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: 0
MEMBER SECRETARY:
SUPERVISORS: THOẠI NAM
DIỆP THANH ĐĂNG

—ooo—

STUDENTS: ĐỖ NGUYỄN AN HUY - 2110193

HCMC, 04/2025

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 footnotes and 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 infrigment 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. Especially, I want to extend my gratitude towards my supervisors, Mr.Thoại Nam and Mr. Diệp Thanh Đăng, who have acted as constant counselors and advisors right from the project's inception throughout. They have provided guidance on the project's direction and laid the academic basis for this project, upon which my work is essentially built upon. Furthermore, they inspired me to work hard and push through all the technical obstacles. Without them, this project wouldn't have reached this point of creation.

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

List of Tables

List of Images

Chapter I Introduction

The demand for computation power has always been increasing relentlessly. Increasingly complex computation problems arise and accordingly more computation power is required to solve them. Much engineering efforts have 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 algorithms and data structures need to be devised.

Noticeably, multi-producer single-consumer (MPSC) is one of those data structures that are utilized heavily in distributed computing, forming the backbone of many applications. Consequently, an MPSC can easily present a performance bottleneck if not designed properly. A desirable distributed MPSC should be able to exploit the highly concurrent nature of distributed computing. Currently, in the literature, most distributed data structures are designed from the ground up, completely disregarding the any existing data structures developed in the shared memory area, e.g. [2]. This is partly due to the historical differences between the programming models utilized in these two areas. However, since the introduction of specialized networking hardware RDMA and the improved support of the remote memory access (RMA) programming model in MPI-3, this gap has been bridged. Thus, it has opened up a lot new research ([3]) on reusing the principles in the shared memory literature to distributed computing. One favorable characteristic of concurrent data structures that has been heavily researched in the shared memory literature, which is also equally important in distributed computing, is the property of non-blocking, or in particular, lock-freedom. Lock-freedom guarantees that if some processes suspend or die, other processes can still complete. This provides both progress guarantee and fault-tolerance, especially in distributed computing where nodes can fail any time. Thus, the rest of this document concerns itself with investigating and devising efficient non-blocking distributed MPSCs. Interestingly, we choose to adapt current MPSC algorithms in the shared-memory literature to distributed context, which enables a wealth of accumulated knowledge in this literature.

1.1 Motivation

Lock-free MPSC and other FIFO variants, such as multi-producer multi-consumer (MPMC), concurrent single-producer single-consumer (SPSC), are heavily studied in the shared memory literature, dating back from the 1980s-1990s [4], [5], [6] and more recently [7], [8]. It comes as no surprise that algorithms in this domain are highly developed and optimized for performance and scalability. However, most research about MPSC or FIFO algorithms in general completely disregard the available state-of-the-art algorithms in the shared memory literature. With the new RDMA networking hardware support and capabilities added to MPI-3 RMA API: lock-free shared-memory algorithms can 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 algorithms: Port and adapt potential lock-free shared-memory MSPCs to distributed context using the MPI-3 RMA programming model. If this approach proves to be effective, a huge intellectual reuse of shared-memory MSPC algorithms into the distributed domain is possible. Consequently, there may be no need to develop distributed MPSC algorithms from the ground up.

1.2 Objective

This thesis aims to:

- Investigate state-of-the-art shared-memory MPSCs.
- Select and appropriately modify potential MPSC algorithms so they can be implemented in popular distributed programming environments.
- Port MPSC algorithms using MPI-3 RMA.
- Evaluate various theoretical aspects of ported MPSC algorithms: Correctness, progress guarantee, time complexity analysis.
- Benchmark the ported MPSC algorithms and compare them with current distributed MPSCs in the literature.
- Discover distributed-environment-specific optimization opportunities for ported MPSC algorithms.

1.3 Scope

- For related works on shared-memory MPSCs, we only focus on linearizable MPSCs that support at least lock-free `enqueue` and `dequeue` operations.
- Any implementation details, benchmarking and optimizations assume MPI-3 settings.
- For optimizations, we focus on performance-related metrics, e.g. time-complexity (theoretically), throughput (empirically).

1.4 Structure

The rest of this report is structured as follows:

Chapter II discusses the theoretical foundation this thesis is based on and the technical terminology that's heavily utilized in this domain. As mentioned, this thesis investigates state-of-the-art shared-memory MPSCs. Therefore, we discuss the theory related to the design of concurrent algorithms such as lock-freedom and linearizability, the practical challenges such as the ABA problem and safe memory reclamation problem. We then explore the utilities offered by C++11 to implement concurrent algorithms and MPI-3 to port shared memory algorithms.

Chapter III surveys the shared-memory literature for state-of-the-art queue algorithms, specifically MPSC and SPSC algorithms (as SPSC can be modified to implement

MPSC). We specifically focus on algorithms that have the potential to be ported efficiently to distributed context, such as NUMA-aware or can be made to be NUMA-aware. We then conclude with a comparison of the most potential shared-memory queue algorithms.

Chapter IV documents distributed-versions of potential shared-memory MPSC algorithms surveys in Chapter III. It specifically presents our adaptation efforts of existing algorithms in the shared-memory literature to make their distributed implementations feasible.

Chapter V discusses various interesting theoretical aspects of our distributed MPSC algorithms in Chapter IV, specifically correctness (linearizability), progress guarantee (lock-freedom and wait-freedom), performance model. Our analysis of performance model helps back our empirical findings in Chapter VI, together, they work hand-in-hand to help us discover optimization opportunities.

Chapter VI introduces our benchmarking setup, including metrics, environments, benchmark/microbenchmark suites and conducting methods. We aim to demonstrate some preliminary results on how well ported shared-memory MPSCs can compare to existing distributed MPSCs. Finally, we discuss important factors that affect the runtime properties distributed MPSC algorithm, which have partly been explained by our theoretical analysis in Chapter V.

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

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 tp accessed 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 treatments to achieve high performance. One specific challenge is that this type of applications is hard to model in traditional MPI APIs. The introduction of MPI RMA (remote memory access) in MPI-2 and its improvement in MPI-3 has significantly improved MPI's capability to express irregular applications comfortably.

2.2 Multiple-producer, single-consumer (MPSC)

Multiple-producer, single-consumer (MPSC) 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 in parallel. Concurrent queues have many important applications, namely event handling, scheduling, etc. One class of concurrent FIFOs is MPSC, where one consumer may run in parallel with multiple producers. The reasons we're interested in MPSCs instead of the more general multiple-producer, multiple-consumer data structures (MPMCs) are that (1) high-performance and high-scalability MPSCs are much simpler to design than MPMCs while (2) MPSCs are noticeably as powerful as MPMCs - its consensus number equals the number of producers [9]. Thus, MPSCs can see as many use cases as MPMCs while being easily scalable and performant.

2.3 Progress guarantee

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. Lock-based 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 halt the whole program. 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 lock-based algorithms 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. These algorithms are said to be **blocking**. 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 to especially interesting subclasses of non-blocking algorithms. Unlike lock-based algorithms, they provide **progress guarantee**.

2.3.1 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. 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.

2.3.2 Wait-free algorithms

Wait-freedom is a stronger progress guarantee than lock-freedom. 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 bounded time. Wait-freedom is useful to have 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.

2.4 Correctness - Linearizability

Correctness of concurrent algorithms is hard to defined, especially when it comes to the semantics of concurrent data structures like MPSC. 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 $m_1, m_2, \ldots, m_n$.
- 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 $l_1 < l_2 < \ldots < l_n$, the effect of these n concurrent method calls $m_1, m_2, \ldots, m_n$ must be equivalent to calling $m_1, m_2, \ldots, m_n$ **sequentially**, one after the other in that order.

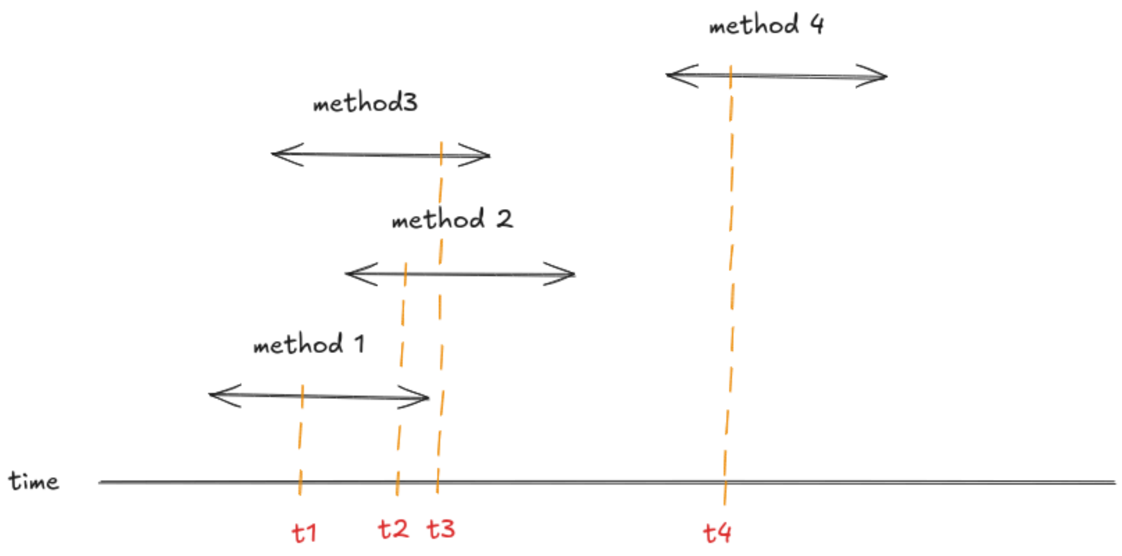

Figure 1: Linerization points of method 1, method 2, method 3, method 4 happens at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially

2.5 Common issues when designing lock-free algorithms

2.5.1 ABA problem

In implementing concurrent lock-free algorithms, hardware atomic instructions are utilized to achieve linearizability. The most popular atomic operation instruction is compare-and-swap (CAS). The reason for its popularity is (1) CAS is a **universal atomic instruction** - it has the **concensus number** of ∞ - which means it's the most powerful atomic instruction [11] (2) CAS is implemented in most hardware (3) some concurrent lock-free data structures such as MPSC can only be implemented using powerful atomic instructions such as CAS. The semantic 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. Concurrent algorithms often utilize CAS as follows:

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 `new value` because `old value` is no longer there, so its associated resources are not valid.

This scheme is susceptible to the notorious 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.

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` the memory location to the same value. Some notable schemes are **monotonic version tag** ([6]) and **hazard pointer** ([12]).

2.5.2 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. Solutions 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 push to the garbage collector. In non-garbage-collected programming environments, however, custom schemes must be utilized. Examples include using a reference counter to count the number of processes holding a reference to some memory and **hazard pointer** [12] to announce to other processes that some memory is not to be freed.

2.6 MPI-3

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 [1].

2.6.1 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 [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 acess 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.

2.6.2 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`) [1]. These remote communication operations only requires one side to specify.

2.6.3 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`.

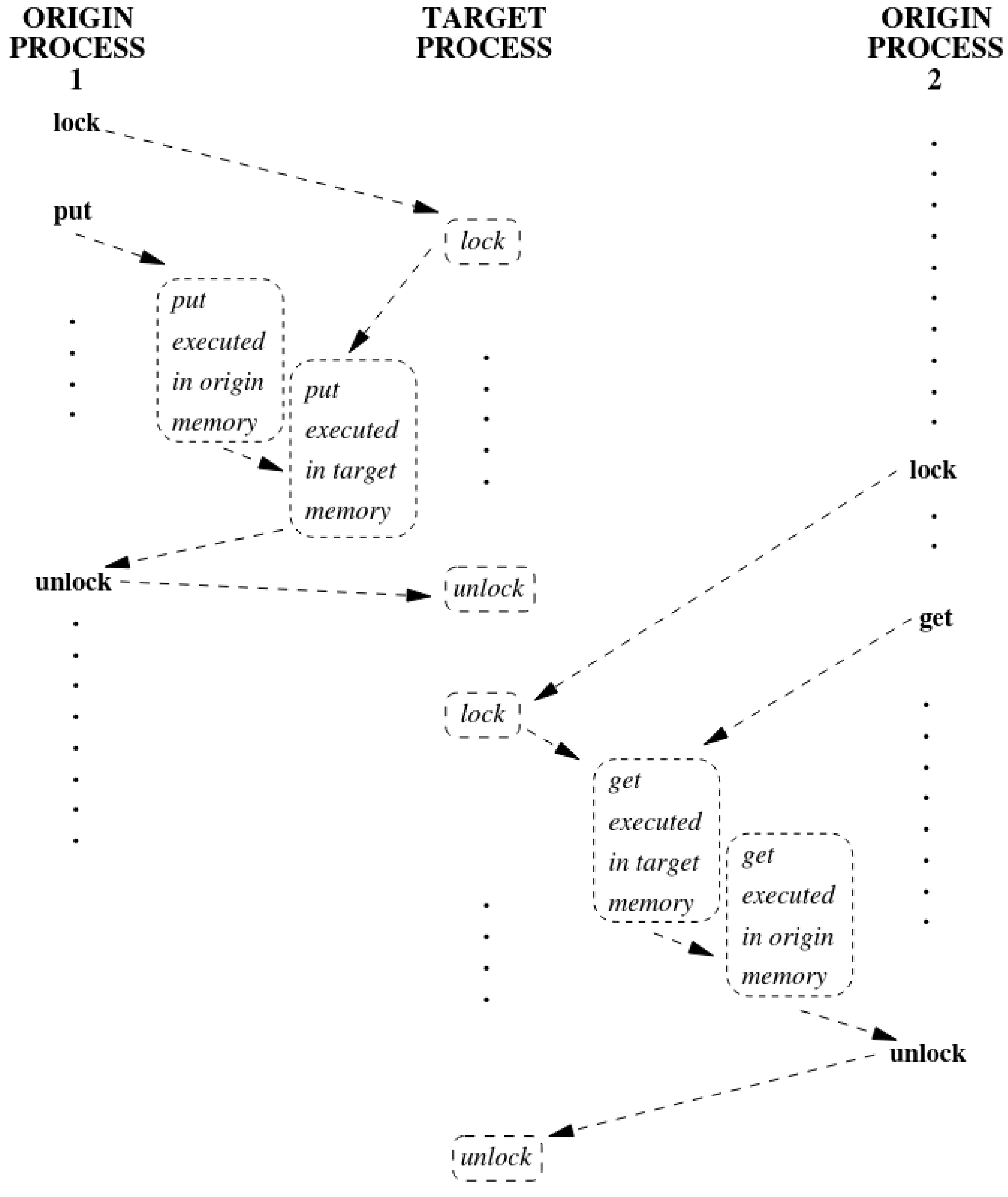

Figure 2: An illustration of passive target communication. Dashed arrows represent synchronization (source: [1])

2.7 Pure MPI approach of porting shared memory 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 [1], these functions are specified as follows:

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 MPSCs.
- 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 1: An example snippet showcasing our synchronization approach in MPI RMA

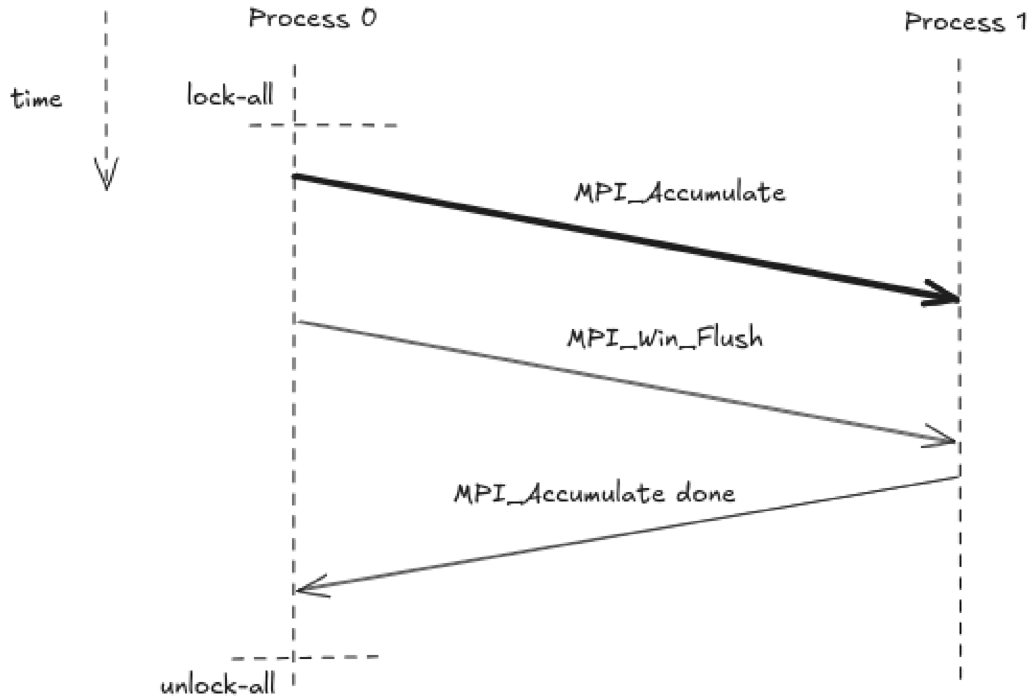

Figure 3: An illustration of our synchronization approach in MPI RMA

Chapter III Related works

There exists numerous research into the design of lock-free shared memory MPMCs and SPSCs. Interestingly, research into lock-free MPSCs are noticeably scarce. Although in principle, MPMCs and SPSCs can both be adapted for MPSCs use cases, specialized MPSCs can usually yield much more performance. In reality, we have only found 4 papers that are concerned with direct support of lock-free MPSCs: LTQueue [7], DQueue [9], WRLQueue [14] and Jiffy [8]. Table 2 summarizes the charateristics of these algorithms.

MPSCs	**LTQueue**	**DQueue**	**WR-LQueue**	**Jiffy**
ABA solution	Load-link/ Store-con-ditional	Incorrect custom scheme (*)	Custom scheme	Custom scheme
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	Un-bounded	Un-bounded	Un-bounded	Un-bounded

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

LTQueue [7] is the earliest wait-free shared memory MPSC to our knowledge. This algorithm is wait-free with $O(\log n)$ time complexity for both enqueues and dequeues, with n being the number of enqueuers. Their main idea is to split the MPSC among the enqueuers so that each enqueuer maintains a local SPSC data structure, which is only shared with the dequeuer. This improves the MPSC's scalability as multiple enqueues can complete the same time. The enqueuers shared a distributed counter and use it to label each item in their local SPSC with a specific timestamp. The timestamps are organized into nodes of a min-heap-like tree so that the dequeuer can look at the root of tree to determine which local SPSC to dequeue next. The min-heap property of the tree is preserved by a novel wait-free timestamp-refreshing operation. Memory reclamation becomes trivial as each MPSC entry is only shared by one enqueuer and one dequeuer in the local SPSC. The algorithm avoids ABA problem by utilizing load-link/store-conditional (LL/SC). This, on the other hand, presents a challenge in directly porting LTQueue as LL/SC is not widely available as the more popular CAS instruction.

DQueue [9] focuses on optimizing performance. It aims to be cache-friendly by having each enqueuer batches their updates in a local buffer to decrease cache misses. It also try to replace expensive atomic instructions such as CAS as many as possible. The MPSC is represented as a linked list of segments (which is an array). To enqueue, the enqueuer reserves a slot in the segment list and enqueues the value into the local buffer. If the local buffer is full, the enqueuer flushes the buffer and writes it onto every reserved slot in the segment list. The producer dequeues the values in the segment list in order, upon encountering a reserved but empty slot, it helps all enqueuers flush their local buffers. For memory reclamation, DQueue utilized a dedicated garbage collection thread that reclaims all fully dequeued segments. However, their algorithm is flawed and a segment maybe freed while some process is holding a reference to it.

WRLQueue [14] is a lock-free MPSC for embedded real-time system. Its main purpose is to avoid excessive modification of storage space. WRLQueue is simplfy a pair of buffer, one is worked on by multiple enqueuers and the other is work on by the dequeuer. The enqueuers batch their enqueues and write multiple elements onto the buffer once at a time. The dequeuer upon invocation will swap its buffer with the enqueuer's buffers to dequeue from it. However, this requires the dequeuer to wait for all enqueue operations to complete in their buffer. If an enqueue suspends or dies, the dequeuer will have to wait forever, this clearly violates the property of non-blocking.

Jiffy [8] is a fast and memory-efficient wait-free MPSC by avoiding excessive allocation of memory. Like DQueue, Jiffy represents the queue as a linked list of segments. Each enqueue reserves a slot in the segment, extends the linked-list as appropriately, writes the value into the slot and sets a per-slot flag to indicate that the slot is ready to be dequeued. To dequeue, the dequeuer repeatedly scan all the slots to find the first-ready-to-be-dequeue slot. Jiffy shows significant good memory usage and throughput compared to other previous state-of-the-art MPMCs.

Chapter IV Distributed MPSCs

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

- LTQueuev1 (Section 4.3) is a direct modification of LTQueue [7] without any usage of LL/SC.
- LTQueueV2 (Section 4.4) is inspired by the timestamp-refreshing idea of LTQueue [7] and repeated-rescan of Jiffy [8]. Although it still bears some resemblance to LTQueue, we believe it to be more optimized for distributed context.

MPSC	**LTQueueV1**	**LTQueueV2**
Correctness	Linearizable	Linearizable
Progress guarantee of dequeue	Wait-free	Wait-free
Progress guarantee of enqueue	Wait-free	Wait-free
Worst-case time complexity of dequeue	$O(\log n)$ R + $O(\log n)$ A	constant R + $O(n)$ A
Worst-case time complexity of enqueue	$O(\log n)$ R + $O(\log n)$ A	constant R + constant A
ABA solution	Unique timestamp	No harmful ABA problem
Memory reclamation	Custom scheme	Custom scheme
Number of elements	Unbounded	Unbounded

Table 3: Characteristic summary of our proposed distributed MPSCs. n is the number of enqueuers, R stands for **remote operation** and A stands for **atomic operation**

In this section, we present our proposed distributed MPSCs in detail. Any other discussions about theoretical aspects of these algorithms such as linearizability, progress guarantee, time complexity are deferred to Chapter V.

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 primitives in pseudocode

Although we use MPI-3 RMA to implement these algorithms, the algorithm specifications themselves are not inherently tied to MPI-3 RMA interfaces. 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 distributed SPSC

The two algorithms we propose here both utilize a 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 shared data in this SPSC is hosted on the enqueuer.

Types

`data_t` = The type of data stored

Shared variables

`First`: `remote<uint64_t>`

The index of the last undequeued entry. Hosted at the enqueuer.

`Last`: `remote<uint64_t>`

The index of the last unenqueued entry. Hosted at the enqueuer.

`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`
`First_buf = Last_buf = 0`

Dequeuer initialization

Initialize `Capacity`
`First_buf = Last_buf = 0`

The procedures of the enqueuer are given as follows.

Procedure 2: `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 3: `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 12-14). 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 4: `bool spsc_dequeue(data_t* output)`

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

`spsc_dequeue` first computes the new `First` value (line 15). If the queue is empty as indicating by the difference the new `First` value and `Last-buf` (line 16), 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 17). If the queue is still empty (line 18), we signal failure (line 19). Otherwise, we proceed to read the top value at `First_buf % Capacity` (line 20) into `output`, increment `First` (line 21) - effectively dequeue the element, update the value of `First_buf` (line 22) and signal success (line 23).

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

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

`spsc_readFront`$_d$ first checks if the SPSC is empty based on the difference between `First_buf` and `Last_buf` (line 24). If this check fails, we refresh `Last_buf` (line 25) and recheck (line 26). If the recheck fails, signal failure (line 27). If the SPSC is not empty, we read the queue entry at `First_buf % Capacity` into `output` (line 28) and signal success (line 29).

4.3 LTQueueV1 - Modified LTQueue without LL/SC

This algorithm presents our most straightforward effort to port LTQueue [7] 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 programming environments. We have to replace any usage of LL/SC in the original LTQueue algorithm. Compare-and-swap is unavoidable in distributed MPSCs, so we use the well-known monotonic timestamp scheme to guard against ABA problem.

4.3.1 Structure

The structure of our modified LTQueue is shown as in Image 1.

We differentiate between 2 types of nodes: **enqueuer nodes** (represented as the rectangular boxes at the bottom of Image 1) and normal **tree nodes** (represented as the circular boxes in Image 1).

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 Pseudocode

Below is the types utilized in LTQueueV1.

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

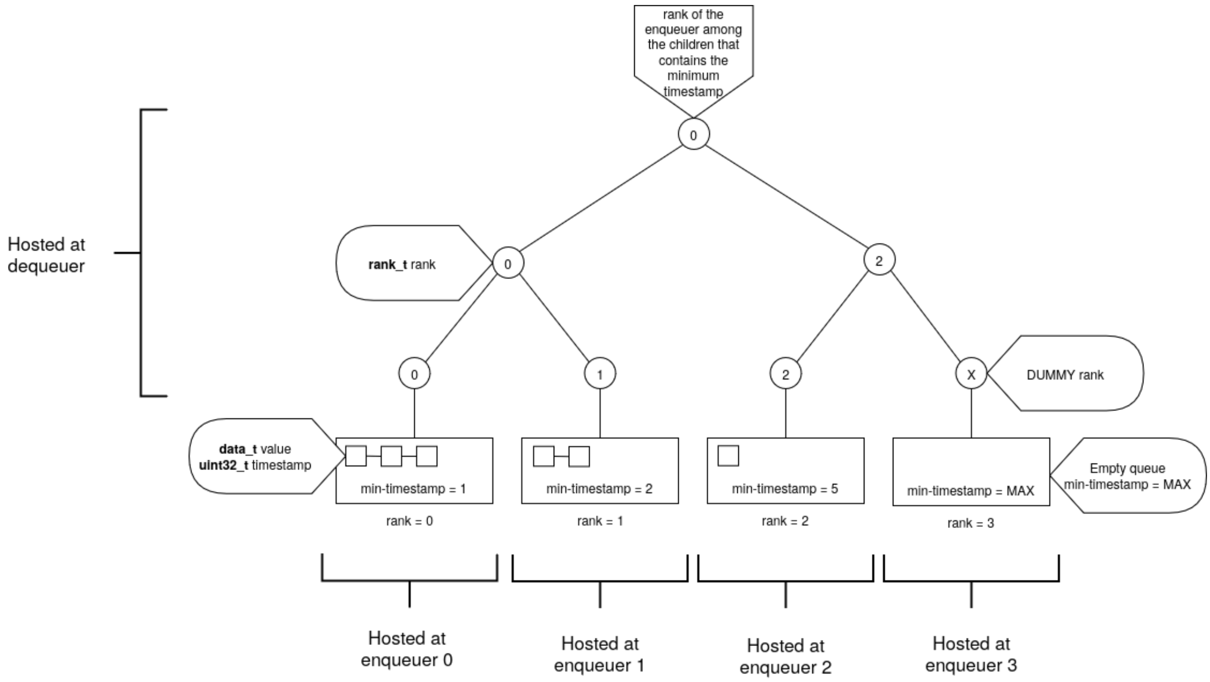

Image 1: LTQueueV1's structure

`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 followed.

Note that we have described a very specific and simple way to organize the tree nodes in LTQueue 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 7), `children()` (Procedure 8), `leafNodeIndex()` (Procedure 9) with this representation in mind. However, our algorithm doesn't strictly require this representation and can be subtituted 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 LTQueue.
>
> `Nodes`: `remote<node_t>`
>
> > An array with `Tree_size` entries storing all the tree nodes present in the LTQueue 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 6: `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.

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.

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

Procedure 7: `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 8: `vector<uint32_t> children(uint32_t index)`

```
left_child = index * 2 + 1
right_child = left_child + 1
res = vector<uint32_t>()
if (left_child >= Tree_size)
    return res
res.push(left_child)
if (right_child >= Tree_size)
    return res
res.push(right_child)
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 9: `uint32_t leafNodeIndex(uint32_t enqueuer_rank)`

```
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 Image 1.

The followings are the enqueuer procedures.

Procedure 10: `bool enqueue(data_t value)`

```
14 timestamp = fetch_and_add_sync(Counter, 1)
15 spsc_enqueue(&Spsc, (value, timestamp))
16 propagate_e()
```

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). Finally, `enqueue` propagates the changes by invoking propagate$_e$() (line 16).

Procedure 11: `void propagate_e()`

```
18 if (!refreshTimestamp_e())
19 | refreshTimestamp_e()
20 if (!refreshLeaf_e())
21 | refreshLeaf_e()
22 current_node_index = leafNodeIndex(Self_rank)
23 repeat
24 | current_node_index = parent(current_node_index)
25 | if (!refresh_e(current_node_index))
26 | | refresh_e(current_node_index)
27 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 18-19), refreshing of the enqueuer's leaf node (line 20-21), refreshing of internal nodes (line 23-27). On line 20-27, we refresh every tree node that lies between the enqueuer node and the root node.

Procedure 12: `bool refreshTimestamp_e()`

```
28 min_timestamp = timestamp_t {}
29 aread_sync(Min_timestamp, &min_timestamp)
30 {old-timestamp, old-version} = min_timestamp
31 front = (data_t {}, timestamp_t {})
32 is_empty = spsc_readFront(Spsc, &front)
33 if (is_empty)
34 | return compare_and_swap_sync(Min_timestamp,
   | timestamp_t {old-timestamp, old-version},
   | timestamp_t {MAX_TIMESTAMP, old-version + 1})
35 else
36 | 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 310 and CAS `Min_timestamp` accordingly (line 33-36).

Procedure 13: `bool refreshNode_e(uint32_t current_node_index)`

```
37 current_node = node_t {}
38 aread_sync(Nodes, current_node_index, &current_node)
39 {old-rank, old-version} = current_node.rank
40 min_rank = DUMMY_RANK
41 min_timestamp = MAX_TIMESTAMP
42 for child_node_index in children(current_node)
43     child_node = node_t {}
44     aread_sync(Nodes, child_node_index, &child_node)
45     {child_rank, child_version} = child_node
46     if (child_rank == DUMMY_RANK) continue
47     child_timestamp = timestamp_t {}
48     aread_sync(Timestamps[enqueuerOrder(child_rank)], &child_timestamp)
49     if (child_timestamp < min_timestamp)
50         min_timestamp = child_timestamp
51         min_rank = child_rank
52 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`$_e$ 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 42). For each child node, we read the rank stored in it (line 44), 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 48). 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 50-51). We then try to CAS the rank inside the current internal node to this rank.

Procedure 14: `bool refreshLeaf_e()`

```
leaf_node_index = leafNodeIndex(Self_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(Min_timestamp, &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 $\texttt{refreshLeaf}_e$ 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 58) and CAS the leaf node's rank accordingly (line 60).

The followings are the dequeuer procedures.

Procedure 15: `bool dequeue(data_t* output)`

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

To dequeue a value, `dequeue` reads the rank stored inside the root node (line 62). If the rank is `DUMMY_RANK`, the MPSC is treated as empty and failure is signaled (line 64). Otherwise, we invoke `spsc_dequeue` on the SPSC of the enqueuer with the obtained rank (line 66). We then extract out the real data and set it to `output` (line 68). We finally propagate the dequeue from the enqueuer node that corresponds to the obtained rank (line 69) and signal success (line 70).

Procedure 16: $\texttt{void propagate}_d\texttt{(uint32\_t enqueuer\_rank)}$

```
71 if (!refreshTimestamp_d(enqueuer_rank))
72     refreshTimestamp_d(enqueuer_rank)
73 if (!refreshLeaf_d(enqueuer_rank))
74     refreshLeaf_d(enqueuer_rank)
75 current_node_index = leafNodeIndex(enqueuer_rank)
76 repeat
77     current_node_index = parent(current_node_index)
78     if (!refresh_d(current_node_index))
79         refresh_d(current_node_index)
80 until current_node_index == 0
```

The $\texttt{propagate}_d$ procedure is similar to $\texttt{propagate}_e$, with appropriate changes to accommodate the dequeuer.

Procedure 17: `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 18: `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 19: `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 LTQueueV2 - Optimized LTQueue for distributed context

4.4.1 Motivation

Even though the straightforward LTQueue algorithm we have ported in Section 4.3 pretty much preserve the original algorithm's characteristics, that is wait-freedom and time complexity of $\Theta(\log n)$ for both `enqueue` and `dequeue` operations (which we will prove in Chapter V), we have to be aware that this is $\Theta(\log n)$ remote operations, which is potentially expensive and a bottleneck in the algorithm.

Therefore, to be more suitable for distributed context, we propose a new algorithm that's inspired by LTQueue, in which 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.

4.4.2 Structure

The structure of LTQueueV2 is shown as in Figure 4.

Each enqueuer hosts a distributed SPSC as in LTQueueV1 (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.

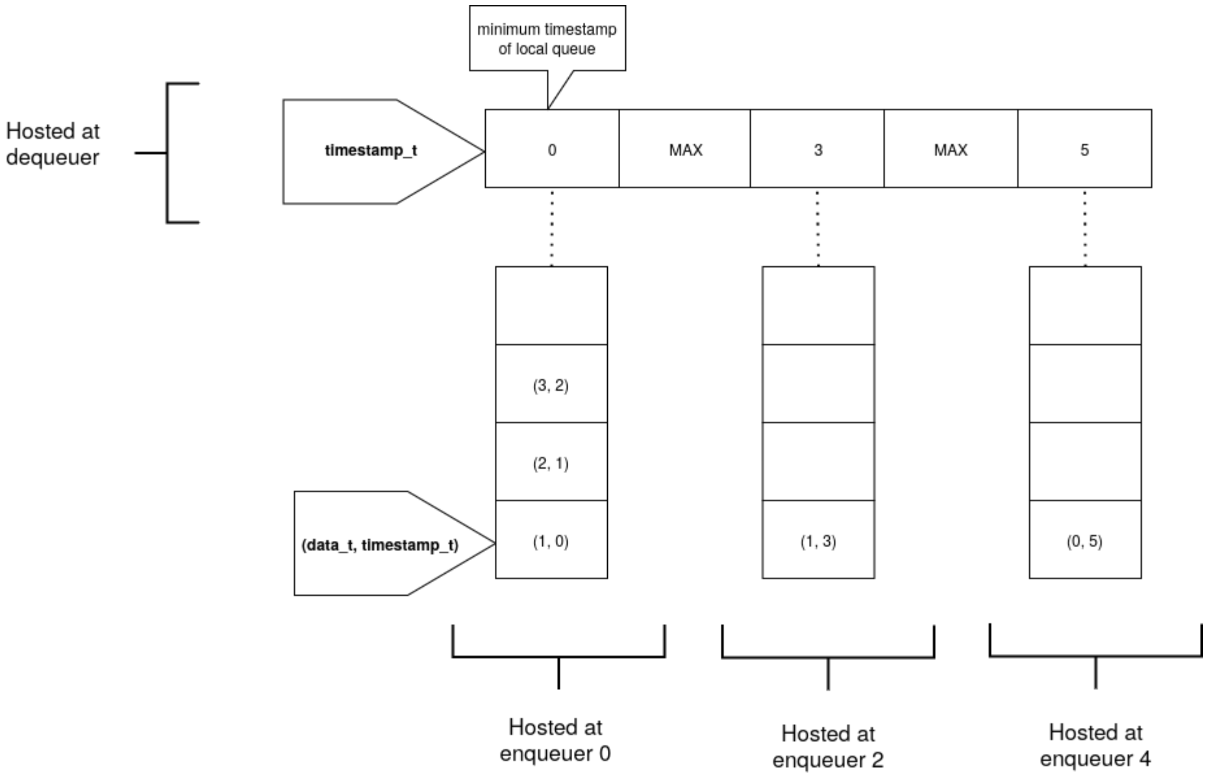

Figure 4: Basic structure of LTQueueV2

4.4.3 Pseudocode

We first introduce the types and shared variables utilized in LTQueueV2.

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.
>
> `Dequeuer_rank`: `uint32_t`
>
> > The rank of the dequeuer process. This is read-only.

Similar to the idea of assigning an order to each enqueuer in LTQueueV1, the following procedure computes an enqueuer's order based on its rank:

Procedure 20: `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 21: `uint64_t enqueuerRank(uint64_t enqueuer_order)`

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

Enqueuer-local variables

- `Dequeuer_rank`: `uint64_t`
- `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`
- `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.

The enqueuer operations are given as follows.

Procedure 22: `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 5-6, the enqueuer tries to refresh its slot's timestamp.

Procedure 23: `bool refreshEnqueue(timestamp_t ts)`

```
8  enqueuer_order = enqueueOrder(Self_rank)
9  old_timestamp = timestamp_t {}
10 aread_sync(&Slots, enqueuer_order, &old_timestamp)
11 front = (data_t {}, timestamp_t {})
12 success = spsc_readFront(Spsc, &front)
13 new_timestamp = success ? front.timestamp : MAX_TIMESTAMP
14 if (new_timestamp != ts)
15 |  return true
16 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 its slot's old timestamp (line 10) and the current front element in the SPSC (line 12). If the SPSC is empty, the new timestamp is set to `MAX_TIMESTAMP`, otherwise, the front element's timestamp (line 13). Note that `refreshEnqueue` immediately succeeds if the new timestamp is different from the timestamp `ts` of the element it enqueues (line 15). Otherwise, it tries to CAS its slot's timestamp with the new timestamp (line 16).

The dequeuer operations are given as follows.

Procedure 24: `bool dequeue(data_t* output)`

```
17 rank = readMinimumRank()
18 if (rank == DUMMY_RANK)
19 |  return false
20 output_with_timestamp = (data_t {}, timestamp_t {})
21 if (!spsc_dequeue(Spsc, &output_with_timestamp))
22 |  return false
23 *output = output_with_timestamp.data
24 if (!refreshDequeue(rank))
25 |  refreshDequeue(rank)
26 return true
```

To dequeue a value, `dequeue` first reads the rank of the enqueuer whose slot currently stores the minimum timestamp (line 17). If the obtained rank is `DUMMY_RANK`, failure is signaled (line 18-19). Otherwise, it tries to dequeue the SPSC of the corresponding enqueuer (line 21). It then tries to refresh the enqueuer's slot's timestamp to potentially notify the enqueuer of the dequeue (line 24-25). It then signals success (line 26).

Procedure 25: `uint64_t readMinimumRank()`

```
27 buffered_slots = timestamp_t[Enqueuer_count] {}
28 for index in 0..Enqueuer_count
29 |  aread_async(Slots, index, &bufferred_slots[index])
30 flush(Slots)
31 if every entry in bufferred_slots is MAX_TIMESTAMP
32 |  return DUMMY_RANK
33 for index in 0..Enqueuer_count
34 |  aread_async(Slots, index, &bufferred_slots[index])
35 flush(Slots)
36 rank = DUMMY_RANK
37 min_timestamp = MAX_TIMESTAMP
38 for index in 0..Enqueuer_count
39 |  timestamp = buffered_slots[index]
40 |  if (min_timestamp < timestamp)
41 |  |  rank = enqueuerRank(index)
42 |  |  min_timestamp = timestamp
43 return 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 27). It then performs 2 scans of `Slots` and read every entry into `buffered_slots` (line 28-33). If the first scan finds only `MAX_TIMESTAMP`s, `DUMMY_RANK` is returned (line 32). From there, based on `bufferred_slots`, it returns the rank of the enqueuer whose bufferred slot stores the minimum timestamp (line 38-43).

Procedure 26: `refreshDequeue(rank: int)` **returns** `bool`

```
44 enqueuer_order = enqueuerOrder(rank)
45 old_timestamp = timestamp_t {}
46 aread_sync(&Slots, enqueuer_order, &old_timestamp)
47 front = (data_t {}, timestamp_t {})
48 success = spsc_readFront(Spscs[enqueuer_order], &front)
49 new_timestamp = success ? front.timestamp : MAX_TIMESTAMP
50 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 46) and the front element (line 48). If the SPSC is empty, the new timestamp is set to `MAX_TIMESTAMP`, otherwise, it's the front element's timestamp (line 49). It finally tries to CAS the slot with the new timestamp (line 50).

Chapter V Theoretical aspects

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

5.1 Terminology

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

Definition 5.1.1 In an SPSC/MPSC, 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 5.1.2 In an SPSC/MPSC, an enqueue operation e is said to be **unmatched** if no dequeue operation **matches** it.

Definition 5.1.3 In an SPSC/MPSC, a dequeue operation d is said to be **unmatched** if no enqueue operation **matches** it, in other word, d returns `false`.

5.2 Formalization

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.

5.2.1 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 5.2.1.1 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 5.2.1.2 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 5.2.1.3 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 5.2.1.4 A **method call** is **pending** if its invocation event is $\perp$.

Definition 5.2.1.5 A **history** is a set of well-formed **method calls**.

Definition 5.2.1.6 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 5.2.1.7 $\rightarrow$ 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 5.2.1.8 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 5.2.1.9 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 5.2.1.10 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 5.2.1.11 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.

5.2.1.1 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 5.2.1.1.12 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.

5.2.1.2 Linearizable MPSC

An MPSC supports 2 methods:
- `enqueue` which accepts an input parameter and returns a boolean.
- `dequeue` which accepts an output parameter and returns a boolean.

Definition 5.2.1.2.13 An MPSC is **linearizable** if and only if any history produced from the MPSC 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.

5.2.2 ABA-safety

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

Definition 5.2.2.14 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 5.2.2.15 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 5.2.2.16 A **CAS-sequence** on a variable `v` is a sequence of instructions of a method m such that:
- The first instruction is a load $v_0 =$`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 5.2.2.17 A **successful CAS-sequence** on a variable v is a **CAS-sequence** on v that ends with a successful CAS.

Definition 5.2.2.18 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.

5.3 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.

5.3.1 Linearizability

We prove that our simple distributed SPSC is linearizable.

Theorem 5.3.1.1 *(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 2) that returns `false` is line 3.
- The linearization point of an `spsc_enqueue` call (Procedure 2) that returns `true` is line 7.
- The linearization point of an `spsc_dequeue` call (Procedure 4) that returns `false` is line 17.
- The linearization point of an `spsc_dequeue` call (Procedure 4) that returns `true` is line 21.
- The linearization point of `spsc_readFront`$_e$ call (Procedure 3) that returns `false` is line 10 or line 12 if line 10 is passed.
- The linearization point of `spsc_readFront`$_e$ call (Procedure 3) that returns `true` is line 12.
- The linearization point of `spsc_readFront`$_d$ call (Procedure 5) that returns `false` is line 25.
- The linearization point of `spsc_readFront`$_d$ call (Procedure 5) that returns `true` is right after line 25 (or right before line 28 if line 25 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 4 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 2 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 4 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 20 can see e's value. Therefore, d's linearization point at line 21 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 16 has been matched. Therefore, `First = Last` at line 16, or `First >= Last_buf`, therefore, the if condition at line 16-19 is entered. Also by the 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. Then, line 19 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 21 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 25-27 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 27 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 21 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 21 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 25-27 and execute line 28 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 5.2.1.1.12, our SPSC is linearizable. □

5.3.2 Progress guarantee

Our simple distributed SPSC is wait-free:

- `spsc_dequeue` (Procedure 4) does not execute any loops or wait for any other method calls.
- `spsc_enqueue` (Procedure 2) does not execute any loops or wait for any other method calls.
- `spsc_readFront`$_e$ (Procedure 3) does not execute any loops or wait for any other method calls.
- `spsc_readFront`$_d$ (Procedure 5) does not execute any loops or wait for any other method calls.

5.3.3 ABA problem

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

5.3.4 Memory reclamation

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

5.4 Theoretical proofs of LTQueueV1

5.4.1 Notation

The structure of LTQueueV1 is presented again in Image 2.

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**.

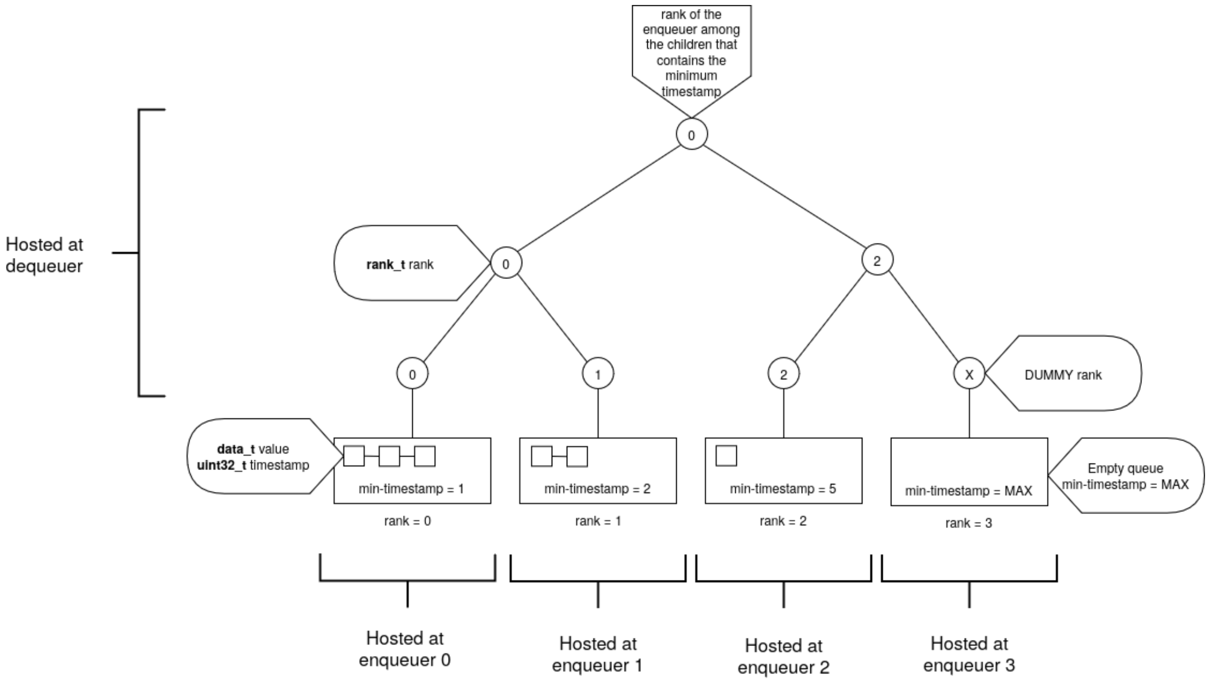

Image 2: LTQueueV1's structure

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

Definition 5.4.1.2 For an enqueue or a dequeue op, the rank of the enqueuer it affects is denoted as $rank(\texttt{op})$.

Definition 5.4.1.3 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 5.4.1.4 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 5.4.1.5 For an enqueue or a dequeue op, the set of nodes that it calls refreshNode (Procedure 13 or Procedure 18) or refreshLeaf (Procedure 14 or Procedure 19) on is denoted as $path(\texttt{op})$.

Definition 5.4.1.6 For an enqueue or a dequeue, **timestamp-refresh phase** refer to its execution of line 18-19 in propagate$_e$ (Procedure 11) or line 71-72 in propagate$_d$ (Procedure 16).

Definition 5.4.1.7 For an enqueue op, and a node $n \in path(\texttt{op})$, **node-n-refresh phase** refer to its execution of:

- Line 20-21 of `propagate`$_e$ (Procedure 11) if n is a leaf node.
- Line 25-26 of `propagate`$_e$ (Procedure 11) to refresh n's rank if n is a non-leaf node.

Definition 5.4.1.8 For a dequeue `op`, and a node $n \in path(\texttt{op})$, **node-n-refresh phase** refer to its execution of:

- Line 73-74 of `propagate`$_d$ (Procedure 16) if n is a leaf node.
- Line 78-79 of `propagate`$_d$ (Procedure 16) to refresh n's rank if n is a non-leaf node.

Definition 5.4.1.9 `refreshTimestamp`$_e$ (Procedure 12) is said to start its **CAS-sequence** if it finishes line 29. `refreshTimestamp`$_e$ is said to end its **CAS-sequence** if it finishes line 34 or line 36.

Definition 5.4.1.10 `refreshTimestamp`$_d$ (Procedure 17) is said to start its **CAS-sequence** if it finishes line 83. `refreshTimestamp`$_d$ is said to end its **CAS-sequence** if it finishes line 88 or line 90.

Definition 5.4.1.11 `refreshNode`$_e$ (Procedure 13) is said to start its **CAS-sequence** if it finishes line 38. `refreshNode`$_e$ is said to end its **CAS-sequence** if it finishes line 52.

Definition 5.4.1.12 `refreshNode`$_d$ (Procedure 18) is said to start its **CAS-sequence** if it finishes line 92. `refreshNode`$_d$ is said to end its **CAS-sequence** if it finishes line 106.

Definition 5.4.1.13 `refreshLeaf`$_e$ (Procedure 14) is said to start its **CAS-sequence** if it finishes line 55. `refreshLeaf`$_e$ is said to end its **CAS-sequence** if it finishes line 60.

Definition 5.4.1.14 `refreshLeaf`$_d$ (Procedure 19) is said to start its **CAS-sequence** if it finishes line 109. `refreshLeaf`$_d$ is said to end its **CAS-sequence** if it finishes line 114.

5.4.2 ABA problem

We use CAS instructions on:

- Line 34 and line 36 of `refreshTimestamp`$_e$ (Procedure 12).
- Line 52 of `refreshNode`$_e$ (Procedure 13).
- Line 60 of `refreshLeaf`$_e$ (Procedure 14).
- Line 88 and line 90 of `refreshTimestamp`$_d$ (Procedure 17).
- Line 106 of `refreshNode`$_d$ (Procedure 18).
- Line 114 of `refreshLeaf`$_e$ (Procedure 19).

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 LTQueueV1.

5.4.3 Linearizability

Theorem 5.4.3.1 In LTQueueV1, 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 5.4.3.2 In LTQueueV1, 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 5.4.3.3 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 5.4.3.4 Only one dequeue operation and one enqueue operation can operate concurrently on an enqueuer node.

Theorem 5.4.3.5 The SPSC at an enqueuer node contains items with increasing timestamps.

Proof Each enqueue would `FAA` the distributed counter (line 14 in Procedure 10) and enqueue into the SPSC an item with the timestamp obtained from that counter. Applying Corollary 5.4.3.4, 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 5.4.3.6 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 11) and $\texttt{propagate}_\texttt{d}$ (Procedure 16) work. □

Theorem 5.4.3.7 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 $\texttt{refreshNode}_\texttt{e}$, $\texttt{refreshNode}_\texttt{d}$ and $\texttt{refreshLeaf}_\texttt{e}$, $\texttt{refreshLeaf}_\texttt{d}$ works. □

Theorem 5.4.3.8 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 $\texttt{refreshTimestamp}_\texttt{e}$ (Procedure 12) or $\texttt{refreshTimestamp}_\texttt{d}$ (Procedure 17) that affects the enqueuer node corresponding to $rank(\texttt{op})$ 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 5.4.3.9 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 5.4.3.10 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 $min\text{-}ts\big(rank(n, t_x), t_y\big)$ 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, $min\text{-}ts\big(rank(n, t_x), t_y\big)$ 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, $\mathit{min\text{-}ts}(r, t_x)$ 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 $\mathit{rank}(n_0, t_x)$ 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), $\mathit{min\text{-}ts}\big(\mathit{rank}(n_0, t_x), t_y\big)$ 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 $t_{\mathit{start}\text{-}0} < t_{\mathit{start}\text{-}1} < t_{\mathit{start}\text{-}2} < \ldots < t_{\mathit{start}\text{-}k}$ and end them at $t_{\mathit{end}\text{-}0} < t_{\mathit{end}\text{-}1} < t_{\mathit{end}\text{-}2} < \ldots < t_{\mathit{end}\text{-}k}$. By definition, $t_{\mathit{end}\text{-}0} = t_e(n')$ and $t_{\mathit{start}\text{-}0} = t_s(n')$. We can prove that $t_{\mathit{end}\text{-}i} < t_{\mathit{start}\text{-}(i+1)}$ because successful CAS-sequences cannot overlap.

Due to how `refreshNode` (Procedure 13 and Procedure 18) is defined, for any $k \ge i \ge 1$:

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

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

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

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

Because $t_{\mathit{ts}\text{-}i}(c_i) \le t_{\mathit{end}\text{-}i}$ and $t_{\mathit{ts}\text{-}i}(c_{i-1}) \ge t_{\mathit{end}\text{-}(i-1)}$ and (2), we have for any $k \ge i \ge 1$,
$\mathit{min\text{-}ts}(\mathit{rank}(n', t_{\mathit{end}\text{-}i}), t_{\mathit{end}\text{-}i})$
$\le \mathit{min\text{-}ts}\Big(\mathit{rank}\Big(n', t_{\mathit{end}\text{-}(i-1)}\Big), t_{\mathit{end}\text{-}(i-1)}\Big)$. $(*)$

$rank(n', t_x)$ 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 46 of Procedure 13 and line 100 of Procedure 18). So, if `refreshNode` actually set 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 constantlt `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 5.4.3.11 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 $t \in \left[t_{enqueue}, t_1\right]$. (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 5.4.3.9, 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 $\left[t_{end\text{-}(i-1)}, t_1\right]$ 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, 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 5.4.3.10, $min\text{-}ts(rank(n_i, t_r), t_s)$ 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 5.4.3.10, $min\text{-}ts(rank(n_i, t_r), t_s) \leq c$ 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 5.4.3.12 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, $min\text{-}spsc\text{-}ts(rank(root, t_r), t_s) \leq c$ 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 5.4.3.11, we have that $min\text{-}ts(rank(root, t_r), t_s) \leq c$ for any $t_r, t_s \in T$.

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

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

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 5.4.3.13 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 $rank(n, t) \neq r$ 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 5.4.3.4, 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 5.4.3.8, 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 5.4.3.14 In LTQueueV1, 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 does not allow multiple dequeues, from the start of d at t_0 until after line 4 of dequeue (Procedure 15) 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 5.4.3.12, $min\text{-}spsc\text{-}ts\big(rank(root, t_x), t_y\big) \leq c$ for $t_x, t_y \in [t_0, t_1]$. Therefore, d reads out a rank r such that $min\text{-}spsc\text{-}ts(r, t) \leq c$ 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 5.4.3.15 In LTQueueV1, 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 5.4.3.16 In LTQueueV1, 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 5.4.3.15, 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 5.4.3.17 In LTQueueV1, 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 5.4.3.15, either e_1 precedes or overlaps with d_1.

If e_0 precedes d_1, applying Theorem 5.4.3.14 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 5.4.3.16 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 4 of dequeue (Procedure 15). 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 5.4.3.13, 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 5.4.3.13, 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 5.4.3.4, 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 LTQueue 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 5.4.3.11, d_1 will favor e_0 over e_1, which is a contradiction.

We have proved the theorem. □

Theorem 5.4.3.18 In LTQueueV1, 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 5.4.3.14:

- 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 5.4.3.19 *(Linearizability of LTQueueV1)* The LTQueueV1 algorithm is linearizable.

Proof Suppose some history H produced from the modified LTQueueV1 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 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, $X \nRightarrow_{H'} X$.

It's obvious that $X \nRightarrow_{H'} X$.

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 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 $X \rightarrow_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y, so (4) cannot apply. Applying Theorem 5.4.3.16, 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 $Y \rightarrow_{H'} X$, 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 5.4.3.17,
  - 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 5.4.3.1.
- A dequeue can only be matched by one enqueue: This follows from Theorem 5.4.3.2.
- 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 5.4.3.17, 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 5.4.3.15, 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 5.4.3.11.
- An enqueue that returns `true` will be matched if there are enough dequeues after that: Based on how Procedure 10 is defined, when an enqueue returns `true`, it has successfully execute `spsc_enqueue`. By Theorem 5.4.3.11, at some point, it would eventually be matched.
- An enqueue that returns `false` will never be matched: Based on how Procedure 10 is defined, when an enqueue returns `false`, the state of LTQueue 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. $\square$

5.4.4 Progress guarantee

Notice that every loop in LTQueueV1 is bounded, and no method have to wait for another. Therefore, LTQueueV1 is wait-free.

5.4.5 Memory reclamation

Notice that LTQueueV1 pushes the memory reclamation problem to the underlying SPSC. Because the underlying SPSC is memory-safe, LTQueueV1 is also memory-safe.

5.4.6 Performance model

5.5 Theoretical proofs of LTQueueV2

5.5.1 Notation

As a refresher, Figure 5 shows the structure of LTQueueV2.

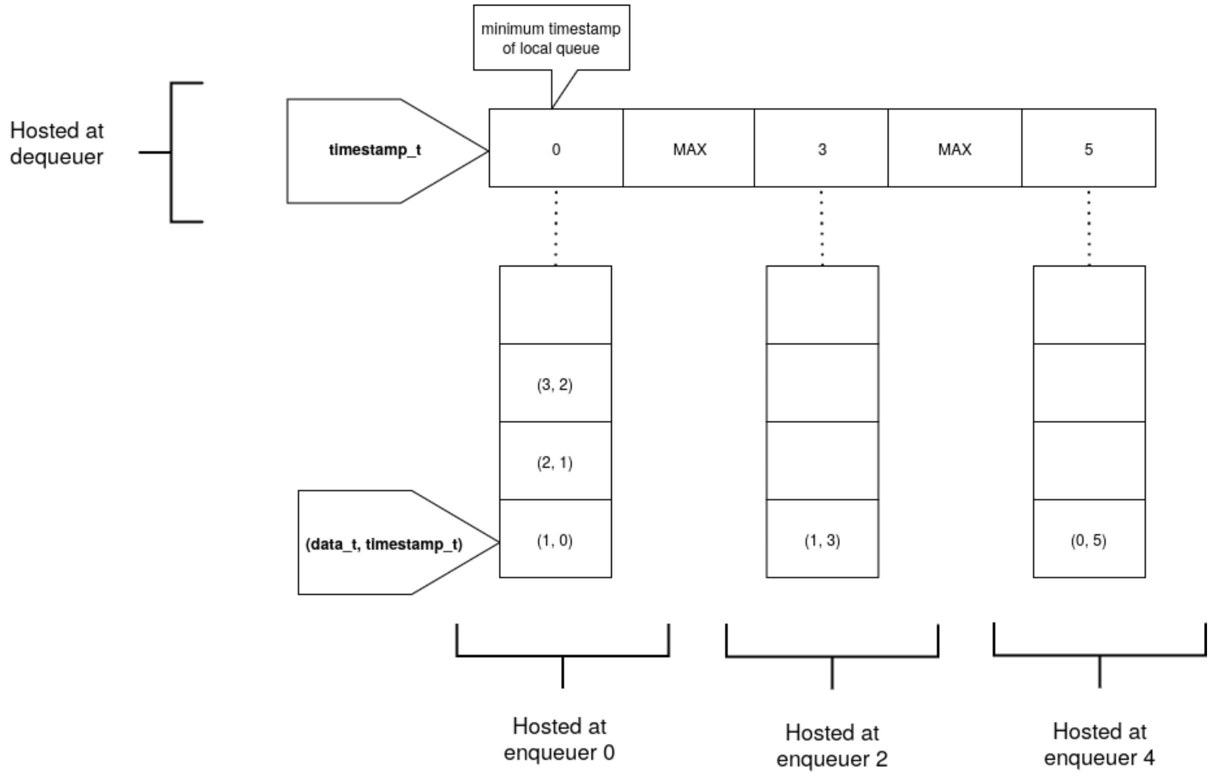

Figure 5: Basic structure of LTQueueV2

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 LTQueueV2 algorithm to the definitions introduced in Section 5.2.2.

Definition 5.5.1.1 A **CAS-sequence** on a slot `s` of an enqueue that affects `s` is the sequence of instructions from line 10 to line 16 of its `refreshEnqueue` (Procedure 23).

Definition 5.5.1.2 A **slot-modification instruction** on a slot `s` of an enqueue that affects `s` is line 16 of `refreshEnqueue` (Procedure 23).

Definition 5.5.1.3 A **CAS-sequence** on a slot `s` of a dequeue that affects `s` is the sequence of instructions from line 46 to line 50 of its `refreshDequeue` (Procedure 26).

Definition 5.5.1.4 A **slot-modification instruction** on a slot `s` of a dequeue that affects `s` is line 50 of `refreshDequeue` (Procedure 26).

Definition 5.5.1.5 A **CAS-sequence** of a dequeue/enqueue is said to **observe a slot value of s_0** if it loads s_0 at line 10 of `refreshEnqueue` or line 46 of `refreshDequeue`.

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

Definition 5.5.1.6 For an enqueue or dequeue op, $rank(op)$ is the rank of the enqueuer whose local SPSC is affected by op.

Definition 5.5.1.7 For an enqueuer whose rank is r, the value stored in its corresponding slot at time t is denoted as $slot(r, t)$.

Definition 5.5.1.8 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 5.5.1.9 For an enqueue, **slot-refresh phase** refer to its execution of line 5-6 of Procedure 22.

Definition 5.5.1.10 For a dequeue, **slot-refresh phase** refer to its execution of line 24-25 of Procedure 24.

Definition 5.5.1.11 For a dequeue, **slot-scan phase** refer to its execution of line 27-43 of Procedure 25.

5.5.2 ABA problem

Noticeably, we use no scheme to avoid ABA problem in LTQueueV2. 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 LTQueueV2 is ABA-safe, as introduced in Section 5.2.2.

Notice that we only use `CAS`es on:

- Line 16 of `refreshEnqueue` (Procedure 23), which is part of an enqueue.
- Line 48 of `refreshDequeue` (Procedure 26), which is part of a dequeue.

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

Theorem 5.5.2.1 *(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 5.5.2.2 *(Monotonicity of SPSC timestamps)* Each SPSC in LTQueueV2 contains elements with increasing timestamps.

Proof Each enqueue would `FAA` the distributed counter (line 3 in Procedure 22) and enqueue into the local SPSC an item with the timestamp obtained from the counter. Applying Theorem 5.5.2.1, 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 5.5.2.3 A `refreshEnqueue` (Procedure 23) can only changes a slot to a value other than `MAX_TIMESTAMP`.

Proof For `refreshEnqueue` to change the slot's value, the condition on line 14 must be `false`. Then, `new_timestamp` must equal to `ts`, which is not `MAX_TIMESTAMP`. It's obvious that the `CAS` on line 16 changes the slot to a value other than `MAX_TIMESTAMP`. □

Theorem 5.5.2.4 *(ABA safety of dequeue)* Assume that the 64-bit distributed counter never overflows, dequeue (Procedure 24) 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.

Due to Theorem 5.5.2.1, there can only be at most one enqueue at one point in time within d.

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. This means another process (dequeuer/enqueuer) has seen the value e enqueued and CAS `s` for e before t_d. By Theorem 5.5.2.1, 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 5.5.2.1 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 5.5.2.5 *(ABA safety of enqueue)* Assume that the 64-bit distributed counter never overflows, enqueue (Procedure 22) 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.

Due to Theorem 5.5.2.1, there can only be at most one enqueue at one point in time within e.

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`. 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 run. 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. 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 5.5.2.2, 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. 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 5.5.2.1, 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.

For e' to set `s` to the same value as d, e''s `spsc_readFront` must serialize after d's `spsc_dequeue`.

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 5.5.2.1 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 5.5.2.6 *(ABA safety)* Assume that the 64-bit distributed counter never overflows, Slot-queue is ABA-safe.

Proof This follows from Theorem 5.5.2.5 and Theorem 5.5.2.4. □

5.5.3 Linearizability

Theorem 5.5.3.7 In LTQueueV2, 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 5.5.3.8 In LTQueueV2, 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 5.5.3.9 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` or `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 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 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 5.5.2.1. 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 5.5.3.10 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 5.5.3.11 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.

By Theorem 5.5.3.9, there must be a successful refresh call that observes the effect of `spsc_enqueue` happening at t'', $t'' \in [t', t_0]$.

By the same reasoning as in Theorem 5.5.2.6, any successful slot-modification instructions happening after t'' must observe the effect of `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 5.5.3.12 In LTQueueV2, 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 25 is defined and by Theorem 5.5.3.11, 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 5.5.3.13 In LTQueueV2, 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 5.5.3.14 In LTQueueV2, 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 5.5.3.13, 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 5.5.3.15 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 5.5.3.13, 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, than 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 preceds 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 5.5.3.16 In LTQueueV2, 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 5.5.3.15, 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 5.5.3.17 *(Linearizability of LTQueueV2)* LTQueueV2 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 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 $X \rightarrow_{H'} Y$, then due to (1), $X \Rightarrow_{H'} Y$. By definition, X precedes Y, so (4) cannot apply. Applying Theorem 5.5.3.14, either
  - X isn't matched, (2) cannot apply. Therefore, $Y \nRightarrow_{H'} X$.
  - X matches e' and $e' \neq Y$. Therefore, X does not match Y, or (2) cannot apply. Therefore, $Y \nRightarrow_{H'} X$.

  Therefore, in this case, $X \Rightarrow_{H'} Y$ and $Y \nRightarrow_{H'} X$.
- If $Y \rightarrow_{H'} X$, 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 $X \nRightarrow_{H'} Y$.
- 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 $X \nRightarrow_{H'} Y$.
  - 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 $Y \nRightarrow_{H'} X$.

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 5.5.3.15,
  - Both X and Y aren't matched, then (3) cannot apply. Therefore, in this case, $Y \nRightarrow_{H'} X$.
  - X is matched but Y is not matched, then (3) cannot apply. Therefore, in this case, $Y \nRightarrow_{H'} X$.
  - 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 $Y \nRightarrow_{H'} X$.
- If $Y \rightarrow_{H'} X$, this case is symmetric to the first case. We obtain $Y \Rightarrow_{H'} X$ but $X \nRightarrow_{H'} Y$.
- 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 5.5.3.7.
- A dequeue can only be matched by one enqueue: This follows from Theorem 5.5.3.8.
- 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 5.5.3.15, 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 5.5.3.13, 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 5.5.3.11.
- An enqueue that returns `true` will be matched if there are enough dequeues after that: Based on how Procedure 22 is defined, when an enqueue returns `true`, it has successfully execute `spsc_enqueue`. By Theorem 5.5.3.11, at some point, it would eventually be matched.
- An enqueue that returns `false` will never be matched: Based on how Procedure 22 is defined, when an enqueue returns `false`, the state of LTQueueV2 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. $\square$

5.5.4 Progress guarantee

Notice that every loop in LTQueueV2 is bounded, and no method have to wait for another. Therefore, LTQueueV2 is wait-free.

5.5.5 Memory reclamation

Notice that LTQueueV2 pushes the memory reclamation problem to the underlying SPSC. Because the underlying SPSC is memory-safe, LTQueueV2 is also memory-safe.

5.5.6 Performance model

Chapter VI Preliminary results

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

Chapter VII Conclusion & Future works

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri.

References

[1] 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

[2] 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.

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

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

[6] 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.

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

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

[9] 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.

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

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

[12] M. M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects," 2004, *IEEE Press*. doi: 10.1109/TPDS.2004.8.

[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] 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.