearization point, every successful `spsc_enqueue` e' such that $e' < d$ has been matched by d' such that $d' < d$, then d would be unmatched and return `false`. By this assumption, any `spsc_enqueue` e that has executed its linearization point at Line 7 before d's Line 18 has been matched. Therefore, `First = Last` at Line 18, or `First >= Last_buf`, therefore, the if condition at Line 18 - Line 21 is entered. Also by the assumption, any `spsc_enqueue` e that has executed its linearization point at Line 7 before d's Line 20 has been matched. Therefore, `First = Last` at Line 20. Then, Line 21 is executed and d returns `false`.

A dequeue would return `true` and match an enqueue when the queue is not empty: To put more precisely, for an `spsc_dequeue` d, if there exists a successful `spsc_enqueue` e' such that $e' < d$ and has not been matched by a dequeue d' such that $d' < e'$, then d would be match some e and return `true`. By this assumption, some e' must have executed its linearization point at Line 7 but is still unmatched by the time d starts. Then, `First < Last`, so d must match some enqueue e and returns `true`.

An enqueue would return `false` when the queue is full: To put more precisely, for an `spsc_enqueue` e, if by e's linearization point, the number of unmatched successful `spsc_enqueue` $e' < e$ by the time e starts equals `Capacity`, then e returns `false`. By this assumption, any d' that matches e' must satisfy $e < d'$, or d' must execute its synchronization point at Line 23 after Line 1 and Line 4 of e, then e's Line 5 must have executed and return `false`.

An enqueue would return `true` when the queue is not full and the number of elements should increase by one: To put more precisely, for an `spsc_enqueue` e, if by e's linearization point, the number of unmatched successful `spsc_enqueue` $e' < e$ by the time e starts is fewer than `Capacity`, then e returns `true`. By this assumption, `First < Last` at least until e's linearization point and because Line 7 must be executed, which means the number of elements should increase by one.

A read-front would return `false` when the queue is empty: To put more precisely, for a read-front r, if by r's linearization point, every successful `spsc_enqueue` e' such that $e' < r$ has been matched by d' such that $d' < d$, then r would return `false`. That means any unmatched successful `spsc_enqueue` e must have executed its linearization point at Line 7 after r's, or `First = Tail` before r's linearization point

- For an enqueuer's read-front, if r doesn't pass Line 10, the statement holds. If r passes Line 10, by the assumption, r would execute Line 14, because r sees that `First = Tail`.
- For an dequeuer's read-front, r must enter Line 27 because `First_buf = Tail_buf`, due to from the dequeuer's point of view, `First_buf = First` and `Last_buf <= Last`. Similarly, r must execute Line 29 and return `false`.

A read-front would return `true` and the first element in the queue is read out: To put more precisely, for a read-front r, if before r's linearization point, there exists some unmatched successful `spsc_enqueue` e' such that $e' < r$, then r would read out the same value as the first d such that $r < d$. By this assumption, any d' that matches some of these successful `spsc_enqueue` e' must execute its linearization point at Line 23 after r's linearization point. Therefore, `First < Last` until r's linearization point.

- For an enqueuer's read-front, r must not execute Line 11 and Line 14. Therefore, Line 15 is executed, and `First_buf` at this point is the same as `First_buf` of the first d such that $r < d$, because we have just read it at Line 12, and any successful $d' > r$ must execute Line 23 after Line 15, therefore, `First` has no chance to be incremented between Line 12 and Line 15.
- For a dequeuer's read-front, r must not execute Line 27 - Line 29 and execute Line 30 instead. It's trivial that r reads out the same value as the first dequeue d such that $r < d$ because there can only be one dequeuer.

In conclusion, for any completed history of method calls our SPSC can produce, we have defined a way to sequentially order them in a way that conforms to SPSC's sequential specification. By Definition A.2.1.1.12, our SPSC is linearizable. □

A.3.2 Progress guarantee

Our simple distributed SPSC is wait-free:

- `spsc_dequeue` (Procedure 6) does not execute any loops or wait for any other method calls.
- `spsc_enqueue` (Procedure 4) does not execute any loops or wait for any other method calls.
- `spsc_readFront`$_e$ (Procedure 5) does not execute any loops or wait for any other method calls.
- `spsc_readFront`$_d$ (Procedure 7) does not execute any loops or wait for any other method calls.

A.3.3 Theoretical performance

A summary of the theoretical performance of our simple SPSC is provided in Table 6. In the following discussion, R means remote operations and L means local operations.

Operations	Time-complexity
`spsc_enqueue`	$R + L$
`spsc_dequeue`	$R + L$
`spsc_readFront`$_e$	$R + L$
`spsc_readFront`$_d$	R

Table 6: Theoretical performance summary of our simple distributed SPSC. R means remote operations and L means local operations.

For `spsc_enqueue`, we consider the procedure Procedure 4. In the usual case, the remote operation on Line 3 is skipped and so only 2 remote puts are performed on Line 6 and Line 7. The Data array on Line 6 is hosted on the enqueuer, so this is actually a local operation, while the control variable is hosted on the dequeuer, so Line 7 is truly a remote operation. Therefore, theoretically, it's one remote operation plus a local one.

For `spsc_dequeue`, we consider the procedure Procedure 6. Similarly, in the usual case, the remote operation on Line 19 is skipped and only the 2 lines Line 22 and Line 23 are executed always. Here, it's the other way around, the access to the `Data` array on Line 22 is a truly remote operation while the access to the First control variable is a local one. Therefore, theoretically, it's one remote operation plus a local one.

For `spsc_readFront`$_e$, we consider the procedure Procedure 5. The operation on Line 12 is a truly remote operation, as the `First` control variable is hosted on the dequeuer. The operation on Line 15 is a remote operation, as the `Data` array is hosted on the enqueuer. This means, theoretically, it also takes one remote operation plus a local one.

For `spsc_readFront`$_d$, we consider the procedure Procedure 7. Only the operation on Line 30 is executed always, which results in a truly remote operation as the Data array is hosted on the enqueuer. Therefore, it only takes one remote operation.

A.4 Theoretical proofs of dLTQueue

In this section, we provide proofs covering all of our interested theoretical aspects in dLTQueue.

A.4.1 Proof-specific notations

The structure of dLTQueue is presented again in Figure 25.

As a reminder, the bottom rectangular nodes are called the **enqueuer nodes** and the circular node are called the **tree nodes**. Tree nodes that are attached to an enqueuer node are called **leaf nodes**, otherwise, they are called **internal nodes**. Each **enqueuer node** is hosted on the enqueuer that corresponds to it. The enqueuer nodes accomodate an instance of our distributed SPSC in Section 4.2 and a `Min_timestamp` variable representing the minimum timestamp inside the SPSC. Each **tree node** stores a rank of a enqueuer that's attached to the subtree which roots at the **tree node**.

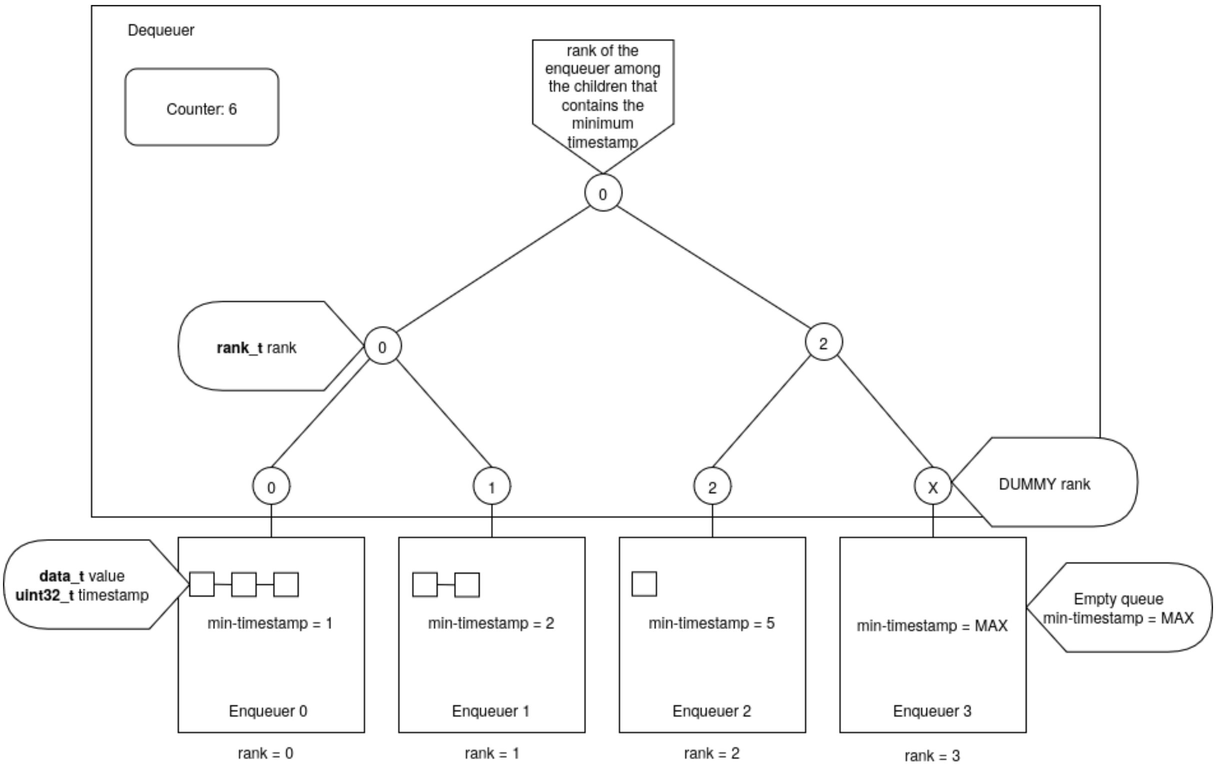


Figure 25: dLTQueue's structure.

We will refer `propagate`$_e$ and `propagate`$_d$ as `propagate` if there's no need for discrimination. Similarly, we will sometimes refer to `refreshNode`$_e$ and `refreshNode`$_d$ as `refreshNode`, `refreshLeaf`$_e$ and `refreshLeaf`$_d$ as `refreshLeaf`, `refreshTimestamp`$_e$ and `refreshTimestamp`$_d$ as `refreshTimestamp`.

Definition A.4.1.1 For a tree node n, the rank stored in n at time t is denoted as $rank(n, t)$.

Definition A.4.1.2 For an enqueue or a dequeue `op`, the rank of the enqueuer it affects is denoted as $rank(\texttt{op})$.

Definition A.4.1.3 For an enqueuer whose rank is r, the `Min_timestamp` value stored in its enqueuer node at time t is denoted as $min\text{-}ts(r, t)$. If r is `DUMMY_RANK`, $min\text{-}ts(r, t)$ is `MAX_TIMESTAMP`.

Definition A.4.1.4 For an enqueuer with rank r, the minimum timestamp among the elements between `First` and `Last` in its SPSC at time t is denoted as $min\text{-}spsc\text{-}ts(r, t)$. If r is dummy, $min\text{-}spsc\text{-}ts(r, t)$ is `MAX`.

Definition A.4.1.5 For an enqueue or a dequeue `op`, the set of nodes that it calls `refreshNode` (Procedure 15 or Procedure 20) or `refreshLeaf` (Procedure 16 or Procedure 21) on is denoted as $path(\texttt{op})$.

Definition A.4.1.6 For an enqueue or a dequeue, **timestamp-refresh phase** refer to its execution of Line 19 - Line 20 in `propagate`$_e$ (Procedure 13) or Line 72 - Line 73 in `propagate`$_d$ (Procedure 18).

Definition A.4.1.7 For an enqueue `op`, and a node $n \in path(\texttt{op})$, **node-n-refresh phase** refer to its execution of:

- Line 21 - Line 22 of `propagate`$_e$ (Procedure 13) if n is a leaf node.
- Line 26 - Line 27 of `propagate`$_e$ (Procedure 13) to refresh n's rank if n is a non-leaf node.

Definition A.4.1.8 For a dequeue `op`, and a node $n \in path(\texttt{op})$, **node-n-refresh phase** refer to its execution of:

- Line 74 - Line 75 of `propagate`$_d$ (Procedure 18) if n is a leaf node.
- Line 79 - Line 80 of `propagate`$_d$ (Procedure 18) to refresh n's rank if n is a non-leaf node.

Definition A.4.1.9 `refreshTimestamp`$_e$ (Procedure 14) is said to start its **CAS-sequence** if it finishes Line 30. `refreshTimestamp`$_e$ is said to end its **CAS-sequence** if it finishes Line 35 or Line 37.

Definition A.4.1.10 `refreshTimestamp`$_d$ (Procedure 19) is said to start its **CAS-sequence** if it finishes Line 84. `refreshTimestamp`$_d$ is said to end its **CAS-sequence** if it finishes Line 89 or Line 91.

Definition A.4.1.11 `refreshNode`$_e$ (Procedure 15) is said to start its **CAS-sequence** if it finishes Line 39. `refreshNode`$_e$ is said to end its **CAS-sequence** if it finishes Line 53.

Definition A.4.1.12 `refreshNode`$_d$ (Procedure 20) is said to start its **CAS-sequence** if it finishes Line 93. `refreshNode`$_d$ is said to end its **CAS-sequence** if it finishes Line 107.

Definition A.4.1.13 `refreshLeaf`$_e$ (Procedure 16) is said to start its **CAS-sequence** if it finishes Line 56. `refreshLeaf`$_e$ is said to end its **CAS-sequence** if it finishes Line 61.

Definition A.4.1.14 `refreshLeaf`$_d$ (Procedure 21) is said to start its **CAS-sequence** if it finishes Line 110. `refreshLeaf`$_d$ is said to end its **CAS-sequence** if it finishes Line 115.

A.4.2 Correctness

This section establishes the correctness of dLTQueue introduced in Section 4.3.

A.4.2.1 ABA problem

We use CAS instructions on:

- Line 35 and Line 37 of `refreshTimestamp`$_e$ (Procedure 14).
- Line 53 of `refreshNode`$_e$ (Procedure 15).
- Line 61 of `refreshLeaf`$_e$ (Procedure 16).
- Line 89 and Line 91 of `refreshTimestamp`$_d$ (Procedure 19).
- Line 107 of `refreshNode`$_d$ (Procedure 20).
- Line 115 of `refreshLeaf`$_d$ (Procedure 21).

Notice that at these locations, we increase the associated version tags of the CAS-ed values. These version tags are 32-bit in size, therefore, practically, ABA problem can't virtually occur. It's safe to assume that there's no ABA problem in dLTQueue.

A.4.2.2 Memory reclamation

Notice that dLTQueue pushes the memory reclamation problem to the underlying SPSC. If the underlying SPSC is memory-safe, dLTQueue is also memory-safe.

A.4.2.3 Linearizability

Theorem A.4.2.3.1 In dLTQueue, an enqueue can only match at most one dequeue.

Proof A dequeue indirectly performs a value dequeue through `spsc_dequeue`. Because `spsc_dequeue` can only match one `spsc_enqueue` by another enqueue, the theorem holds. □

Theorem A.4.2.3.2 In dLTQueue, a dequeue can only match at most one enqueue.

Proof This is trivial as a dequeue can only read out at most one value, so it can only match at most one enqueue. □

Theorem A.4.2.3.3 Only the dequeuer and one enqueuer can operate on an enqueuer node.

Proof This is trivial based on how the algorithm is defined. □

We immediately obtain the following result.

Corollary A.4.2.3.4 Only one dequeue operation and one enqueue operation can operate concurrently on an enqueuer node.

Theorem A.4.2.3.5 The SPSC at an enqueuer node contains items with increasing timestamps.

Proof Each enqueue would `FAA` the distributed counter (Line 14 in Procedure 12) and enqueue into the SPSC an item with the timestamp obtained from that counter. Applying Corollary A.4.2.3.4, we know that items are enqueued one at a time into the SPSC. Therefore, later items are enqueued by later enqueues, which obtain increasing values by `FAA`-ing the shared counter. The theorem holds. □

Theorem A.4.2.3.6 For an enqueue or a dequeue `op`, if `op` modifies an enqueuer node and this enqueuer node is attached to a leaf node l, then $path(\texttt{op})$ is the set of nodes lying on the path from l to the root node.

Proof This is trivial considering how $\texttt{propagate}_{\texttt{e}}$ (Procedure 13) and $\texttt{propagate}_{\texttt{d}}$ (Procedure 18) work. □

Theorem A.4.2.3.7 For any time t and a node n, $rank(n, t)$ can only be `DUMMY_RANK` or the rank of an enqueuer that's attached to the subtree rooted at n.

Proof This is trivial considering how `refreshNode`$_e$, `refreshNode`$_d$ and `refreshLeaf`$_e$, `refreshLeaf`$_d$ works. □

Theorem A.4.2.3.8 If an enqueue or a dequeue `op` begins its **timestamp-refresh phase** at t_0 and finishes at time t_1, there's always at least one successful call to `refreshTimestamp`$_e$ (Procedure 14) or `refreshTimestamp`$_d$ (Procedure 19) that affects the enqueuer node corresponding to $rank($`op`$)$ and this successful call starts and ends its **CAS-sequence** between t_0 and t_1.

Proof Suppose the interested **timestamp-refresh phase** affects the enqueuer node n.

Notice that the **timestamp-refresh phase** of both enqueue and dequeue consists of at most 2 `refreshTimestamp` calls affecting n.

If one of the two `refreshTimestamps` of the **timestamp-refresh phase** succeeds, then the theorem obviously holds.

Consider the case where both fail.

The first `refreshTimestamp` fails because there's another `refreshTimestamp` on n ending its **CAS-sequence** successfully after t_0 but before the end of the first `refreshTimestamp`'s **CAS-sequence**.

The second `refreshTimestamp` fails because there's another `refreshTimestamp` on n ending its **CAS-sequence** successfully after t_0 but before the end of the second `refreshTimestamp`'s **CAS-sequence**. This another `refreshTimestamp` must start its **CAS-sequence** after the end of the first successful `refreshTimestamp`, otherwise, it would overlap with the **CAS-sequence** of the first successful `refreshTimestamp`, but successful **CAS-sequences** on the same enqueuer node cannot overlap as ABA problem does not occur. In other words, this another `refreshTimestamp` starts and successfully ends its **CAS-sequence** between t_0 and t_1.

We have proved the theorem. □

Theorem A.4.2.3.9 If an enqueue or a dequeue begins its **node-n-refresh phase** at t_0 and finishes at t_1, there's always at least one successful `refreshNode` or `refreshLeaf` calls affecting n and this successful call starts and ends its **CAS-sequence** between t_0 and t_1.

Proof This is similar to the above proof. □

Theorem A.4.2.3.10 Consider a node n. If within t_0 and t_1, any dequeue d where $n \in path(d)$ has finished its **node-n-refresh phase**, then $min\text{-}ts(rank(n, t_x), t_y)$ is monotonically decreasing for $t_x, t_y \in [t_0, t_1]$.

Proof We have the assumption that within t_0 and t_1, all dequeue where $n \in path(d)$ has finished its **node-n-refresh phase**. Notice that if n satisfies this assumption, any child of n also satisfies this assumption.

We will prove a stronger version of this theorem: Given a node n, time t_0 and t_1 such that within $[t_0, t_1]$, any dequeue d where $n \in path(d)$ has finished its **node-n-refresh phase**. Consider the last dequeue's **node-n-refresh phase** before t_0 (there maybe none). Take $t_s(n)$ and $t_e(n)$ to be the starting and ending time of the CAS-sequence of the last successful **n-refresh call** during this phase, or if there is none, $t_s(n) = t_e(n) = 0$. Then, $min\text{-}ts(rank(n, t_x), t_y)$ is monotonically decreasing for $t_x, t_y \in [t_e(n), t_1]$.

Consider any enqueuer node of rank r that's attached to a satisfied leaf node. For any n' that is a descendant of n, during $t_s(n')$ and t_1, there's no call to `spsc_dequeue`. Because:

- If an `spsc_dequeue` starts between t_0 and t_1, the dequeue that calls it hasn't finished its **node-n'-refresh phase**.
- If an `spsc_dequeue` starts between $t_s(n')$ and t_0, then a dequeue's **node-n'-refresh phase** must start after $t_s(n')$ and before t_0, but this violates our assumption of $t_s(n')$.

Therefore, there can only be calls to `spsc_enqueue` during $t_s(n')$ and t_1. Thus, $min\text{-}spsc\text{-}ts(r, t_x)$ can only decrease from `MAX_TIMESTAMP` to some timestamp and remain constant for $t_x \in [t_s(n'), t_1]$. (1)

Similarly, there can be no dequeue that hasn't finished its **timestamp-refresh phase** during $t_s(n')$ and t_1. Therefore, $min\text{-}ts(r, t_x)$ can only decrease from `MAX_TIMESTAMP` to some timestamp and remain constant for $t_x \in [t_s(n'), t_1]$. (2)

Consider any satisfied leaf node n_0. There can't be any dequeue that hasn't finished its **node-n_0-refresh phase** during $t_e(n_0)$ and t_1. Therefore, any successful `refreshLeaf` affecting n_0 during $[t_e(n_0), t_1]$ must be called by an enqueue. Because there's no `spsc_dequeue`, this `refreshLeaf` can only set $rank(n_0, t_x)$ from `DUMMY_RANK` to r and this remains r until t_1, which is the rank of the enqueuer whose node it's attached to. Therefore, combining with (1), $min\text{-}ts(rank(n_0, t_x), t_y)$ is monotonically decreasing for $t_x, t_y \in [t_e(n_0), t_1]$. (3)

Consider any satisfied non-leaf node n' that is a descendant of n. Suppose during $[t_e(n'), t_1]$, we have a sequence of successful **n'-refresh calls** that start their CAS-sequences at $t_{start\text{-}0} < t_{start\text{-}1} < t_{start\text{-}2} < \ldots < t_{start\text{-}k}$ and end them at $t_{end\text{-}0} < t_{end\text{-}1} < t_{end\text{-}2} < \ldots < t_{end\text{-}k}$. By definition, $t_{end\text{-}0} = t_e(n')$ and $t_{start\text{-}0} = t_s(n')$. We can prove that $t_{end\text{-}i} < t_{start\text{-}(i+1)}$ because successful CAS-sequences cannot overlap.

Due to how `refreshNode` (Procedure 15 and Procedure 20) is defined, for any $k \geq i \geq 1$:

- Suppose $t_{rank\text{-}i}(c)$ is the time `refreshNode` reads the rank stored in the child node c, so $t_{start\text{-}i} \leq t_{rank\text{-}i}(c) \leq t_{end\text{-}i}$.
- Suppose $t_{ts\text{-}i}(c)$ is the time `refreshNode` reads the timestamp stored in the enqueuer with the rank read previously, so $t_{start\text{-}i} \leq t_{ts\text{-}i}(c) \leq t_{end\text{-}i}$.
- There exists a child c_i such that $rank(n', t_{end\text{-}i}) = rank(c_i, t_{rank\text{-}i}(c_i))$. (4)
- For every child c of n',
  $min\text{-}ts(rank(n', t_{end\text{-}i}), t_{ts\text{-}i}(c_i))$
  $\leq min\text{-}ts(rank(c, t_{rank\text{-}i}(c)), t_{ts\text{-}i}(c))$. (5)

Suppose the stronger theorem already holds for every child c of n'. (6)

For any $i \geq 1$, we have $t_e(c) \leq t_s(n') \leq t_{start\text{-}(i-1)} \leq t_{rank\text{-}(i-1)}(c) \leq t_{end\text{-}(i-1)} \leq t_{start\text{-}i} \leq t_{rank\text{-}i}(c) \leq t_1$. Combining with (5), (6), we have for any $k \geq i \geq 1$,
$min\text{-}ts(rank(n', t_{end\text{-}i}), t_{ts\text{-}i}(c_i))$
$\leq min\text{-}ts(rank(c, t_{rank\text{-}i}(c)), t_{ts\text{-}i}(c))$
$\leq min\text{-}ts\left(rank\left(c, t_{rank\text{-}(i-1)}(c)\right), t_{ts\text{-}i}(c)\right)$.

Choose $c = c_{i-1}$ as in (4). We have for any $k \geq i \geq 1$,
$min\text{-}ts(rank(n', t_{end\text{-}i}), t_{ts\text{-}i}(c_i))$
$\leq min\text{-}ts(rank\left(c_{i-1}, t_{rank\text{-}(i-1)}(c_{i-1})\right), t_{ts\text{-}i}(c_{i-1}))$
$= min\text{-}ts(rank\left(n', t_{end\text{-}(i-1)}\right), t_{ts\text{-}i}(c_{i-1})$.

Because $t_{ts\text{-}i}(c_i) \leq t_{end\text{-}i}$ and $t_{ts\text{-}i}(c_{i-1}) \geq t_{end\text{-}(i-1)}$ and (2), we have for any $k \geq i \geq 1$,
$min\text{-}ts(rank(n', t_{end\text{-}i}), t_{end\text{-}i})$
$\leq min\text{-}ts\left(rank\left(n', t_{end\text{-}(i-1)}\right), t_{end\text{-}(i-1)}\right)$. $(*)$

$rank(n', t_x)$ can only change after each successful `refreshNode`, therefore, the sequence of its value is $rank(n', t_{end\text{-}0})$, $rank(n', t_{end\text{-}1})$, ..., $rank(n', t_{end\text{-}k})$. $(**)$

Note that if `refreshNode` observes that an enqueuer has a `Min_timestamp` of `MAX_TIMESTAMP`, it would never try to CAS n''s rank to the rank of that enqueuer (Line 47 of Procedure 15 and Line 101 of Procedure 20). So, if `refreshNode` actually sets the rank of n' to some non-`DUMMY_RANK` value, the corresponding enqueuer must actually has a non-`MAX_TIMESTAMP` `Min-timestamp` *at some point*. Due to (2), this is constant up until t_1. Therefore, $min\text{-}ts(rank(n', t_{end\text{-}i}), t))$ is constant for any $t \geq t_{end\text{-}i}$ and $k \geq i \geq 1$. $min\text{-}ts(rank(n', t_{end\text{-}0}), t))$ is constant for any $t \geq t_{end\text{-}0}$ if there's a `refreshNode` before t_0. If there's no `refreshNode` before t_0, it is constantly `MAX_TIMESTAMP`. So, $min\text{-}ts(rank(n', t_{end\text{-}i}), t))$ is constant for any $t \geq t_{end\text{-}i}$ and $k \geq i \geq 0$. $(***)$

Combining $(*)$, $(**)$, $(***)$, we obtain the stronger version of the theorem. □

Theorem A.4.2.3.11 If an enqueue e obtains a timestamp c, finishes at time t_0 and is still **unmatched** at time t_1, then for any subrange T of $[t_0, t_1]$ that does not overlap with a dequeue, $min\text{-}ts(rank(root, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

Proof We will prove a stronger version of this theorem: Suppose an enqueue e obtains a timestamp c, finishes at time t_0 and is still **unmatched** at time t_1. For every $n_i \in path(e)$, n_0 is the leaf node and n_i is the parent of n_{i-1}, $i \geq 1$. If e starts and finishes its **node-n_i-refresh phase** at $t_{start\text{-}i}$ and $t_{end\text{-}i}$ then for any subrange T of $[t_{end\text{-}i}, t_1]$ that does not overlap with a dequeue d where $n_i \in path(d)$ and d hasn't finished its **node n_i refresh phase**, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

If $t_1 < t_0$ then the theorem holds.

Take r_e to be the rank of the enqueuer that performs e.

Suppose e enqueues an item with the timestamp c into the local SPSC at time $t_{enqueue}$. Because it's still unmatched up until t_1, c is always in the local SPSC during $t_{enqueue}$ to t_1. Therefore, $min\text{-}spsc\text{-}ts(r_e, t) \leq c$ for any $t \in [t_{enqueue}, t_1]$. (1)

Suppose e finishes its **timestamp refresh phase** at $t_{r\text{-}ts}$. Because $t_{r\text{-}ts} \geq t_{enqueue}$, due to (1), $min\text{-}ts(r_e, t) \leq c$ for every $t \in [t_{r\text{-}ts}, t_1]$. (2)

Consider the leaf node $n_0 \in path(e)$. Due to (2), $rank(n_0, t)$ is always r_e for any $t \in [t_{end\text{-}0}, t_1]$. Also due to (2), $min\text{-}ts(rank(n_0, t_r), t_s) \leq c$ for any $t_r, t_s \in [t_{end\text{-}0}, t_1]$.

Consider any non-leaf node $n_i \in path(e)$. We can extend any subrange T to the left until we either:

- Reach a dequeue d such that $n_i \in path(d)$ and d has just finished its **node-n_i-refresh phase**.
- Reach $t_{end\text{-}i}$.

Consider one such subrange T_i.

Notice that T_i always starts right after a **node-n_i-refresh phase**. Due to Theorem A.4.2.3.9, there's always at least one successful `refreshNode` in this **node-n_i-refresh phase**.

Suppose the stronger version of the theorem already holds for n_{i-1}. That is, if e starts and finishes its **node-n_{i-1}-refresh phase** at $t_{start\text{-}(i-1)}$ and $t_{end\text{-}(i-1)}$ then for any subrange T of $[t_{end\text{-}(i-1)}, t_1]$ that does not overlap with a dequeue d where $n_i \in path(d)$ and d hasn't finished its **node n_{i-1} refresh phase**, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

Extend T_i to the left until we either:

- Reach a dequeue d such that $n_i \in path(d)$ and d has just finished its **node-n_{i-1}-refresh phase**.
- Reach $t_{end\text{-}(i-1)}$.

Take the resulting range to be T_{i-1}. Obviously, $T_i \subseteq T_{i-1}$.

T_{i-1} satisifies both criteria:

- It's a subrange of $\left[t_{end\text{-}(i-1)}, t_1\right]$.
- It does not overlap with a dequeue d where $n_i \in path(d)$ and d hasn't finished its **node-n_{i-1}-refresh phase**.

Therefore, $min\text{-}ts(rank(n_{i-1}, t_r), t_s) \leq c$ for any $t_r, t_s \in T_{i-1}$.

Consider the last successful `refreshNode` on n_i ending not after T_i starts. Take $t_{s'}$ and $t_{e'}$ to be the start and end time of this `refreshNode`'s CAS-sequence. Because right at the start of T_i, a **node-n_i-refresh phase** just ends, this `refreshNode` must be within this **node-n_i-refresh phase**. (4)

This `refreshNode`'s CAS-sequence must be within T_{i-1}. This is because right at the start of T_{i-1}, a **node-n_{i-1}-refresh phase** just ends and $T_{i-1} \supseteq T_i$, T_{i-1} must cover the **node-n_i-refresh phase** whose end T_i starts from. Combining with (4), $t_{s'} \in T_{i-1}$ and $t_{e'} \in T_i$. (5)

Due to how `refreshNode` is defined and the fact that n_{i-1} is a child of n_i:

- t_{rank} is the time `refreshNode` reads the rank stored in n_{i-1}, so that $t_{s'} \leq t_{rank} \leq t_{e'}$. Combining with (5), $t_{rank} \in T_{i-1}$.
- t_{ts} is the time `refreshNode` reads the timestamp from that rank $t_{s'} \leq t_{ts} \leq t_{e'}$. Combining with (5), $t_{ts} \in T_{i-1}$.
- There exists a time t', $t_{s'} \leq t' \leq t_{e'}$, $min\text{-}ts(rank(n_i, t_{e'}), t') \leq min\text{-}ts(rank(n_{i-1}, t_{rank}), t_{ts})$. (6)

From (6) and the fact that $t_{rank} \in T_{i-1}$ and $t_{ts} \in T_{i-1}$, $min\text{-}ts(rank(n_i, t_{e'}), t') \leq c$.

There shall be no `spsc_dequeue` starting within $t_{s'}$ till the end of T_i because:

- If there's an `spsc_dequeue` starting within T_i, then T_i's assumption is violated.
- If there's an `spsc_dequeue` starting after $t_{s'}$ but before T_i, its dequeue must finish its **node-n_i-refresh phase** after $t_{s'}$ and before T_i. However, then $t_{e'}$ is no longer the end of the last successful `refreshNode` on n_i not after T_i.

Because there's no `spsc_dequeue` starting in this timespan, $min\text{-}ts(rank(n_i, t_{e'}), t_{e'}) \leq min\text{-}ts(rank(n_i, t_{e'}), t') \leq c$.

If there's no dequeue between $t_{e'}$ and the end of T_i whose **node-n_i-refresh phase** hasn't finished, then by Theorem A.4.2.3.10, $min\text{-}ts(rank(n_i, t_r), t_s)$ is monotonically decreasing for any t_r, t_s starting from $t_{e'}$ till the end of T_i. Therefore, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ for any $t_r, t_s \in T_i$.

Suppose there's a dequeue whose **node-n_i-refresh phase** is in progress some time between $t_{e'}$ and the end of T_i. By definition, this dequeue must finish it before T_i. Because $t_{e'}$ is the time of the last successful refresh on n_i before T_i, $t_{e'}$ must be within the **node-n_i-refresh phase** of this dequeue and there should be no dequeue after that. By the way $t_{e'}$ is defined, technically, this dequeue has finished its **node-n_i-refresh phase** right at $t_{e'}$. Therefore, similarly, we can apply Theorem A.4.2.3.10, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ for any $t_r, t_s \in T_i$.

By induction, we have proved the stronger version of the theorem. Therefore, the theorem directly follows. □

Corollary A.4.2.3.12 Suppose *root* is the root tree node. If an enqueue e obtains a timestamp c, finishes at time t_0 and is still **unmatched** at time t_1, then for any subrange T of $[t_0, t_1]$ that does not overlap with a dequeue, $\textit{min-spsc-ts}(\textit{rank}(\textit{root}, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

Proof Call t_{start} and t_{end} to be the start and end time of T.

Applying Theorem A.4.2.3.11, we have that $\textit{min-ts}(\textit{rank}(\textit{root}, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

Fix t_r so that $\textit{rank}(\textit{root}, t_r) = r$. We have that $\textit{min-ts}(r, t) \leq c$ for any $t \in T$.

$\textit{min-ts}(r, t)$ can only change due to a successful `refreshTimestamp` on the enqueuer node with rank r. Consider the last successful `refreshTimestamp` on the enqueuer node with rank r not after T. Suppose that `refreshTimestamp` reads out the minimum timestamp of the local SPSC at $t' \leq t_{start}$.

Therefore, $\textit{min-ts}(r, t_{start}) = \textit{min-spsc-ts}(r, t') \leq c$.

We will prove that after t' until t_{end}, there's no `spsc_dequeue` on r running.

Suppose the contrary, then this `spsc_dequeue` must be part of a dequeue. By definition, this dequeue must start and end before t_{start}, else it violates the assumption of T. If this `spsc_dequeue` starts after t', then its `refreshTimestamp` must finish after t' and before t_{start}. But this violates the assumption that the last `refreshTimestamp` not after t_{start} reads out the minimum timestamp at t'.

Therefore, there's no `spsc_dequeue` on r running during $[t', t_{end}]$. Therefore, $\textit{min-spsc-ts}(r, t)$ remains constant during $[t', t_{end}]$ because it's not `MAX_TIMESTAMP`.

In conclusion, $\textit{min-spsc-ts}(r, t) \leq c$ for $t \in [t', t_{end}]$.

We have proved the theorem. □

Theorem A.4.2.3.13 Given a rank r. If within $[t_0, t_1]$, there's no uncompleted enqueues on rank r and all matching dequeues for any completed enqueues on rank r has finished, then $\textit{rank}(n, t) \neq r$ for every node n and $t \in [t_0, t_1]$.

Proof If n doesn't lie on the path from root to the leaf node that's attached to the enqueuer node with rank r, the theorem obviously holds.

Due to Corollary A.4.2.3.4, there can only be one enqueue and one dequeue at a time at an enqueuer node with rank r. Therefore, there is a sequential ordering among the enqueues and a sequential ordering within the dequeues. Therefore, it's sensible to talk about the last enqueue before t_0 and the last matched dequeue d before t_0.

Since all of these dequeues and enqueues work on the same local SPSC and the SPSC is linearizable, d must match the last enqueue. After this dequeue d, the local SPSC is empty.

When d finishes its **timestamp-refresh phase** at $t_{ts} \leq t_0$, due to Theorem A.4.2.3.8, there's at least one successful `refreshTimestamp` call in this phase. Because the last enqueue has been matched, $\textit{min-ts}(r, t) =$ `MAX_TIMESTAMP` for any $t \in [t_{ts}, t_1]$.

Similarly, for a leaf node n_0, suppose d finishes its **node-n_0-refresh phase** at $t_{r\text{-}0} \geq t_{ts}$, then $rank(n_0, t) =$ `DUMMY_RANK` for any $t \in [t_{r\text{-}0}, t_1]$. (1)

For any non-leaf node $n_i \in path(d)$, when d finishes its **node-n_i-refresh phase** at $t_{r\text{-}i}$, there's at least one successful `refreshNode` call during this phase. Suppose this `refreshNode` call starts and ends at $t_{start\text{-}i}$ and $t_{end\text{-}i}$. Suppose $rank(n_{i-1}, t) \neq r$ for $t \in \left[t_{r\text{-}(i-1)}, t_1\right]$. By the way `refreshNode` is defined after this `refreshNode` call, n_i will store some rank other than r. Because of (1), after this up until t_1, r never has a chance to be visible to a `refreshNode` on node n_i during $[n_{i-1}, t]$. In other words, $rank(n_i, t) \neq r$ for $t \in [t_{r\text{-}i}, t_1]$.

By induction, we obtain the theorem. □

Theorem A.4.2.3.14 In dLTQueue, if an enqueue e precedes another dequeue d, then either:

- d isn't matched.
- d matches e.
- e matches d' and d' precedes d.
- d matches e' and e' precedes e.
- d matches e' and e' overlaps with e.

Proof If d doesn't match anything, the theorem holds. If d matches e, the theorem also holds. Suppose d matches e', $e' \neq e$.

If e matches d' and d' precedes d, the theorem also holds. Suppose e matches d' such that d precedes d' or is unmatched. (1)

Suppose e obtains a timestamp of c and e' obtains a timestamp of c'.

Because e precedes d and because an MPSC queue does not allow multiple dequeues, from the start of d at t_0 until after Line 65 of dequeue (Procedure 17) at t_1, e has finished and there's no dequeue running that has *actually performed spsc_dequeue*. Also by t_0 and t_1, e is still unmatched due to (1).

Applying Corollary A.4.2.3.12, $\textit{min-spsc-ts}(rank(root, t_x), t_y) \leq c$ for $t_x, t_y \in [t_0, t_1]$. Therefore, d reads out a rank r such that $\textit{min-spsc-ts}(r, t) \leq c$ for $t \in [t_0, t_1]$. Consequently, d dequeues out a value with a timestamp not greater than c. Because d matches e', $c' \leq c$. However, $e' \neq e$ so $c' < c$.

This means that e cannot precede e', because if so, $c < c'$.

Therefore, e' precedes e or overlaps with e. □

Theorem A.4.2.3.15 In dLTQueue, if d matches e, then either e precedes or overlaps with d.

Proof If d precedes e, none of the local SPSCs can contain an item with the timestamp of e. Therefore, d cannot return an item with a timestamp of e. Thus d cannot match e.

Therefore, e either precedes or overlaps with d. □

Theorem A.4.2.3.16 In dLTQueue, If a dequeue d precedes another enqueue e, then either:

- d isn't matched.
- d matches e' such that e' precedes or overlaps with e and $e' \neq e$.

Proof If d isn't matched, the theorem holds.

Suppose d matches e'. Applying Theorem A.4.2.3.15, e' must precede or overlap with d. In other words, d cannot precede e'.

If e precedes or is e', then d must precede e', which is contradictory.

Therefore, e' must precede e or overlap with e. □

Theorem A.4.2.3.17 In dLTQueue, if an enqueue e_0 precedes another enqueue e_1, then either:

- Both e_0 and e_1 aren't matched.
- e_0 is matched but e_1 is not matched.
- e_0 matches d_0 and e_1 matches d_1 such that d_0 precedes d_1.

Proof If both e_0 and e_1 aren't matched, the theorem holds.

Suppose e_1 matches d_1. By Theorem A.4.2.3.15, either e_1 precedes or overlaps with d_1.

If e_0 precedes d_1, applying Theorem A.4.2.3.14 for d_1 and e_0:

- d_1 isn't matched, contradictory.
- d_1 matches e_0, contradictory.
- e_0 matches d_0 and d_0 precedes d_1, the theorem holds.
- d_1 matches e_1 and e_1 precedes e_0, contradictory.
- d_1 matches e_1 and e_1 overlaps with e_0, contradictory.

If d_1 precedes e_0, applying Theorem A.4.2.3.16 for d_1 and e_0:

- d_1 isn't matched, contradictory.
- d_1 matches e_1 and e_1 precedes or overlaps with e_0, contradictory.

Consider that d_1 overlaps with e_0, then d_1 must also overlap with e_1. Call r_1 the rank of the enqueuer that performs e_1. Call t to be the time d_1 atomically reads the root's rank on Line 21 of dequeue (Procedure 17). Because d_1 matches e_1, d_1 must read out r_1 at t_1.

If e_1 is the first enqueue of rank r_1, then t must be after e_1 has started, because otherwise, due to Theorem A.4.2.3.13, r_1 would not be in *root* before e_1.

If e_1 is not the first enqueue of rank r_1, then t must also be after e_1 has started. Suppose the contrary, t is before e_1 has started:

- If there's no uncompleted enqueue of rank r_1 at t and they are all matched by the time t, due to Theorem A.4.2.3.13, r_1 would not be in *root* at t. Therefore, d_1 cannot read out r_1, which is contradictory.
- If there's some unmatched enqueue of rank r_1 at t, d_1 will match one of these enqueues instead because:
  - There's only one dequeue at a time, so unmatched enqueues at t remain unmatched until d_1 performs an `spsc_dequeue`.
  - Due to Corollary A.4.2.3.4, all the enqueues of rank r_1 must finish before another starts. Therefore, there's some unmatched enqueue of rank r_1 finishing before e_1.
  - The local SPSC of the enqueuer node of rank r_1 is serializable, so $d_1$will favor one of these enqueues over e_1.

Therefore, t must happen after e_1 has started. Right at t, no dequeue is actually modifying the dLTQueue state and e_0 has finished. If e_0 has been matched at t then the theorem holds. If e_0 hasn't been matched at t, applying Theorem A.4.2.3.11, d_1 will favor e_0 over e_1, which is a contradiction.

We have proved the theorem. □

Theorem A.4.2.3.18 In dLTQueue, if a dequeue d_0 precedes another dequeue d_1, then either:

- d_0 isn't matched.
- d_1 isn't matched.
- d_0 matches e_0 and d_1 matches e_1 such that e_0 precedes or overlaps with e_1.

Proof If d_0 isn't matched or d_1 isn't matched, the theorem holds.

Suppose d_0 matches e_0 and d_1 matches e_1.

Suppose the contrary, e_1 precedes e_0. Applying Theorem A.4.2.3.14:

- Both e_0 and e_1 aren't matched, which is contradictory.
- e_1 is matched but e_0 is not matched, which contradictory.
- e_1 matches d_1 and e_0 matches d_0 such that d_1 precedes d_0, which is contradictory.

Therefore, the theorem holds. □

Theorem A.4.2.3.19 *(Linearizability of dLTQueue)* The dLTQueue algorithm is linearizable.

Proof Suppose some history H produced from the modified dLTQueue algorithm.

If H contains some pending method calls, we can just wait for them to complete (because the algorithm is wait-free, which we will prove later). Therefore, now we consider all H to contain only completed method calls. So, we know that if a dequeue or an enqueue in H is matched or not.

If there are some unmatched enqueues, we can append dequeues sequentially to the end of H until there's no unmatched enqueues. Consider one such H'.

We already have a strict partial order $\to_{H'}$ on H'.

Because the queue is MPSC, there's already a total order among the dequeues.

We will extend $\to_{H'}$ to a strict total order $\Rightarrow_{H'}$ on H' as follows:
- If $X \to_{H'} Y$ then $X \Rightarrow_{H'} Y$. (1)
- If a dequeue d matches e then $e \Rightarrow_{H'} d$. (2)
- If a dequeue d_0 matches e_0 and another dequeue d_1 matches e_1 such that $d_0 \Rightarrow_{H'} d_1$ then $e_0 \Rightarrow_{H'} e_1$. (3)
- If a dequeue d overlaps with an enqueue e but does not match e, $d \Rightarrow_{H'} e$. (4)

We will prove that $\Rightarrow_{H'}$ is a strict total order on H'. That is, for every pair of different method calls X and Y, either exactly one of these is true $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ and for any X, $X \not\Rightarrow_{H'} X$.

It's obvious that $X \not\Rightarrow_{H'} X$.

If X and Y are dequeues, because there's a total order among the dequeues, either exactly one of these is true: $X \to_{H'} Y$ or $Y \to_{H'} X$. Then due to (1), either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$. Notice that we cannot obtain $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ from (2), (3), or (4). Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(*)$

If X is a dequeue and Y is a enqueue, in this case (3) cannot help us obtain either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$, so we can disregard it.
- If $X \to_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y, so (4) cannot apply. Applying Theorem A.4.2.3.16, either
  - X isn't matched, (2) cannot apply. Therefore, $Y \not\Rightarrow_{H'} X$.
  - X matches e' and $e' \neq Y$. Therefore, X does not match Y, or (2) cannot apply. Therefore, $Y \not\Rightarrow_{H'} X$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ and $Y \not\Rightarrow_{H'} X$.
- If $Y \to_{H'} X$, then due to (1), $Y \Rightarrow_{H'} X$. By definition, Y precedes X, so (4) cannot apply. Even if (2) applies, it can only help us obtain $Y \Rightarrow_{H'} X$.
  Therefore, in this case, $Y \Rightarrow_{H'} X$ and $X \not\Rightarrow_{H'} Y$.
- If X overlaps with Y:
  - If X matches Y, then due to (2), $Y \Rightarrow_{H'} X$. Because X matches Y, (4) cannot apply. Therefore, in this case $Y \Rightarrow_{H'} X$ but $X \not\Rightarrow_{H'} Y$.
  - If X does not match Y, then due to (4), $X \Rightarrow_{H'} Y$. Because X doesn't match Y, (2) cannot apply. Therefore, in this case $X \Rightarrow_{H'} Y$ but $Y \not\Rightarrow_{H'} X$.

Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(**)$

If X is an enqueue and Y is an enqueue, in this case (2) and (4) are irrelevant:

- If $X \rightarrow_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y. Applying Theorem A.4.2.3.17,
  - Both X and Y aren't matched, then (3) cannot apply. Therefore, in this case, $Y \not\Rightarrow_{H'} X$.
  - X is matched but Y is not matched, then (3) cannot apply. Therefore, in this case, $Y \not\Rightarrow_{H'} X$.
  - X matches d_x and Y matches d_y such that d_x precedes d_y, then (3) applies and we obtain $X \Rightarrow_{H'} Y$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ but $Y \not\Rightarrow_{H'} X$.
- If $Y \rightarrow_{H'} X$, this case is symmetric to the first case. We obtain $Y \Rightarrow_{H'} X$ but $X \not\Rightarrow_{H'} Y$.
- If X overlaps with Y, because in H', all enqueues are matched, then, X matches d_x and d_y. Because d_x either precedes or succeeds d_y, Applying (3), we obtain either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ and there's no way to obtain the other.

Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(* * *)$

From $(*)$, $(* *)$, $(* * *)$, we have proved that $\Rightarrow_{H'}$ is a strict total ordering that is consistent with $\rightarrow_{H'}$. In other words, we can order method calls in H' in a sequential manner. We will prove that this sequential order is consistent with FIFO semantics:

- An enqueue can only be matched by one dequeue: This follows from Theorem A.4.2.3.1.
- A dequeue can only be matched by one enqueue: This follows from Theorem A.4.2.3.2.
- The order of item dequeues is the same as the order of item enqueues: Suppose there are two enqueues e_1, e_2 such that $e_1 \Rightarrow_{H'} e_2$ and suppose they match d_1 and d_2. Then we have obtained $e_1 \Rightarrow_{H'} e_2$ either because:
  - (3) applies, in this case $d_1 \Rightarrow_{H'} d_2$ is a condition for it to apply.
  - (1) applies, then e_1 precedes e_2, by Theorem A.4.2.3.17, d_1 must precede d_2, thus $d_1 \Rightarrow_{H'} d_2$.

  Therefore, if $e_1 \Rightarrow_{H'} e_2$ then $d_1 \Rightarrow_{H'} d_2$.
- An enqueue can only be matched by a later dequeue: Suppose there is an enqueue e matched by d. By (2), obviously $e \Rightarrow_{H'} d$.
  - If the queue is empty, dequeues return `false`. Suppose a dequeue d such that any $e \Rightarrow_{H'} d$ is all matched by some d' and $d' \Rightarrow_{H'} d$, we will prove that d is unmatched. By Theorem A.4.2.3.15, d can only match an enqueue e_0 that precedes or overlaps with d.
    - If e_0 precedes d, by our assumption, it's already matched by another dequeue.
    - If e_0 overlaps with d, by our assumption, $d \Rightarrow_{H'} e_0$ because if $e_0 \Rightarrow_{H'} d$, e_0 is already matched by another d'. Then, we can only obtain this because (4) applies, but then d does not match e_0.

  Therefore, d is unmatched.
- A dequeue returns `false` when the queue is empty: To put more precisely, for a dequeue d, if every successful enqueue e' such that $e' \Rightarrow_{H'} d$ has been matched by d' such that $d' \Rightarrow_{H'} d$, then d would be unmatched and return `false`. Suppose the contrary, d matches e. By definition, $e \Rightarrow_{H'} d$. This is a contradiction by our assumption.
- A dequeue returns `true` and matches an enqueue when the queue is not empty: To put more precisely, for a dequeue d, if there exists a successful enqueue e' such that $e' \Rightarrow_{H'} d$ and has not been matched by a dequeue d' such that $d' \Rightarrow_{H'} e'$, then d would be match some e and return `true`. This follows from Theorem A.4.2.3.11.
- An enqueue that returns `true` will be matched if there are enough dequeues after that: Based on how Procedure 12 is defined, when an enqueue returns `true`, it has successfully execute `spsc_enqueue`. By Theorem A.4.2.3.11, at some point, it would eventually be matched.
- An enqueue that returns `false` will never be matched: Based on how Procedure 12 is defined, when an enqueue returns `false`, the state of dLTQueue is not changed, except for the distributed counter. Therefore, it could never be matched.

In conclusion, $\Rightarrow_{H'}$ is a way we can order method calls in H' sequentially that conforms to FIFO semantics. Therefore, we can also order method calls in H sequentially that conforms to FIFO semantics as we only append dequeues sequentially to the end of H to obtain H'.

We have proved the theorem. □

A.4.3 Progress guarantee

Notice that every loop in dLTQueue is bounded, and no method have to wait for another. Therefore, dLTQueue is wait-free.

A.4.4 Theoretical performance

A summary of the theoretical performance of dLTQueue is provided in Table 7, which is already shown in Table 4. In the following discussion, R means remote operations and L means local operations.

Operations	Time-complexity
`enqueue`	$6 \log_2(n) R + 4 \log_2(n) L$
`dequeue`	$4 \log_2(n) R + 6 \log_2(n) L$

Table 7: Theoretical performance summary of dLTQueue. R means remote operations and L means local operations.

For `enqueue`, we consider the procedure Procedure 12. We consider the propagation process, which causes most of the remote operations, while Line 14 and Line 15 are negligible. Notice that the number of node refreshes are proportional to the number of the level of the trees, which is $O(n)$ for n being the number of processes. Each level of the tree in the worst case needs 2 retries, each retry would have to:

- Read the current node (which is a truly remote operation for `enqueue`).
- Read the two child nodes (which is 2 truly remote operations for `enqueue`).
- Read the two `min-timestamp` variables in the two child nodes (which is 2 truly local operations for `enqueue`).
- Compare-and-swap the current node (which is a truly remote opoeration for `enqueue`).

In total, each level requires 6 remote operations and 4 local operations. Therefore, `enqueue` requires about $6 \log_2(n) R + 4 \log_2(n) L$ operations.

For `dequeue`, it's similar to `enqueue` but the other way around, what makes for a remote operation in `enqueue` is a local operation in `dequeue` and otherwise. Therefore, `dequeue` requires about $4 \log_2(n) R + 6 \log_2(n) L$ operations.

A.5 Theoretical proofs of Slotqueue

In this section, we provide proofs covering all of our interested theoretical aspects in Slotqueue.

A.5.1 Proof-specific notations

As a refresher, Figure 26 shows the structure of Slotqueue.

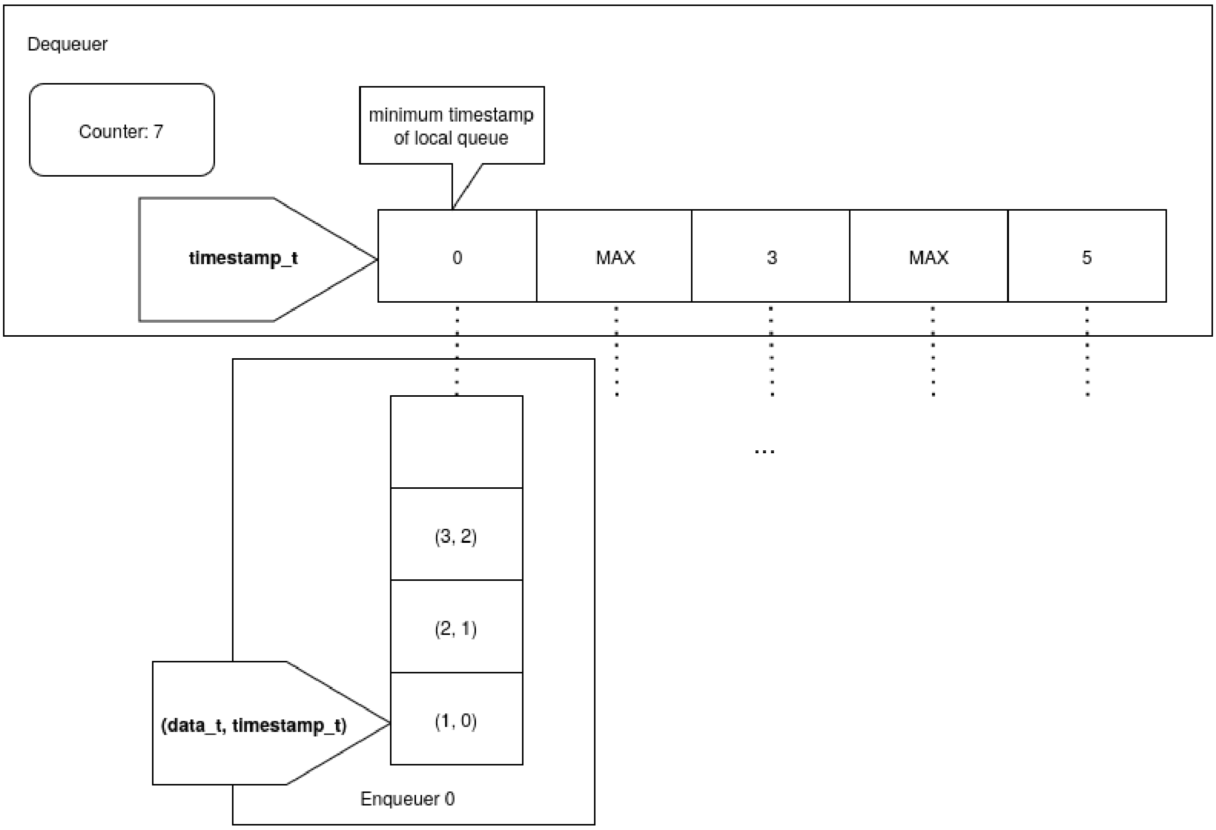


Figure 26: Basic structure of Slotqueue.

Each enqueuer hosts an SPSC that can only accessed by itself and the dequeuer. The dequeuer hosts an array of slots, each slot corresponds to an enqueuer, containing its SPSC's minimum timestamp.

We apply some domain knowledge of Slotqueue algorithm to the definitions introduced in Section A.2.2.

Definition A.5.1.1 A **CAS-sequence** on a slot `s` of an enqueue that affects `s` is the sequence of instructions from Line 15 to Line 20 of its `refreshEnqueue` (Procedure 25).

Definition A.5.1.2 A **slot-modification instruction** on a slot `s` of an enqueue that affects `s` is Line 20 of `refreshEnqueue` (Procedure 25).

Definition A.5.1.3 A **CAS-sequence** on a slot `s` of a dequeue that affects `s` is the sequence of instructions from Line 50 to Line 54 of its `refreshDequeue` (Procedure 28).

Definition A.5.1.4 A **slot-modification instruction** on a slot `s` of a dequeue that affects `s` is Line 54 of `refreshDequeue` (Procedure 28).

Definition A.5.1.5 A **CAS-sequence** of a dequeue/enqueue is said to **observe a slot value of s_0** if it loads s_0 at Line 15 of `refreshEnqueue` or Line 50 of `refreshDequeue`.

The followings are some other definitions that will be used throughout our proof.

Definition A.5.1.6 For an enqueue or dequeue op, $rank(op)$ is the rank of the enqueuer whose local SPSC is affected by op.

Definition A.5.1.7 For an enqueuer whose rank is r, the value stored in its corresponding slot at time t is denoted as $slot(r, t)$.

Definition A.5.1.8 For an enqueuer with rank r, the minimum timestamp among the elements between `First` and `Last` in its local SPSC at time t is denoted as $min\text{-}spsc\text{-}ts(r, t)$.

Definition A.5.1.9 For an enqueue, **slot-refresh phase** refer to its execution of Line 5 - Line 6 of Procedure 24.

Definition A.5.1.10 For a dequeue, **slot-refresh phase** refer to its execution of Line 28 - Line 29 of Procedure 26.

Definition A.5.1.11 For a dequeue, **slot-scan phase** refer to its execution of Line 31 - Line 45 of Procedure 27.

A.5.2 Correctness

This section establishes the correctness of Slotqueue introduced in Section 4.4.

A.5.2.1 ABA problem

Noticeably, we use no scheme to avoid ABA problem in Slotqueue. In actuality, ABA problem does not adversely affect our algorithm's correctness, except in the extreme case that the 64-bit distributed counter overflows, which is unlikely.

We will prove that Slotqueue is ABA-safe, as introduced in Section A.2.2.

Notice that we only use `CAS`es on:

- Line 20 of `refreshEnqueue` (Procedure 25), which is part of an enqueue.
- Line 54 of `refreshDequeue` (Procedure 28), which is part of a dequeue.

Both `CAS`es target some slot in the `Slots` array.

Theorem A.5.2.1.1 *(Concurrent accesses on an SPSC and a slot)* Only one dequeuer and one enqueuer can concurrently modify an SPSC and a slot in the `Slots` array.

Proof This is trivial to prove based on the algorithm's definition. □

Theorem A.5.2.1.2 *(Monotonicity of SPSC timestamps)* Each SPSC in Slotqueue contains elements with increasing timestamps.

Proof Each enqueue would `FAA` the distributed counter (Line 3 in Procedure 24) and enqueue into the local SPSC an item with the timestamp obtained from the counter. Applying Theorem A.5.2.1.1, we know that items are enqueued one at a time into the SPSC. Therefore, later items are enqueued by later enqueues, which obtain increasing values by `FAA`-ing the shared counter. The theorem holds. □

Theorem A.5.2.1.3 A `refreshEnqueue` (Procedure 25) can only change a slot to a value other than `MAX_TIMESTAMP`.

Proof For `refreshEnqueue` to change the slot's value, the condition on Line 18 must be `false`. Then, `new_timestamp` must equal to `ts`, which is not `MAX_TIMESTAMP`. It's obvious that the `CAS` on Line 20 changes the slot to a value other than `MAX_TIMESTAMP`. □

Theorem A.5.2.1.4 *(ABA safety of dequeue)* Assume that the 64-bit distributed counter never overflows, dequeue (Procedure 26) is ABA-safe.

Proof Consider a **successful CAS-sequence** on slot `s` by a dequeue d. Denote t_d as the value this CAS-sequence observes.

If there's no **successful slot-modification instruction** on slot `s` by an enqueue e within d's **successful CAS-sequence**, then this dequeue is ABA-safe.

Suppose the enqueue e executes the *last* **successful slot-modification instruction** on slot `s` within d's **successful CAS-sequence**. Denote t_e to be the value that e sets `s` $(*)$.

If $t_e \neq t_d$, this CAS-sequence of d cannot be successful, which is a contradiction. Therefore, $t_e = t_d$.

Note that e can only set `s` to the timestamp of the item it enqueues. That means, e must have enqueued a value with timestamp t_d. However, by definition $(*)$, t_d is read before e executes the CAS, so d cannot observe t_d because e has CAS-ed slot `s`. This means another process (dequeuer/enqueuer) has seen the value e enqueued and CAS `s` for e before t_d. By Theorem A.5.2.1.1, this "another process" must be another dequeuer d' that precedes d because it overlaps with e.

Because d' and d cannot overlap, while e overlaps with both d' and d, e must be the *first* enqueue on `s` that overlaps with d. Combining with Theorem A.5.2.1.1 and the fact that e executes the *last* **successful slot-modification instruction** on slot `s` within d's **successful CAS-sequence**, e must be the only enqueue that executes a **successful slot-modification instruction** on `s` within d's **successful CAS-sequence**.

During the start of d's successful CAS-sequence till the end of e, `spsc_readFront` on the local SPSC must return the same element, because:

- There's no other dequeue running during this time.
- There's no enqueue other than e running.
- The `spsc_enqueue` of e must have completed before the start of d's successful CAS sequence, because a previous dequeuer d' can see its effect.

Therefore, if we were to move the starting time of d's successful CAS-sequence right after e has ended, we still retain the output of the program because:

- The CAS sequence only reads two shared values: the `rank`th entry of `Slots` and `spsc_readFront()`, but we have proven that these two values remain the same if we were to move the starting time of d's successful CAS-sequence this way.
- The CAS sequence does not modify any values except for the last CAS instruction, and the ending time of the CAS sequence is still the same.
- The CAS sequence modifies the `rank`th entry of `Slots` at the CAS but the target value is the same because inputs and shared values are the same in both cases.

We have proved that if we move d's successful CAS-sequence to start after the *last* **successful slot-modification instruction** on slot `s` within d's **successful CAS-sequence**, we still retain the program's output.

If we apply the reordering for every dequeue, the theorem directly follows. □

Theorem A.5.2.1.5 *(ABA safety of enqueue)* Assume that the 64-bit distributed counter never overflows, enqueue (Procedure 24) is ABA-safe.

Proof Consider a **successful CAS-sequence** on slot `s` by an enqueue e. Denote t_e as the value this CAS-sequence observes.

If there's no **successful slot-modification instruction** on slot `s` by a dequeue d within e's **successful CAS-sequence**, then this enqueue is ABA-safe.

Suppose the dequeue d executes the *last* **successful slot-modification instruction** on slot `s` within e's **successful CAS-sequence**. Denote t_d to be the value that d sets `s`. If $t_d \neq t_e$, this CAS-sequence of e cannot be successful, which is a contradiction $(*)$.

Therefore, $t_d = t_e$.

If $t_d = t_e =$ `MAX_TIMESTAMP`, this means e observes a value of `MAX_TIMESTAMP` before d even sets `s` to `MAX_TIMESTAMP` due to $(*)$. If this `MAX_TIMESTAMP` value is the initialized value of `s`, it's a contradiction, as `s` must be non-`MAX_TIMESTAMP` at some point for a dequeue such as d to enter its CAS sequence. If this `MAX_TIMESTAMP` value is set by an enqueue, it's also a contradiction, as `refreshEnqueue` cannot set a slot to `MAX_TIMESTAMP`. Therefore, this `MAX_TIMESTAMP` value is set by a dequeue d'. If $d' \neq d$ then it's a contradiction, because between d' and d, `s` must be set to be a non-`MAX_TIMESTAMP` value before d can be run, thus, e cannot have observed a value set by d'. Therefore, $d' = d$. But, this means e observes a value set by d, which violates our assumption $(*)$.

Therefore $t_d = t_e = t' \neq$ `MAX_TIMESTAMP`. e cannot observe the value t' set by d due to our assumption $(*)$. Suppose e observes the value t' from `s` set by another enqueue/dequeue call other than d.

If this "another call" is a dequeue d' other than d, d' precedes d. By Theorem A.5.2.1.2, after each dequeue, the front element's timestamp will be increasing, therefore, d' must have set `s` to a timestamp smaller than t_d. However, e observes $t_e = t_d$. This is a contradiction.

Therefore, this "another call" is an enqueue e' other than e and e' precedes e. We know that an enqueue only sets `s` to the timestamp it obtains.

Suppose e' does not overlap with d, then e precedes d. e' can only set `s` to t' if e' sees that the local SPSC has the front element as the element it enqueues. Due to Theorem A.5.2.1.1, this means e' must observe a local SPSC with only the element it enqueues. Then, when d executes `readFront`, the item e' enqueues must have been dequeued out already, thus, d cannot set `s` to t'. This is a contradiction.

Therefore, e' overlaps with d.

Because e' and e cannot overlap, while d overlaps with both e' and e, d must be the *first* dequeue on `s` that overlaps with e. Combining with Theorem A.5.2.1.1 and the fact that d executes the *last* **successful slot-modification instruction** on slot `s` within e's **successful CAS-sequence**, d must be the only dequeue that executes a **successful slot-modification instruction** within e's **successful CAS-sequence**.

During the start of e's successful CAS-sequence till the end of d, `spsc_readFront` on the local SPSC must return the same element, because:

- There's no other enqueue running during this time.
- There's no dequeue other than d running.
- The `spsc_dequeue` of d must have completed before the start of e's successful CAS sequence, because a previous enqueuer e' can see its effect.

Therefore, if we were to move the starting time of e's successful CAS-sequence right after d has ended, we still retain the output of the program because:

- The CAS sequence only reads two shared values: the `rank`th entry of `Slots` and `spsc_readFront()`, but we have proven that these two values remain the same if we were to move the starting time of e's successful CAS-sequence this way.
- The CAS sequence does not modify any values except for the last CAS/store instruction, and the ending time of the CAS sequence is still the same.
- The CAS sequence modifies the `rank`th entry of `Slots` at the CAS but the target value is the same because inputs and shared values are the same in both cases.

We have proved that if we move e's successful CAS-sequence to start after the *last* **successful slot-modification instruction** on slot `s` within e's **successful CAS-sequence**, we still retain the program's output.

If we apply the reordering for every enqueue, the theorem directly follows. □

Theorem A.5.2.1.6 *(ABA safety)* Assume that the 64-bit distributed counter never overflows, Slot-queue is ABA-safe.

Proof This follows from Theorem A.5.2.1.5 and Theorem A.5.2.1.4. □

A.5.2.2 Memory reclamation

Notice that Slotqueue pushes the memory reclamation problem to the underlying SPSC. If the underlying SPSC is memory-safe, Slotqueue is also memory-safe.

A.5.2.3 Linearizability

Theorem A.5.2.3.7 In Slotqueue, an enqueue can only match at most one dequeue.

Proof A dequeue indirectly performs a value dequeue through `spsc_dequeue`. Because `spsc_dequeue` can only match one `spsc_enqueue` by another enqueue, the theorem holds. □

Theorem A.5.2.3.8 In Slotqueue, a dequeue can only match at most one enqueue.

Proof This is trivial as a dequeue can only read out at most one value, so it can only match at most one enqueue. □

Theorem A.5.2.3.9 If an enqueue e begins its **slot-refresh phase** at time t_0 and finishes at time t_1, there's always at least one successful `refreshEnqueue` that either doesn't execute its **CAS sequence** or starts and ends its **CAS-sequence** between t_0 and t_1 or a successful `refreshDequeue` on $rank(e)$ starting and ending its **CAS-sequence** between t_0 and t_1.

Proof If one of the two `refreshEnqueues` succeeds, then the theorem obviously holds.

Consider the case where both fail.

The first `refreshEnqueue` fails because it tries to execute its **CAS-sequence** but there's another `refreshDequeue` executing its **slot-modification instruction** successfully after t_0 but before the end of the first `refreshEnqueue`'s **CAS-sequence**.

The second `refreshEnqueue` fails because it tries to execute its **CAS-sequence** but there's another `refreshDequeue` executing its **slot-modification instruction** successfully after t_0 but before the end of the second `refreshEnqueue`'s **CAS-sequence**. This another `refreshDequeue` must start its **CAS-sequence** after the end of the first successful `refreshDequeue`, due to Theorem A.5.2.1.1. In other words, this another `refreshDequeue` starts and successfully ends its **CAS-sequence** between t_0 and t_1.

We have proved the theorem. □

Theorem A.5.2.3.10 If a dequeue d begins its **slot-refresh phase** at time t_0 and finishes at time t_1, there's always at least one successful `refreshEnqueue` or `refreshDequeue` on $rank(d)$ starting and ending its **CAS-sequence** between t_0 and t_1.

Proof This is similar to the above theorem. □

Theorem A.5.2.3.11 Given a rank r, if an enqueue e on r that obtains the timestamp c completes at t_0 and is still unmatched by t_1, then $slot(r, t) \leq c$ for any $t \in [t_0, t_1]$.

Proof Take t' to be the time e's `spsc_enqueue` takes effect.

At some point after t', e must enter its **slot-refresh phase**. By Theorem A.5.2.3.9, there must be a successful refresh call after t'. If this refresh call executes a **CAS-sequence** at $t'' \geq t'$, $t'' \in [t', t_0]$, this **CAS-sequence** must observe the effect of `spsc_enqueue`. Therefore, $slot(r, t'') \leq c$. If this refresh call doesn't execute a **CAS-sequence**, it must be a `refreshEnqueue` seeing that the front timestamp is different from the enqueued timestamp at t'', $t'' \in [t', t_0]$. Because e is unmatched up until t_1 and due to Theorem A.5.2.1.2, $slot(r, t'') \leq c$.

By the same reasoning as in Theorem A.5.2.1.6, any successful slot-modification instructions happening after t'' must observe the effect of e's `spsc_enqueue`. However, because e is never matched between t'' and t_1, the timestamp c is in the local SPSC the whole timespan $[t'', t_1]$. Therefore, any slot-modification instructions during $[t'', t_1]$ must set the slot's value to some value not greater than c. □

Theorem A.5.2.3.12 In Slotqueue, if an enqueue e precedes another dequeue d, then either:

- d isn't matched.
- d matches e.
- e matches d' and d' precedes d.
- d matches e' and e' precedes e.
- d matches e' and e' overlaps with e.

Proof If d doesn't match anything, the theorem holds. If d matches e, the theorem also holds. Suppose d matches e', $e' \neq e$.

If e matches d' and d' precedes d, the theorem also holds. Suppose e matches d' such that d precedes d' or is unmatched. (1)

Suppose e obtains a timestamp of c and e' obtains a timestamp of c'.

Due to (1), at the time d starts, e has finished but it is still unmatched. By the way Procedure 27 is defined and by Theorem A.5.2.3.11, d would find a slot that stores a timestamp that is not greater than the one e enqueues. In other word, $c' \leq c$. But $c' \neq c$, then $c' < c$. Therefore, e cannot precede e', otherwise, $c < c'$.

So, either e' precedes or overlaps with e. The theorem holds. □

Theorem A.5.2.3.13 In Slotqueue, if d matches e, then either e precedes or overlaps with d.

Proof If d precedes e, none of the local SPSCs can contain an item with the timestamp of e. Therefore, d cannot return an item with a timestamp of e. Thus d cannot match e.

Therefore, e either precedes or overlaps with d. □

Theorem A.5.2.3.14 In Slotqueue, if a dequeue d precedes another enqueue e, then either:

- d isn't matched.
- d matches e' such that e' precedes or overlaps with e and $e' \neq e$.

Proof If d isn't matched, the theorem holds.

Suppose d matches e'. By Theorem A.5.2.3.13, either e' precedes or overlaps with d. Therefore, $e' \neq e$. Furthermore, e cannot precede e', because then d would precede e'.

We have proved the theorem. □

Theorem A.5.2.3.15 If an enqueue e_0 precedes another enqueue e_1, then either:

- Both e_0 and e_1 aren't matched.
- e_0 is matched but e_1 is not matched.
- e_0 matches d_0 and e_1 matches d_1 such that d_0 precedes d_1.

Proof If e_1 is not matched, the theorem holds.

Suppose e_1 matches d_1. By Theorem A.5.2.3.13, either e_1 precedes or overlaps with d_1.

Suppose the contrary, e_0 is unmatched or e_0 matches d_0 such that d_1 precedes d_0, then when d_1 starts, e_0 is still unmatched.

If e_0 and e_1 targets the same rank, it's obvious that d_1 must prioritize e_0 over e_1. Thus d_1 cannot match e_1.

If e_0 targets a later rank than e_1, d_1 cannot find e_1 in the first scan, because the scan is left-to-right, and if it finds e_1 it would later find e_0 that has a lower timestamp. Suppose d_1 finds e_1 in the second scan, that means d_1 finds $e' \neq e_1$ and e''s timestamp is larger than e_1's, which is larger than e_0's. Due to the scan being left-to-right, e' must target a later rank than e_1. If e' also targets a later rank than e_0, then in the second scan, d_1 would have prioritized e_0 that has a lower timestamp. Suppose e' targets an earlier rank than e_0 but later than e_1. Because e_0's timestamp is larger than e''s, it must precede or overlap with e. Similarlt, e_1 must precede or overlap with e. Because e' targets an earlier rank than e_0, e_0's **slot-refresh phase** must finish after e''s. That means e_1 must start after e''s **slot-refresh phase**, because e_0 precedes e_1. But then, e_1 must obtain a timestamp larger than e', which is a contradiction.

Suppose e_0 targets an earlier rank than e_1. If d_1 finds e_1 in the first scan, then in the second scan, d_1 would have prioritize e_0's timestamp. Suppose d_1 finds e_1 in the second scan and during the first scan, it finds $e' \neq e_1$ and e''s timestamp is larger than e_1's, which is larger than e_0's. Due to how the second scan is defined, e' targets a later rank than e_1, which targets a later rank than e_0. Because during the second scan, e_0 is not chosen, its **slot-refresh phase** must finish after e''s. Because e_0 precedes e_1, e_1 must start after e''s **slot-refresh phase**, so it must obtain a larger timestamp than e', which is a contradiction.

Therefore, by contradiction, e_0 must be matched and e_0 matches d_0 such that d_0 precedes d_1. □

Theorem A.5.2.3.16 In Slotqueue, if a dequeue d_0 precedes another dequeue d_1, then either:

- d_0 isn't matched.
- d_1 isn't matched.
- d_0 matches e_0 and d_1 matches e_1 such that e_0 precedes or overlaps with e_1.

Proof If either d_0 isn't matched or d_1 isn't matched, the theorem holds.

Suppose d_0 matches e_0 and d_1 matches e_1.

If e_1 precedes e_0, applying Theorem A.5.2.3.15, we have e_1 matches d_1 and e_0 matches d_0 such that d_1 precedes d_0. This is a contradiction.

Therefore, e_0 either precedes or overlaps with e_1. □

Theorem A.5.2.3.17 *(Linearizability of Slotqueue)* Slotqueue is linearizable.

Proof Suppose some history H produced from the Slot-queueu algorithm.

If H contains some pending method calls, we can just wait for them to complete (because the algorithm is wait-free, which we will prove later). Therefore, now we consider all H to contain only completed method calls. So, we know that if a dequeue or an enqueue in H is matched or not.

If there are some unmatched enqueues, we can append dequeues sequentially to the end of H until there's no unmatched enqueues. Consider one such H'.

We already have a strict partial order $\rightarrow_{H'}$ on H'.

Because the queue is MPSC, there's already a total order among the dequeues.

We will extend $\rightarrow_{H'}$ to a strict total order $\Rightarrow_{H'}$ on H' as follows:
- If $X \rightarrow_{H'} Y$ then $X \Rightarrow_{H'} Y$. (1)
- If a dequeue d matches e then $e \Rightarrow_{H'} d$. (2)
- If a dequeue d_0 matches e_0 and another dequeue d_1 matches e_1 such that $d_0 \Rightarrow_{H'} d_1$ then $e_0 \Rightarrow_{H'} e_1$. (3)
- If a dequeue d overlaps with an enqueue e but does not match e, $d \Rightarrow_{H'} e$. (4)

We will prove that $\Rightarrow_{H'}$ is a strict total order on H'. That is, for every pair of different method calls X and Y, either exactly one of these is true $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ and for any X, $X \not\Rightarrow_{H'} X$.

It's obvious that $X \not\Rightarrow_{H'} X$.

If X and Y are dequeues, because there's a total order among the dequeues, either exactly one of these is true: $X \rightarrow_{H'} Y$ or $Y \rightarrow_{H'} X$. Then due to (1), either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$. Notice that we cannot obtain $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ from (2), (3), or (4). Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(*)$

If X is a dequeue and Y is an enqueue, in this case (3) cannot help us obtain either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$, so we can disregard it.

- If $X \to_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y, so (4) cannot apply. Applying Theorem A.5.2.3.14, either
  - X isn't matched, (2) cannot apply. Therefore, $Y \not\Rightarrow_{H'} X$.
  - X matches e' and $e' \neq Y$. Therefore, X does not match Y, or (2) cannot apply. Therefore, $Y \not\Rightarrow_{H'} X$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ and $Y \not\Rightarrow_{H'} X$.
- If $Y \to_{H'} X$, then due to (1), $Y \Rightarrow_{H'} X$. By definition, Y precedes X, so (4) cannot apply. Even if (2) applies, it can only help us obtain $Y \Rightarrow_{H'} X$.
  Therefore, in this case, $Y \Rightarrow_{H'} X$ and $X \not\Rightarrow_{H'} Y$.
- If X overlaps with Y:
  - If X matches Y, then due to (2), $Y \Rightarrow_{H'} X$. Because X matches Y, (4) cannot apply. Therefore, in this case $Y \Rightarrow_{H'} X$ but $X \not\Rightarrow_{H'} Y$.
  - If X does not match Y, then due to (4), $X \Rightarrow_{H'} Y$. Because X doesn't match Y, (2) cannot apply. Therefore, in this case $X \Rightarrow_{H'} Y$ but $Y \not\Rightarrow_{H'} X$.

Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(* *)$

If X is an enqueue and Y is an enqueue, in this case (2) and (4) are irrelevant:

- If $X \to_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y. Applying Theorem A.5.2.3.15,
  - Both X and Y aren't matched, then (3) cannot apply. Therefore, in this case, $Y \not\Rightarrow_{H'} X$.
  - X is matched but Y is not matched, then (3) cannot apply. Therefore, in this case, $Y \not\Rightarrow_{H'} X$.
  - X matches d_x and Y matches d_y such that d_x precedes d_y, then (3) applies and we obtain $X \Rightarrow_{H'} Y$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ but $Y \not\Rightarrow_{H'} X$.
- If $Y \to_{H'} X$, this case is symmetric to the first case. We obtain $Y \Rightarrow_{H'} X$ but $X \not\Rightarrow_{H'} Y$.
- If X overlaps with Y, because in H', all enqueues are matched, then, X matches d_x and d_y. Because d_x either precedes or succeeds d_y, Applying (3), we obtain either $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ and there's no way to obtain the other.

Therefore, exactly one of $X \Rightarrow_{H'} Y$ or $Y \Rightarrow_{H'} X$ is true. $(* * *)$

From $(*)$, $(* *)$, $(* * *)$, we have proved that $\Rightarrow_{H'}$ is a strict total ordering that is consistent with $\to_{H'}$. In other words, we can order method calls in H' in a sequential manner. We will prove that this sequential order is consistent with FIFO semantics:

- An enqueue can only be matched by one dequeue: This follows from Theorem A.5.2.3.7.
- A dequeue can only be matched by one enqueue: This follows from Theorem A.5.2.3.8.
- The order of item dequeues is the same as the order of item enqueues: Suppose there are two enqueues e_1, e_2 such that $e_1 \Rightarrow_{H'} e_2$ and suppose they match d_1 and d_2. Then we have obtained $e_1 \Rightarrow_{H'} e_2$ either because:
  - (3) applies, in this case $d_1 \Rightarrow_{H'} d_2$ is a condition for it to apply.
  - (1) applies, then e_1 precedes e_2, by Theorem A.5.2.3.15, d_1 must precede d_2, thus $d_1 \Rightarrow_{H'} d_2$.

  Therefore, if $e_1 \Rightarrow_{H'} e_2$ then $d_1 \Rightarrow_{H'} d_2$.
- An enqueue can only be matched by a later dequeue: Suppose there is an enqueue e matched by d. By (2), obviously $e \Rightarrow_{H'} d$.
  - If the queue is empty, dequeues return `false`. Suppose a dequeue d such that any $e \Rightarrow_{H'} d$ is all matched by some d' and $d' \Rightarrow_{H'} d$, we will prove that d is unmatched. By Theorem A.5.2.3.13, d can only match an enqueue e_0 that precedes or overlaps with d.
    - If e_0 precedes d, by our assumption, it's already matched by another dequeue.
    - If e_0 overlaps with d, by our assumption, $d \Rightarrow_{H'} e_0$ because if $e_0 \Rightarrow_{H'} d$, e_0 is already matched by another d'. Then, we can only obtain this because (4) applies, but then d does not match e_0.

  Therefore, d is unmatched.
- A dequeue returns `false` when the queue is empty: To put more precisely, for a dequeue d, if every successful enqueue e' such that $e' \Rightarrow_{H'} d$ has been matched by d' such that $d' \Rightarrow_{H'} d$, then d would be unmatched and return `false`. Suppose the contrary, d matches e. By definition, $e \Rightarrow_{H'} d$. This is a contradiction by our assumption.
- A dequeue returns `true` and matches an enqueue when the queue is not empty: To put more precisely, for a dequeue d, if there exists a successful enqueue e' such that $e' \Rightarrow_{H'} d$ and has not been matched by a dequeue d' such that $d' \Rightarrow_{H'} e'$, then d would be match some e and return `true`. This follows from Theorem A.5.2.3.11.
- An enqueue that returns `true` will be matched if there are enough dequeues after that: Based on how Procedure 24 is defined, when an enqueue returns `true`, it has successfully execute `spsc_enqueue`. By Theorem A.5.2.3.11, at some point, it would eventually be matched.
- An enqueue that returns `false` will never be matched: Based on how Procedure 24 is defined, when an enqueue returns `false`, the state of Slotqueue is not changed, except for the distributed counter. Therefore, it could never be matched.

In conclusion, $\Rightarrow_{H'}$ is a way we can order method calls in H' sequentially that conforms to FIFO semantics. Therefore, we can also order method calls in H sequentially that conforms to FIFO semantics as we only append dequeues sequentially to the end of H to obtain H'.

We have proved the theorem. □

A.5.3 Progress guarantee

Notice that every loop in Slotqueue is bounded, and no method have to wait for another. Therefore, Slotqueue is wait-free.

A.5.4 Theoretical performance

A summary of the theoretical performance of Slotqueue is provided in Table 8, which is already shown in Table 4. By R, we mean remote operations and by L we mean local operations.

Operations	**Time-complexity**
`enqueue`	$4R + 3L$
`dequeue`	$3R + 2nL$

Table 8: Theoretical performance summary of Slotqueue. R means remote operations and L means local operations.

For `enqueue`, we consider Procedure 24. Line 3 causes 1 truly remote operation, as the distributed counter is hosted on the dequeuer. Line 4, as discussed in the theoretical performance of SPSC, causes $R + L$ operations. In the worst case, two `refreshEnqueue` calls are executed. We then consider each `refreshEnqueue` call. Line 10 causes $R + L$ operations. Most of the time, Line 15 - Line 20 are not executed. Therefore, the two `refreshEnqueue` calls cause at most $2R$ operations. So in total, $4R + 3L$ operations are required.

For `dequeue`, we consider Procedure 26. Line 21 causes most of the remote operations: The double scan of the `Slots` array causes about $2nL$ operations. We consider the truly remote operations. Line 25 causes $R + L$ operation. The double retry on Line 28 - Line 29 each causes L operations (Line 50) and R operations. So in total, $3R + 2nL$ operations are required.

References

[1] J. Schuchart, A. Bouteiller, and G. Bosilca, "Using MPI-3 RMA for Active Messages," 2019. doi: 10.1109/ExaMPI49596.2019.00011.

[2] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 3.1.* 2015. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf

[3] Thanh-Dang Diep, Phuong Hoai Ha, and Karl Fürlinger, "A general approach for supporting nonblocking data structures on distributed-memory systems," 2023. doi: https://doi.org/10.1016/j.jpdc.2022.11.006.

[4] B. Brock, A. Buluç, and K. Yelick, "BCL: A Cross-Platform Distributed Data Structures Library," 2019, *Association for Computing Machinery.* doi: 10.1145/3337821.3337912.

[5] John D. Valois, "Implementing Lock-Free Queues," 1994.

[6] L. Lamport, "Specifying Concurrent Program Modules," 1983, *Association for Computing Machinery.* doi: 10.1145/69624.357207.

[7] Mage M.Michael and Michael L.Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," 1996, *Association for Computing Machinery.* doi: 10.1145/248052.248106.

[8] P. Jayanti and S. Petrovic, "Logarithmic-time single deleter, multiple inserter wait-free queues and stacks," 2005, *Springer-Verlag.* doi: 10.1007/11590156_33.

[9] D. Adas and R. Friedman, "A Fast Wait-Free Multi-Producers Single-Consumer Queue," 2022, *Association for Computing Machinery.* doi: 10.1145/3491003.3491004.

[10] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann, 2012.

[11] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," 1990, *Association for Computing Machinery.* doi: 10.1145/78969.78972.

[12] M. Herlihy, "Wait-free synchronization," 1991, *Association for Computing Machinery.* doi: 10.1145/114005.102808.

[13] J. Dinan, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur, "An implementation and evaluation of the MPI 3.0 one-sided communication interface," 2016, *John Wiley and Sons Ltd.* doi: 10.1002/cpe.3758.

[14] J. Wang, Q. Jin, X. Fu, Y. Li, and P. Shi, "Accelerating Wait-Free Algorithms: Pragmatic Solutions on Cache-Coherent Multicore Architectures," 2019. doi: 10.1109/ACCESS.2019.2920781.

[15] Q. Yang, L. Tang, Y. Guo, N. Kuang, S. Zhong, and H. Luo, "WRLqueue: A Lock-Free Queue For Embedded Real-Time System," 2022. doi: 10.1109/HPCC-DSS-SmartCity-DependSys57074.2022.00197.