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

SPECIALIZED PROJECT

STUDYING AND DEVELOPING NONBLOCKING DISTRIBUTED MPSC QUEUES

Major: Computer Science

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

—ooo—

STUDENT: ĐỖ NGUYỄN AN HUY - 2110193

HCMC, 07/2025

Disclaimers

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

Acknowledgements

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

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

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

I also want to give my family the sincerest thanks for their emotional and financial support, without which I couldn't have wholeheartedly 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 do not have the mental capacity for.

Contents

List of Tables

List of Images

Chapter I Introduction

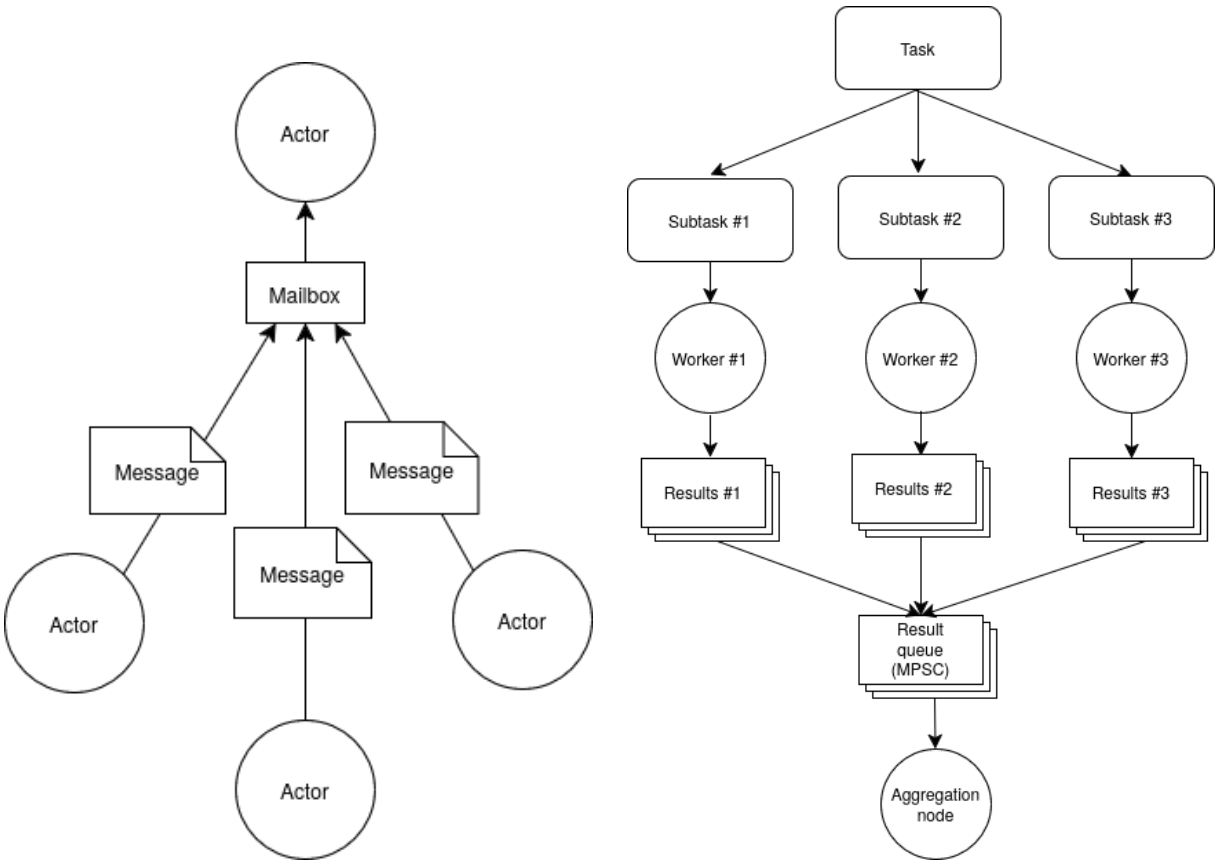
This chapter details the motivation for our research topic: “Studying and developing non-blocking distributed MPSC queues”, based on which we set out the objectives and scope of this study. To summarize, we then come to the formulation of our research question and give a high-level overview of the thesis.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

1.1 Motivation

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

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

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



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

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

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

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

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

1.2 Objective

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

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

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

1.3 Scope

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

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

1.4 Research question

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

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

We further decompose this question into smaller subquestions:

1. How to model the correctness of a distributed MPSC queue algorithm?
2. Which factors contribute to the fault tolerance and performance of distributed MPSC queue algorithms?
3. Which shared-memory programming principles are relevant in modeling and designing distributed MPSC queue algorithms in a fault-tolerant and performant manner?
4. Which shared-memory programming principles need to be modified to more effectively model and design distributed MPSC queue algorithms in a fault-tolerant and performant manner?

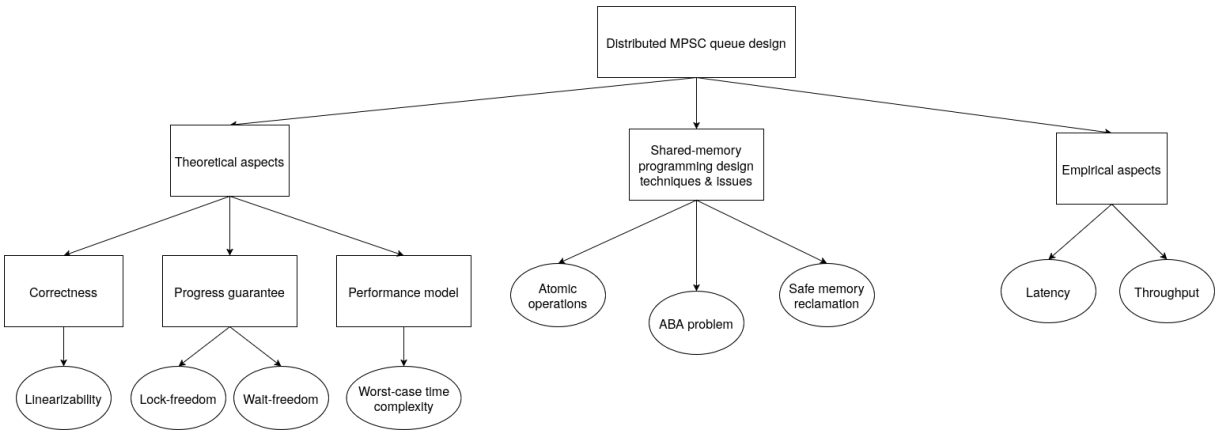


Figure 2: An overview of this thesis.

1.5 Thesis overview

An overview of this thesis is given in Figure 2.

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

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

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

1.6 Structure

The rest of this report is structured as follows:

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

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

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

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

Chapter V details our benchmarking metrics and elaborates on our benchmarking setup. We aim to demonstrate some preliminary results on how well our novel MPSC queue algorithms perform, additionally compared to existing distributed MPSC queues. Finally, we discuss important factors that affect the runtime properties of distributed MPSC queue algorithms.

Chapter VIII 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 to access next) is highly dependent on the values of the data already accessed, hence the unpredictable memory access property because we cannot statically analyze the program to know which data it will access. The control flow is inherently engraved in the data, which is not known until runtime.

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

2.1.1 Actor model as an irregular application

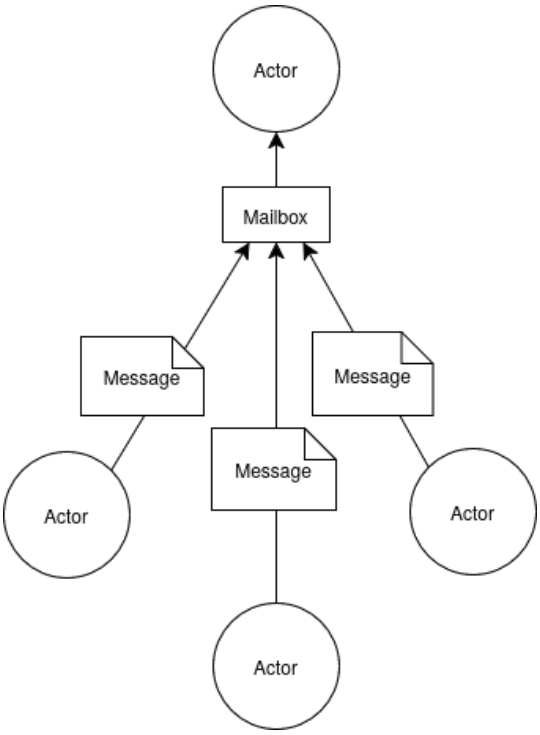


Figure 3: Actor model visualization.

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

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

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

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

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

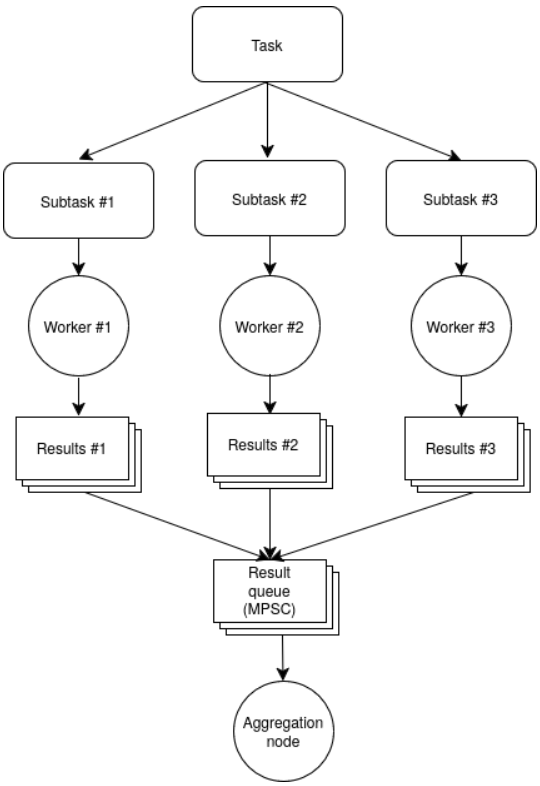


Figure 4: Fan-out/Fan-in pattern visualization.

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

In this pattern, there is a big task that can be split into subtasks to be executed concurrently on some work nodes. In the execution process, each worker produces a result set,

each enqueued back to a result queue located on an aggregation node. The aggregation node can then dequeue from this result queue to perform further processing. Clearly, this result queue exhibits MPSC behavior.

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

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

2.2 MPSC queue

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

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

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

- Unpredictable memory access: As a general data structure, the MPSC queue allows any process to enqueue and dequeue at any time. By nature, its memory access pattern is unpredictable.
- Data-dependent control flow: The consumer's behavior is entirely dependent on whether and which data is available in the MPSC queue. The execution paths of MPSC queues can vary, based on the queue contention i.e. some processes may back off or retry some failed operations; this scenario often arises in lock-free data structures.

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

2.3 Correctness condition of concurrent algorithms

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

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

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

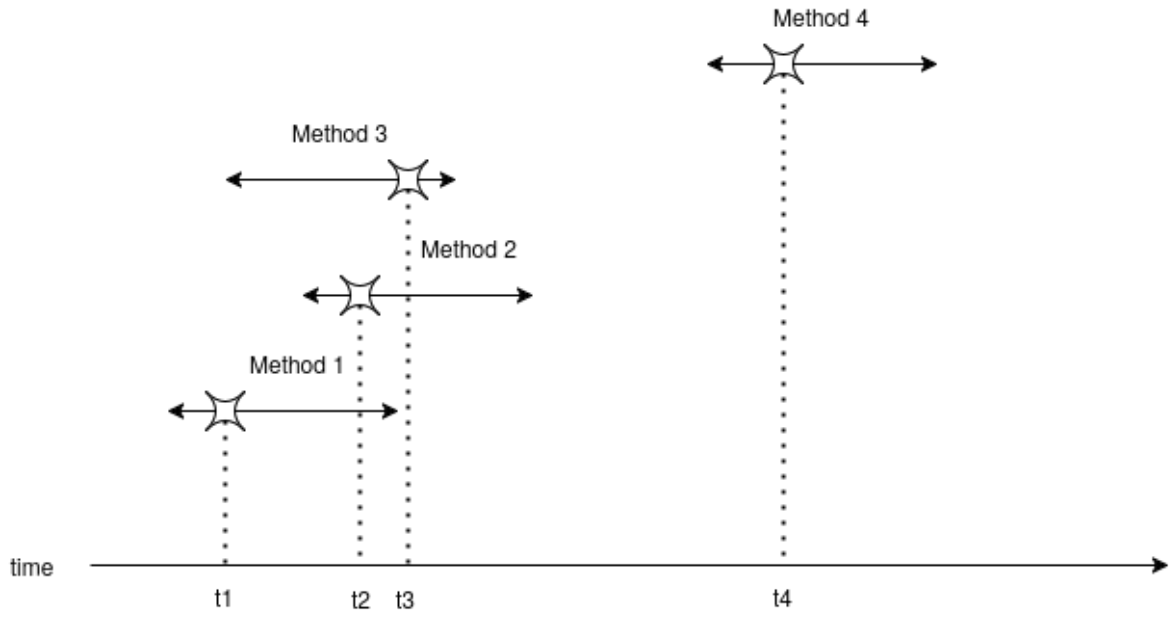


Figure 5: Linearization points of method 1, method 2, method 3, method 4 happen at $t_1 < t_2 < t_3 < t_4$, therefore, their effects will be observed in this order as if we call method 1, method 2, method 3, method 4 sequentially.

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

2.4 Progress guarantee of concurrent algorithms

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

2.4.1 Blocking algorithms

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

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

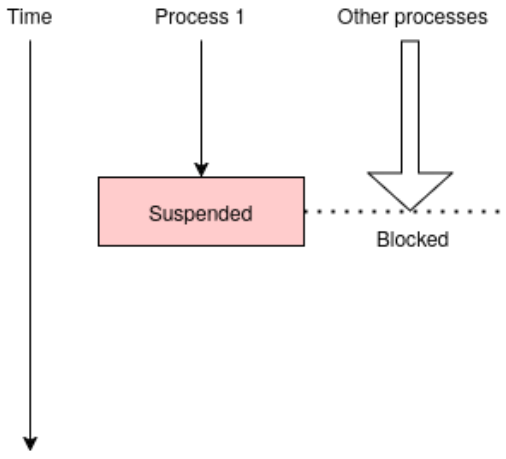


Figure 6: Blocking algorithm: When a process is suspended, it can potentially block other processes from making further progress.

Blocking algorithms introduce many problems such as:

- Deadlock: There is a circular lock-wait dependency among the processes, effectively preventing any processes from making progress.

- Convoy effect: One long process holding the lock will block other shorter processes contending for the lock.
- Priority inversion: A higher-priority process effectively has very low priority because it has to wait for another low priority process.

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

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

2.4.2 Non-blocking algorithms

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

2.4.2.1 Lock-free algorithms

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

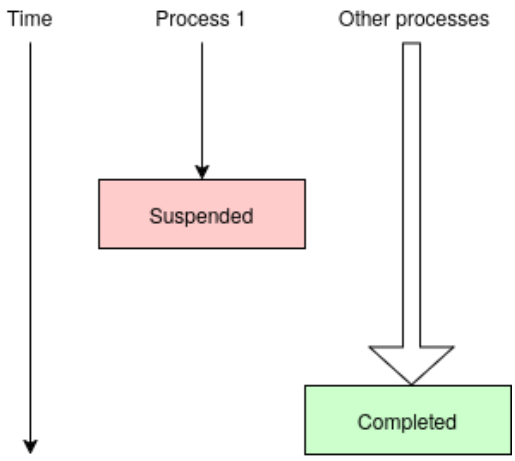


Figure 7: Lock-free algorithm: All the live processes together always finish in a finite amount of steps.

2.4.2.2 Wait-free algorithms

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

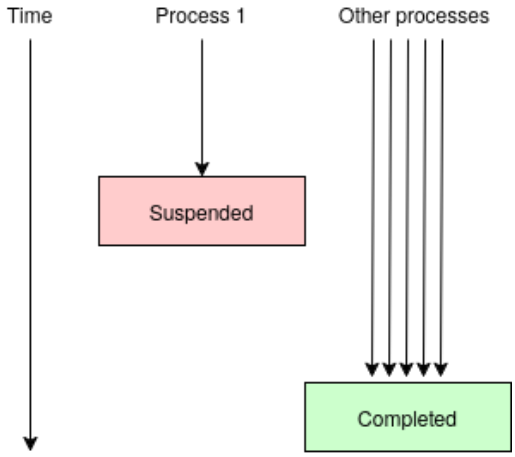


Figure 8: Wait-free algorithm: Any live process always finishes in a finite amount of steps.

2.5 Popular atomic instructions in designing non-blocking algorithms

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

2.5.1 Fetch-and-add (FAA)

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

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

```
1 old_value = *x
2 *x = *x + a
3 return old_value
```

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

2.5.2 Compare-and-swap (CAS)

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

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

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

```
1 if (*x == old_val)
2 |   *x = new_val
3 |   return true
4 return false
```

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

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

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

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

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

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

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

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

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

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

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

- Cache line flushing: If the cache line that caches the memory location x is written back to memory, logically, the memory location x has been accessed and therefore, the store-conditional fails.
- Context switch: If thread X is swapped out by the OS, cache lines may be invalidated and flushed out, which consequently leads to the first scenario.

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

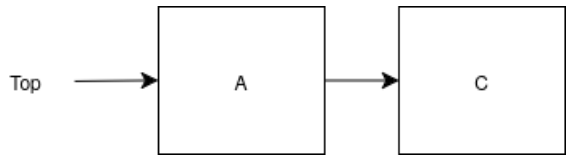
2.6 Common issues when designing non-blocking algorithms

2.6.1 ABA problem

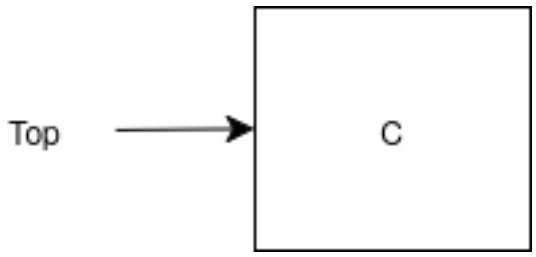
The ABA problem is a notorious problem associated with the compare-and-swap atomic instruction. Because CAS is so widely used in non-blocking algorithms, the ABA problem almost has to always be accounted for.

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

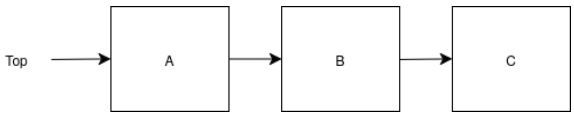
1. Read the current value `old value = read(memory location)`.
2. Compute `new value` from `old value` by manipulating some resources associated with `old value` and allocating new resources for `new value`.
3. Call `CAS(memory location, old value, new value)`. If that succeeds, the new resources for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



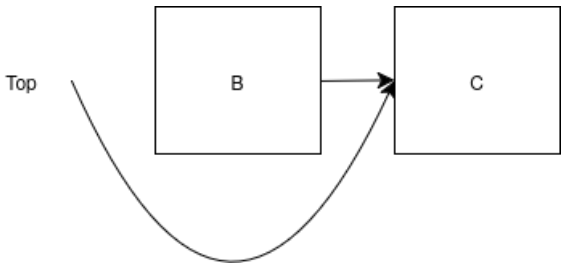
(a) Process X wants to pop a value, it observes Top = A and Top → next = C then suspends.



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



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



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

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

As hinted, this scheme is susceptible to the notorious ABA problem. The following scenario illustrates an example of the 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.

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

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

To safeguard against the 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 is no possibility another process has CAS-ed the memory location to the same value.

A simple scheme that is widely used practically and also in this thesis is the **unique timestamp** scheme. This scheme's idea is simple: for each shared memory location

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

2.6.2 Safe memory reclamation problem

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

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

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

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

Figure 10: Unsafe memory reclamation in a LIFO stack.

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

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

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

2.7.1 MPI-3 RMA

RMA in MPI RMA stands for remote memory access. As introduced in the first section of Section Chapter II, RMA APIs were introduced in MPI-2 and their capabilities are

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

2.7.2 MPI-RMA communication operations

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

To utilize MPI RMA, each process needs to open a memory window to expose a segment of its memory to RMA communication operations such as remote writes (`MPI_PUT`), remote reads (`MPI_GET`) or remote accumulates (`MPI_ACCUMULATE`, `MPI_GET_ACCUMULATE`, `MPI_FETCH_AND_OP`, `MPI_COMPARE_AND_SWAP`) [2]. These remote communication operations only require one side to specify.

2.7.3 MPI-RMA synchronization

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

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

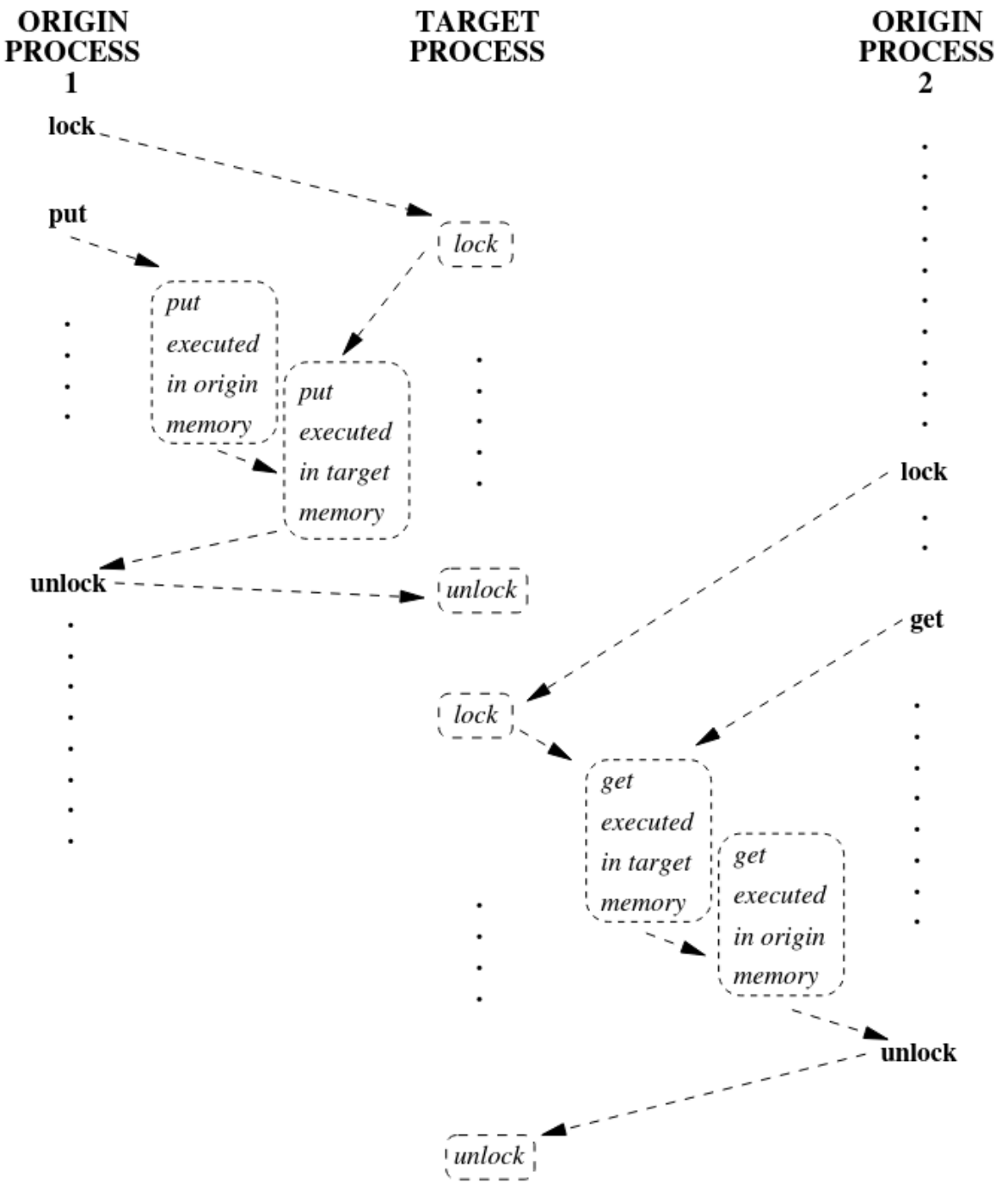


Figure 11: An illustration of passive target communication. Dashed arrows represent synchronization (source: [2]).

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

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

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

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

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

Operation	Usage
`MPI_Win_lock_all`	Starts an 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. This is in line with our intention to use MPI RMA to easily model irregular applications like MPSC queues.
- Unlike **active target synchronization**, `MPI_Win_lock_all` and `MPI_Win_unlock_all` do not need to wait for a matching synchronization call in the target process, and thus, are 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 does 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 effect
MPI_Win_flush_local(...); // Make previous RMA operations take effect
locally
...

MPI_Win_unlock_all(win);
```

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

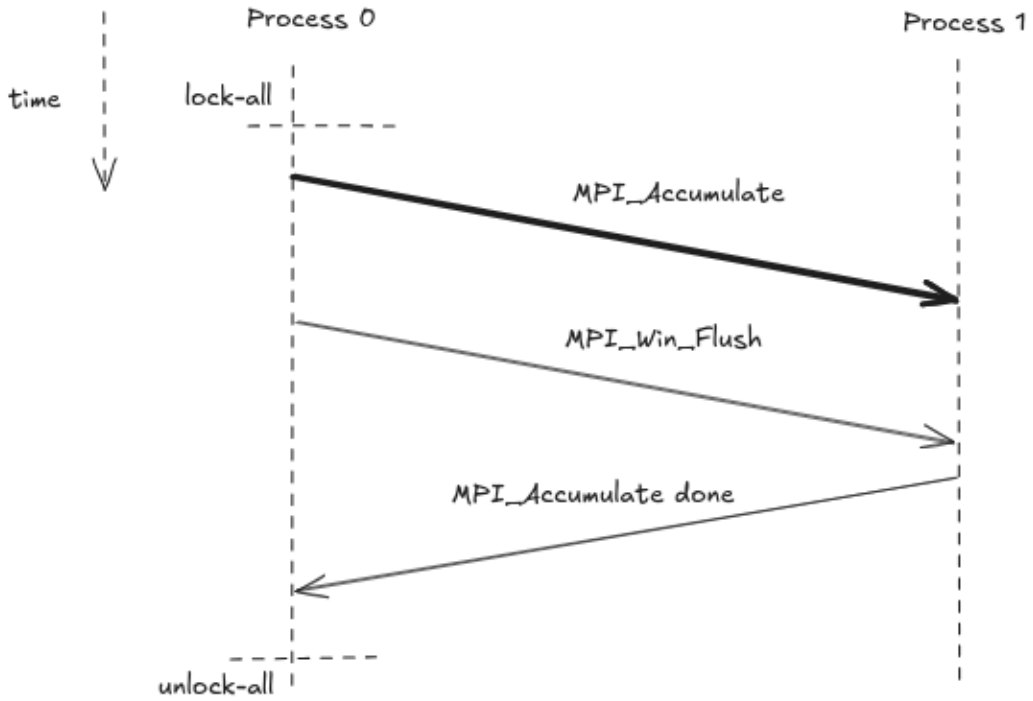


Figure 12: An illustration of our synchronization approach in MPI RMA.

Chapter III Related works

3.1 Non-blocking shared-memory MPSC queues

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

MPSC queues	LTQueue [8]	DQueue [14]	WRLQueue [15]	Jiffy [9]
ABA solution	Load-link/ Store-conditional	Incorrect custom scheme (*)	Custom scheme	Custom scheme
Safe memory reclamation	Custom scheme	Incorrect custom scheme (*)	Custom scheme	Insufficient 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

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

3.1.1 LTQueue

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

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

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

The rectangular nodes at the bottom in Figure 13 represent an enqueuer, whose SPSC contains items with 2 fields: value and timestamp. Every enqueuer has to timestamp its

data before enqueueing. The timestamps can be obtained using a distributed counter shared by all the enqueuers.

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

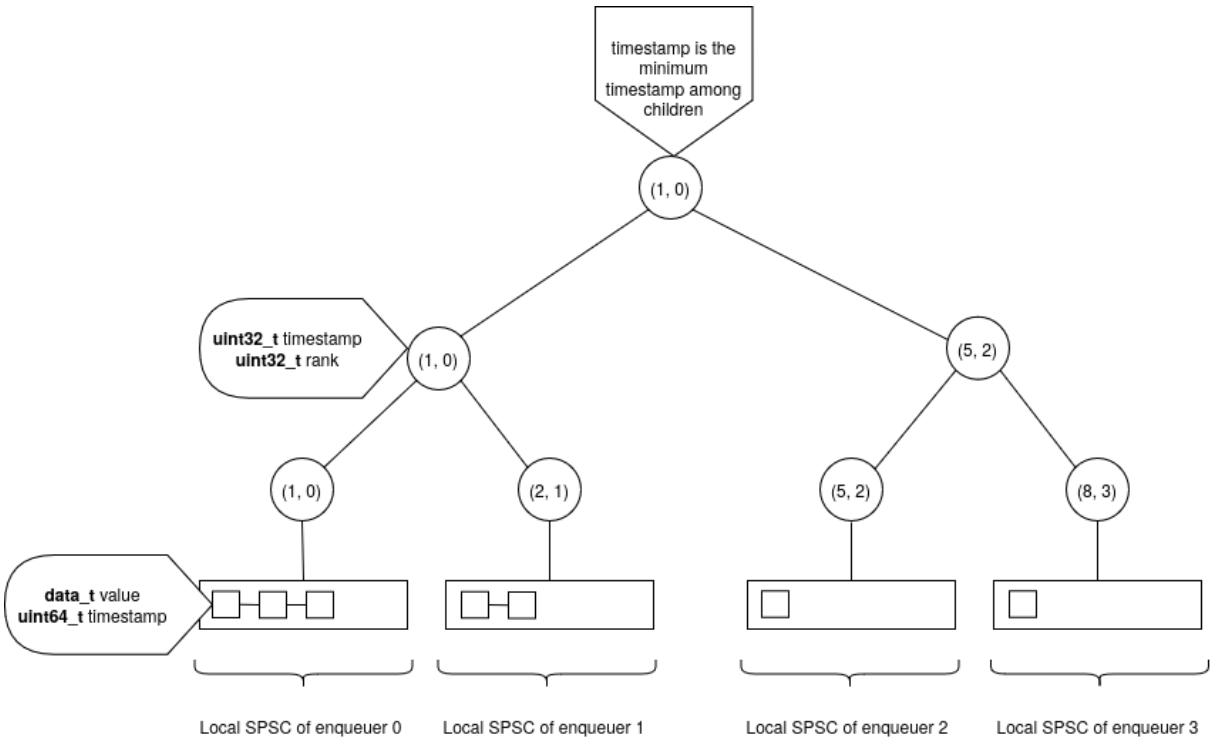


Figure 13: LTQueue's structure.

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

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

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

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

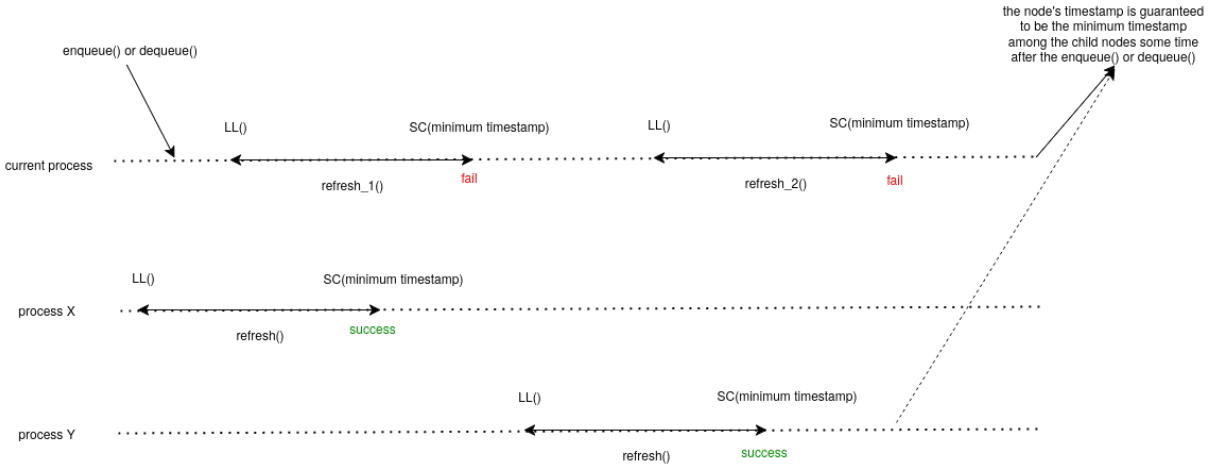


Figure 14: Intuition on how timestamp-refreshing works.

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

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

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

3.1.2 DQueue

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

The basic structure of DQueue is demonstrated in Figure 15.

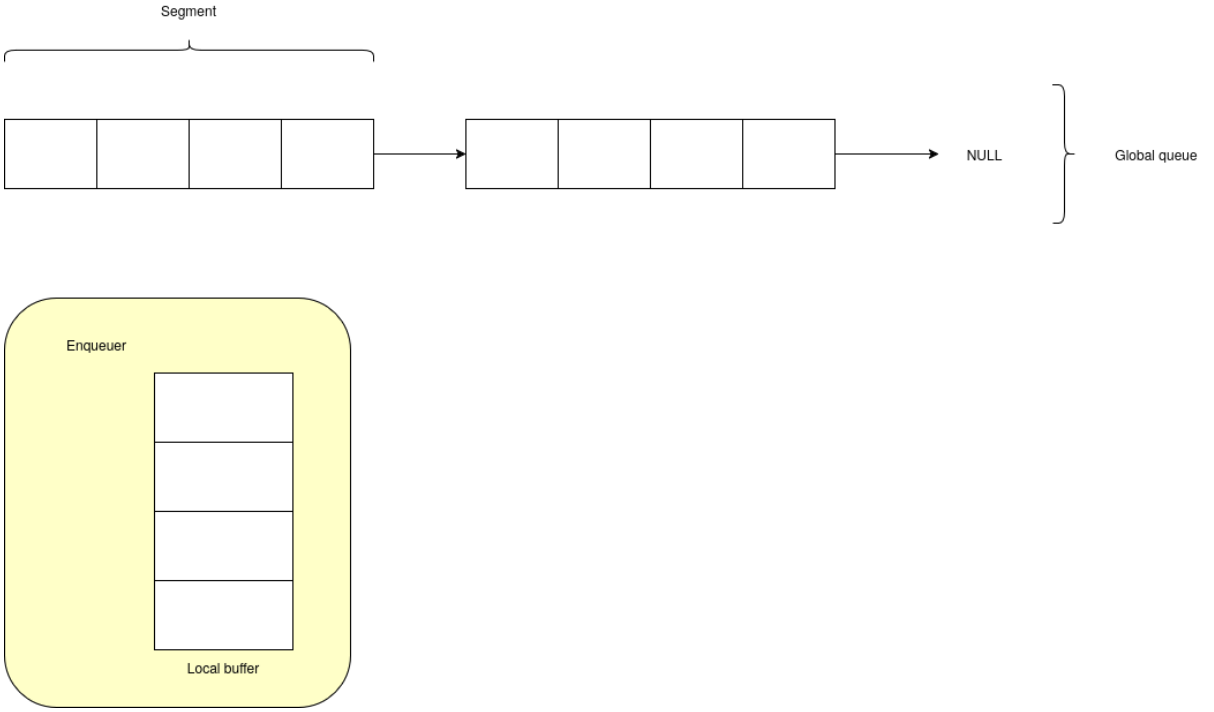


Figure 15: DQueue's structure.

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

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

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

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

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

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

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

3.1.3 WRLQueue

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

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

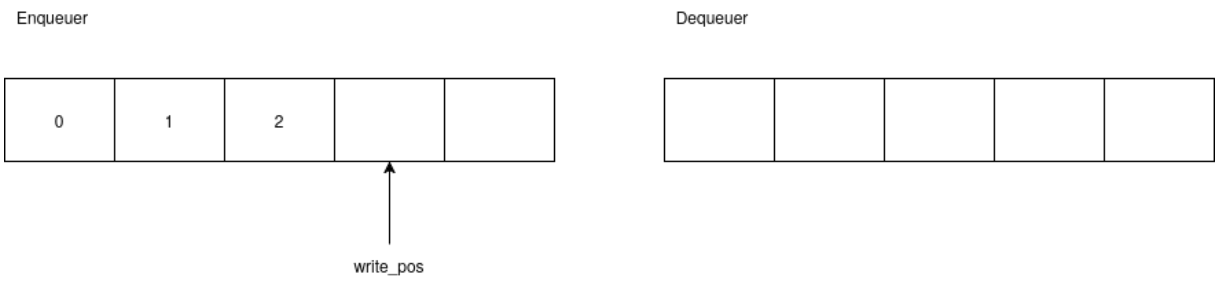


Figure 16: WRLQueue's structure.

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

The dequeuer, upon invocation, will swap its buffer with the enqueuers' buffer to dequeue from it, as in Figure 17. However, WRLQueue explicitly states that the dequeuer

has to wait for all enqueue operations to complete in the other buffer before swapping. If an enqueue suspends or dies, the dequeuer will experience a slowdown; this clearly violates the property of non-blocking. Therefore, we believe that WRLQueue is blocking, concerning its dequeue operation.

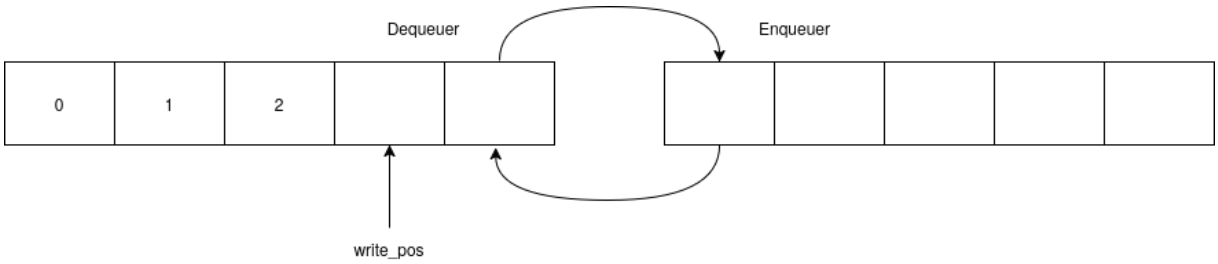


Figure 17: WRLQueue's dequeue operation

3.1.4 Jiffy

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

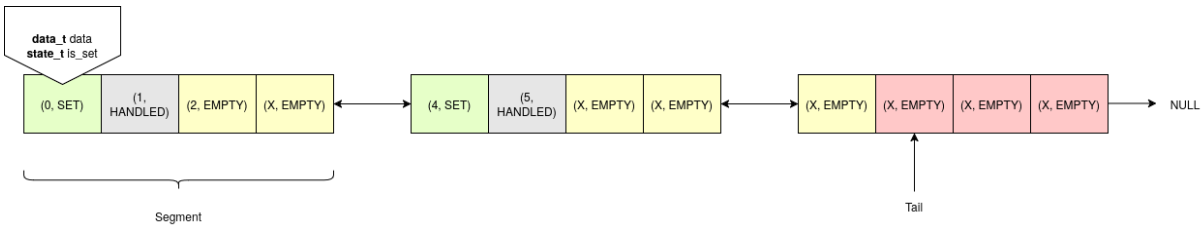


Figure 18: Jiffy's structure.

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

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

To dequeue, the dequeuer would start from the Head index of the first segment, scanning until it finds the first non-`HANDLED` slot before the end of the queue. If there is no such slot, the queue is empty, and the dequeuer would return nothing. If this slot is `SET`, it simply reads the data item in this slot and sets it to `HANDLED`. If this slot is `EMPTY`,

that means this slot has been reserved by an enqueuer that has not finished. In this case, the dequeuer performs a scan forward to find the first `SET` slot. If not found, the dequeuer returns nothing. Otherwise, it continues to repeatedly scan all slots between the first non-`HANDLED` and the last found `SET` slot until the first `SET` slot in this interval is unchanged between 2 scans. Only then, the dequeuer would return the data item in this `SET` slot and mark it as `HANDLED`.

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

Regarding memory reclamation, Jiffy does not specify a sufficient scheme: If one enqueuer is delayed forever, no memory is ever reclaimed. As a consequence, if an enqueuer is delayed for too long, the system will run out of memory, causing other enqueuers to fail without making any progress. Effectively, Jiffy is not wait-free.

3.1.5 Remarks

Out of the 4 investigated MPSC queue algorithms, we quickly eliminate DQueue, WRLQueue, and Jiffy as potential candidates for porting to a distributed environment because they either do not provide a sufficient progress guarantee or protection against the ABA problem and memory reclamation problem. Therefore, we will only adapt LTQueue for distributed environments in the next section. LTQueue also presents some challenges, though, as it utilizes LL/SC for the ABA solution, which does not exist in distributed environments. Consequently, to adapt LTQueue, we have to work around LTQueue's usage of LL/SC.

3.2 Distributed MPSC queues

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

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

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

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

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

- `WriterCnt`: The number of enqueuers currently writing in this queue.
- `Offset`: The index to the first empty entry in the queue. Note that any shared data and control variables are hosted on the dequeuer.

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

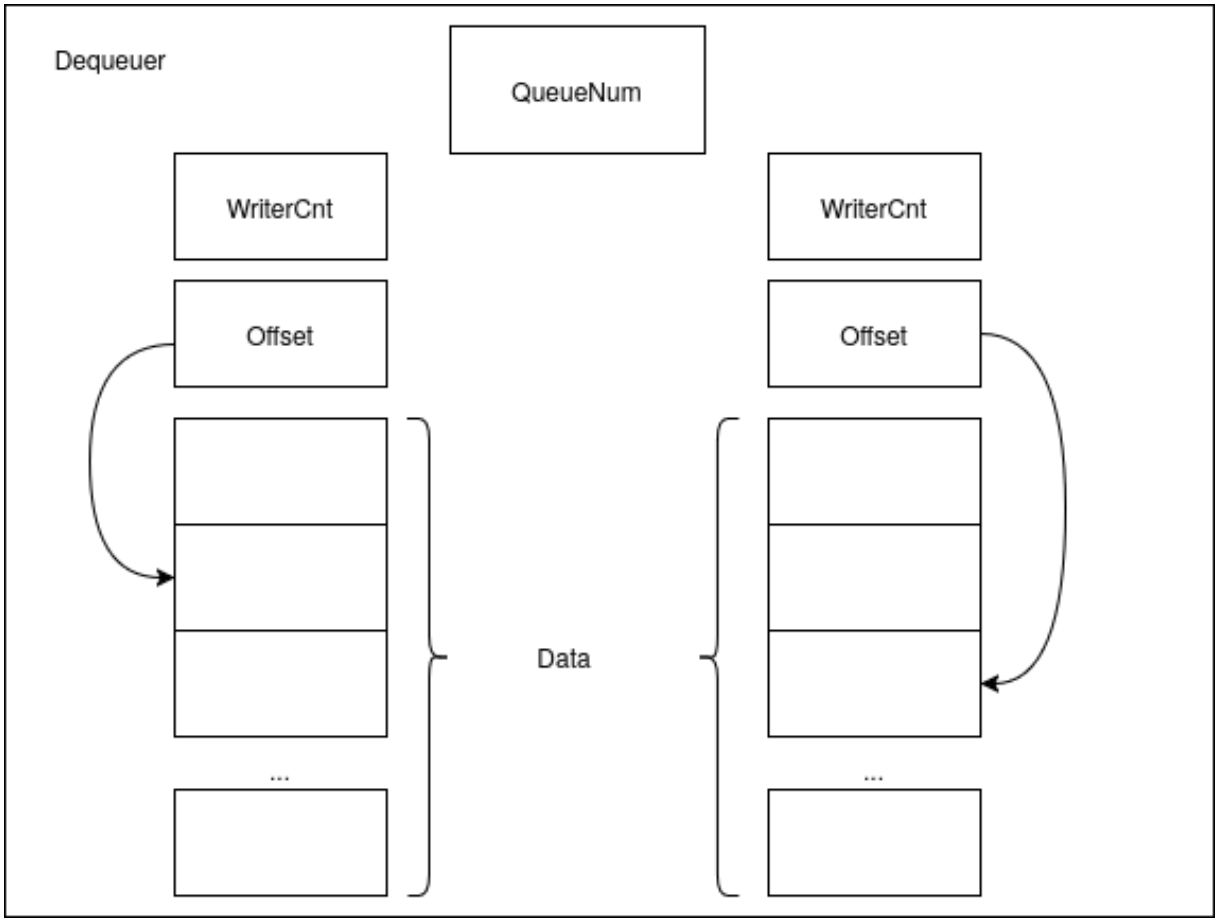


Figure 19: AMQueue's structure.

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

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

Based on our discussion, there is currently no non-blocking distrbuted MPSC queue in the literature. This makes our reasearch the first one of the kind to be about non-blocking distributed MPSC queues. AMQueue will serve as a benchmarking baseline for our MPSC queues in Chapter V.

Chapter IV Distributed MPSC queues

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

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

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

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

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

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

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

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

In the following sections, we present first the one-sided-communication primitives that we assume will be available in our distributed algorithm specification and then our proposed distributed MPSC queue wrappers in detail.

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

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

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

`remote<T>`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

> Issue an asynchronous write of the value at the local memory location pointed to by `src` into the element at position `index` within the distributed array `dest` (where

> `dest` is a pointer to a remotely hosted array of type `T`). The operation may not be completed when the function returns.

`void flush(remote<T> src)`

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

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

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

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

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

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

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

4.2 A simple baseline distributed SPSC

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

support an unbounded number of elements, as long as this SPSC supports the same interface as ours.

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

Types

> `data_t` = The type of data stored.

Shared variables

> `First`: `remote<uint64_t>`
>
> > The index of the first undequeued entry.
> >
> > Hosted at the dequeuer.
>
> `Last`: `remote<uint64_t>`
>
> > The index of the first unenqueued entry.
> >
> > Hosted at the dequeuer.
>
> `Data`: `remote<data_t*>`
>
> > An array of `data_t` of some known capacity.
> >
> > Hosted at the enqueuer.

Enqueuer-local variables

> `Capacity`: A read-only value indicating the capacity of the SPSC.
>
> `First_buf`: The cached value of `First`.
>
> `Last_buf`: The cached value of `Last`.

Dequeuer-local variables

> `Capacity`: A read-only value indicating the capacity of the SPSC.
>
> `First_buf`: The cached value of `First`.
>
> `Last_buf`: The cached value of `Last`.

Enqueuer initialization

> Initialize `First` and `Last` to `0`.
>
> Initialize `Capacity`.
>
> Allocate array in `Data`.
>
> Initialize `First_buf = Last_buf = 0`.

Dequeuer initialization

> Initialize `Capacity`.
>
> Initialize `First_buf = Last_buf = 0`.

The procedures of the enqueuer are given as follows.

Procedure 4: `bool spsc_enqueue(data_t v)`

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

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

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

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

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

The procedures of the dequeuer are given as follows.

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

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

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

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

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

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

4.3 dLTQueue - Straightforward LTQueue adapted for distributed environment

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

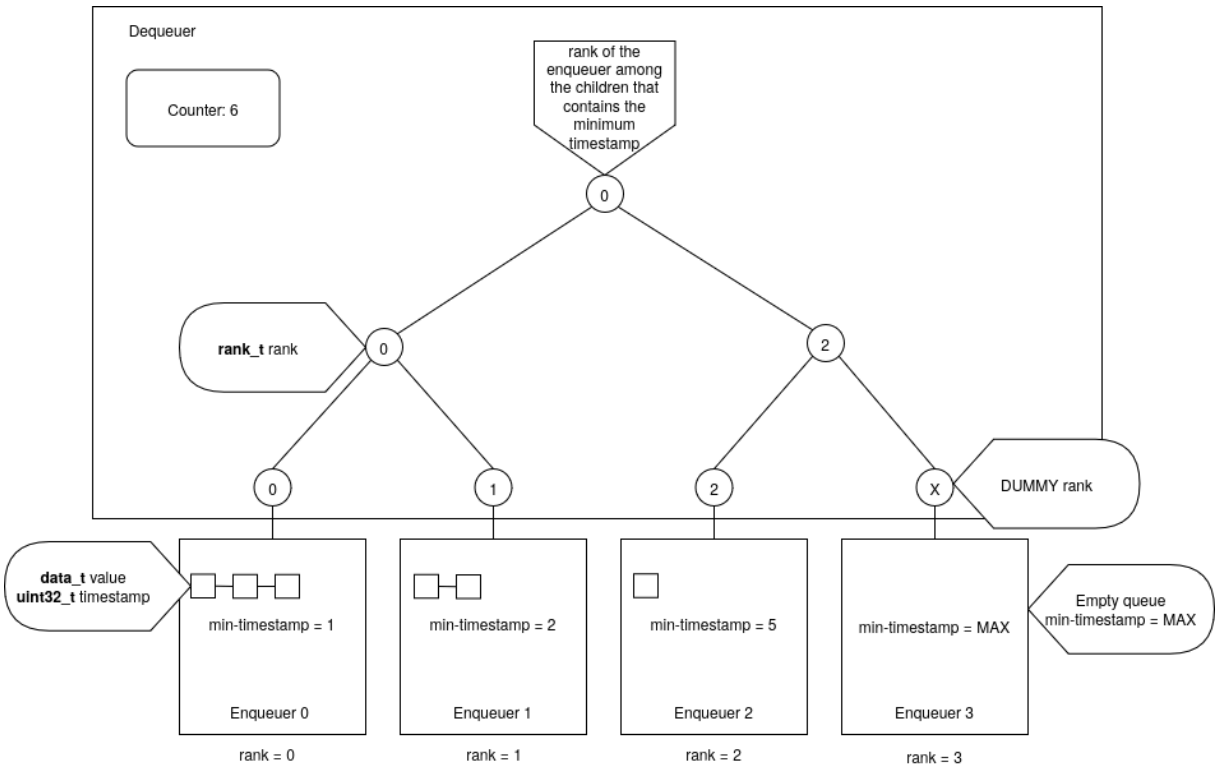


Figure 20: dLTQueue's structure.

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

4.3.1 Overview

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

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

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

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

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

Placement-wise:

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

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

4.3.2 Data structure

Below is the types utilized in dLTQueue.

Types

> `data_t` = The type of the data to be stored.
>
> `spsc_t` = The type of the SPSC, this is assumed to be the distributed SPSC in Section 4.2.
>
> `rank_t` = The type of the rank of an enqueuer process tagged with a unique timestamp (version) to avoid ABA problem.
>
> > **struct**
> >
> > > `value`: `uint32_t`
> > >
> > > `version`: `uint32_t`
> >
> > **end**
>
> `timestamp_t` = The type of the timestamp tagged with a unique timestamp (version) to avoid ABA problem.
>
> > **struct**
> >
> > > `value`: `uint32_t`
> > >
> > > `version`: `uint32_t`
> >
> > **end**
>
> `node_t` = The type of a tree node.
>
> > **struct**
> >
> > > `rank`: `rank_t`
> >
> > **end**

The shared variables in our LTQueue version are as follows.

Note that we have described a very specific and simple way to organize the tree nodes in dLTQueue in a min-heap-like array structure hosted on the sole dequeuer. We will resume our description of the related tree-structure procedures `parent()` (Procedure 8), `children()` (Procedure 9), `leafNodeIndex()` (Procedure 10) with this representation in mind. However, our algorithm does not strictly require this representation and can be substituted with other more-optimized representations & distributed placements, as long as the similar tree-structure procedures are supported.

Shared variables

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

`Tree_size`: `uint64_t`

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

`Nodes`: `remote<node_t>`

> An array with `Tree_size` entries storing all the tree nodes present in the dLTQueue shared by all processes.
>
> Hosted at the dequeuer.
>
> This array is organized in a similar manner as a min-heap: At index `0` is the root node. For every index $i > 0$, $\left\lfloor \frac{i-1}{2} \right\rfloor$ is the index of the parent of node i. For every index $i > 0$, $2i + 1$ and $2i + 2$ are the indices of the children of node i.

`Dequeuer_rank`: `uint32_t`

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

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

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

Enqueuer-local variables

`Process_count`: `uint64_t`

> The number of processes.

`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

`Process_count`: `uint64_t`

> The number of processes.

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

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

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

Enqueuer initialization

> Initialize `Process_count`, `Self_rank` and `Dequeuer_rank`.
>
> Initialize `Spsc` to the initial state.
>
> Initialize `Min_timestamp` to `timestamp_t {MAX_TIMESTAMP, 0}`.

Dequeuer initialization

> Initialize `Process_count`, `Self_rank` and `Dequeuer_rank`.
>
> Initialize `Counter` to `0`.
>
> Initialize `Tree_size` to `Process_count * 2`.
>
> Initialize `Nodes` to an array with `Tree_size` entries. Each entry is initialized to `node_t {DUMMY_RANK}`.

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

4.3.3 Algorithm

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

Procedure 8: `uint32_t parent(uint32_t index)`

```
2 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 9: `vector<uint32_t> children(uint32_t index)`

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

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

Procedure 10: `uint32_t leafNodeIndex(uint32_t enqueuer_rank)`

```
13 return Tree_size + enqueuer_rank
```

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

The followings are the enqueuer procedures.

Procedure 11: `bool enqueue(data_t value)`

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

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

Procedure 12: `void propagate_e()`

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

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

Procedure 13: `bool refreshTimestamp_e()`

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

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

Procedure 14: `bool refreshNode_e(uint32_t current_node_index)`

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

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

Procedure 15: `bool refreshLeaf`$_e$`()`

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

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

The followings are the dequeuer procedures.

Procedure 16: `bool dequeue(data_t* output)`

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

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

Procedure 17: `void propagate_d(uint32_t enqueuer_rank)`

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

The `propagate_d` procedure is similar to `propagate_e`, with appropriate changes to accommodate the dequeuer.

Procedure 18: `bool refreshTimestamp_d(uint32_t enqueuer_rank)`

```
82 enqueuer_order = enqueuer_rank
83 min_timestamp = timestamp_t {}
84 aread_sync(Timestamps, enqueuer_order, &min_timestamp)
85 {old-timestamp, old-version} = min_timestamp
86 front = (data_t {}, timestamp_t {})
87 is_empty = !spsc_readFront(&Spscs[enqueuer_order], &front)
88 if (is_empty)
     | return compare_and_swap_sync(Timestamps, enqueuer_order,
89   | timestamp_t {old-timestamp, old-version},
     | timestamp_t {MAX_TIMESTAMP, old-version + 1})
90 else
     | return compare_and_swap_sync(Timestamps, enqueuer_order,
91   | 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 19: `bool refreshNode_d(uint32_t current_node_index)`

```
92  current_node = node_t {}
93  aread_sync(Nodes, current_node_index, &current_node)
94  {old-rank, old-version} = current_node.rank
95  min_rank = DUMMY_RANK
96  min_timestamp = MAX_TIMESTAMP
97  for child_node_index in children(current_node)
98      child_node = node_t {}
99      aread_sync(Nodes, child_node_index, &child_node)
100     {child_rank, child_version} = child_node
101     if (child_rank == DUMMY_RANK) continue
102     child_timestamp = timestamp_t {}
103     aread_sync(Timestamps[child_rank], &child_timestamp)
104     if (child_timestamp < min_timestamp)
105         min_timestamp = child_timestamp
106         min_rank = child_rank
    return compare_and_swap_sync(Nodes, current_node_index,
107 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 20: `bool refreshLeaf_d(uint32_t enqueuer_rank)`

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

The `refreshLeaf_d` procedure is similar to `refreshLeaf_e`, with appropriate changes to accommodate the dequeuer.

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

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

4.4.1 Overview

The structure of Slotqueue is shown as in Figure 21.

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

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

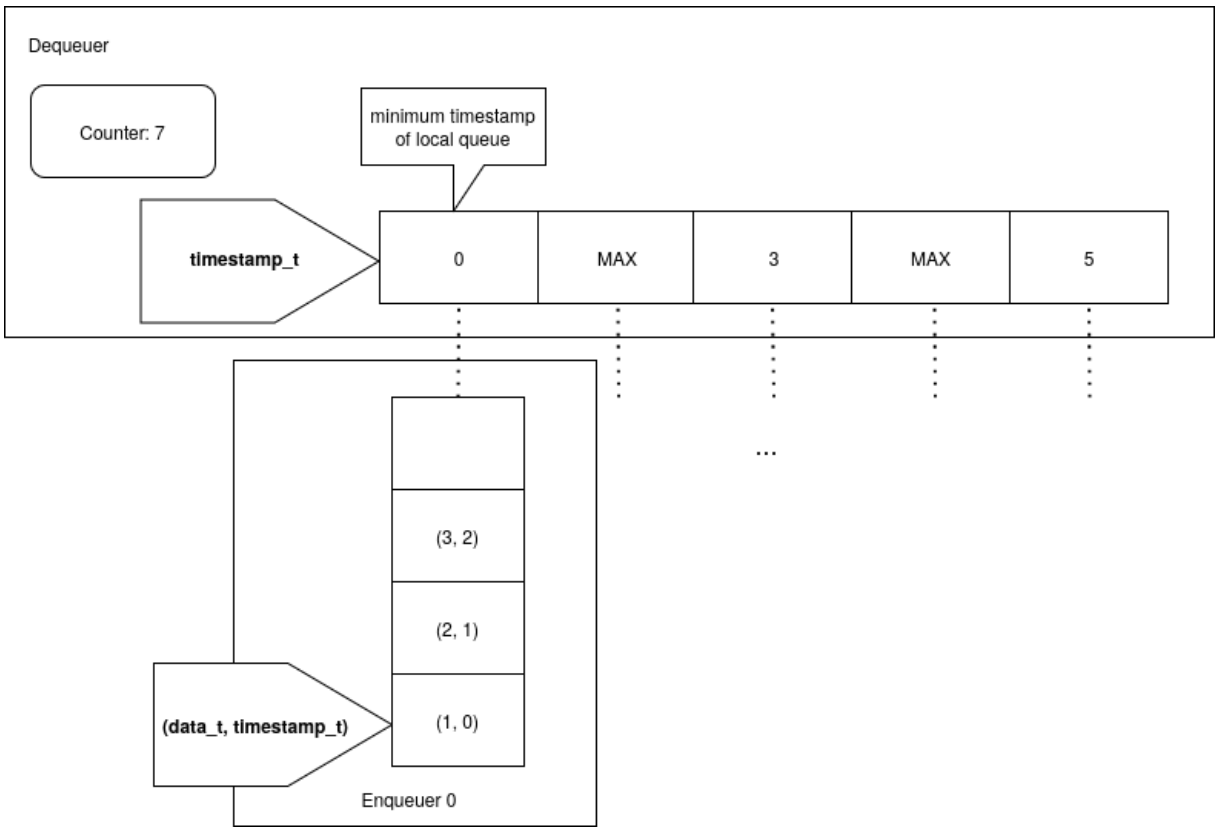


Figure 21: Basic structure of Slotqueue.

4.4.2 Data structure

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

Types

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

Shared variables

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

Enqueuer-local variables

> `Dequeuer_rank`: `uint64_t`
>
> > The rank of the dequeuer.
>
> `Process_count`: `uint64_t`

The number of enqueuers.

`Self_rank`: `uint32_t`

The rank of the current enqueuer process.

`Spsc`: `spsc_t`

This SPSC is synchronized with the dequeuer.

Dequeuer-local variables

`Dequeuer_rank`: `uint64_t`

The rank of the dequeuer.

`Process_count`: `uint64_t`

The number of enqueuers.

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

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

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

Enqueuer initialization

Initialize `Dequeuer_rank`.

Initialize `Process_count`.

Initialize `Self_rank`.

Initialize the local `Spsc` to its initial state.

Dequeuer initialization

Initialize `Dequeuer_rank`.

Initialize `Process_count`.

Initialize `Counter` to 0.

Initialize the `Slots` array with size equal to the number of enqueuers and every entry is initialized to `MAX_TIMESTAMP`.

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

4.4.3 Algorithm

The enqueuer operations are given as follows.

Procedure 21: `bool enqueue(data_t v)`

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

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

Procedure 22: `bool refreshEnqueue(timestamp_t ts)`

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

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

The dequeuer operations are given as follows.

Procedure 23: `bool dequeue(data_t* output)`

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

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

Procedure 24: `uint64_t readMinimumRank()`

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

`readMinimumRank`'s main responsibility is to return the rank of the enqueuer from which we can safely dequeue next. It first creates a local buffer to store the value

read from `Slots` (Line 31). It then performs 2 scans of `Slots` and read every entry into `buffered_slots` (Line 32 - Line 38). If the first scan finds only `MAX_TIMESTAMP`s, `DUMMY_RANK` is returned (Line 35). From there, based on `bufferred_slots`, it returns the rank of the enqueuer whose bufferred slot stores the minimum timestamp (Line 40 - Line 45).

Procedure 25: `refreshDequeue(rank: int)` **returns** `bool`

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

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

Chapter V Preliminary results

Chapter VI Benchmarking

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

Currently, performance-related properties are our main focus.

6.1 Benchmarking metrics

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

6.1.1 Throughput

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

6.1.2 Latency

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

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

In concurrent algorithms, multiple tasks are executed by multiple processes. The key observation is that, if we fix the number of processes, the lower the average latency of a task, the larger the number of tasks that can be completed by a process, which

implies higher throughput. Therefore, good latency often (but not always) implies good throughput.

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

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

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

6.2 Benchmarking baselines

We use three MPSC queue algorithms as benchmarking baselines:

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

6.3 Microbenchmark program

Our microbenchmark is as follows:

All processes share a single MPSC; one of the processes is a dequeuer, and the rest are enqueuers. The enqueuers enqueue a total of 10^4 elements. The dequeuer dequeues 10^4 elements. For MPSC, the MPSC is warmed up before the dequeuer starts.

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

6.4 Benchmarking setup

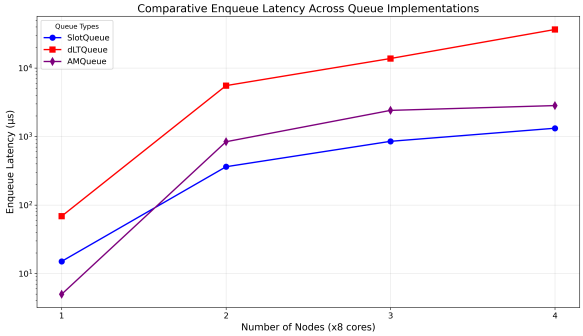
The experiments are carried out on a four-node cluster residing in the HPC Lab at Ho Chi Minh University of Technology. Each node is an Intel Xeon CPU E5-2680 v3, which has 8 cores and 16 GB RAM. The interconnect used is Ethernet and, thus, does not support true one-sided communication.

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

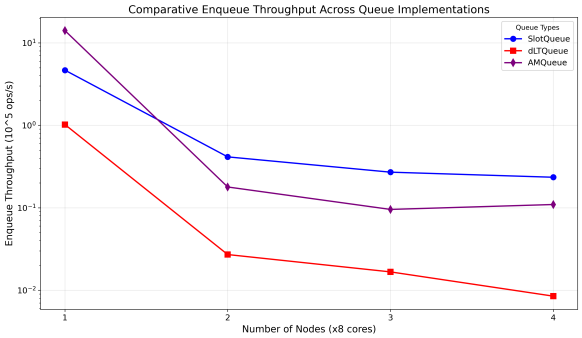
We run the producer-consumer microbenchmark on 1 to 4 nodes to measure both the latency and performance of our MPSC algorithms.

6.5 Benchmarking results

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

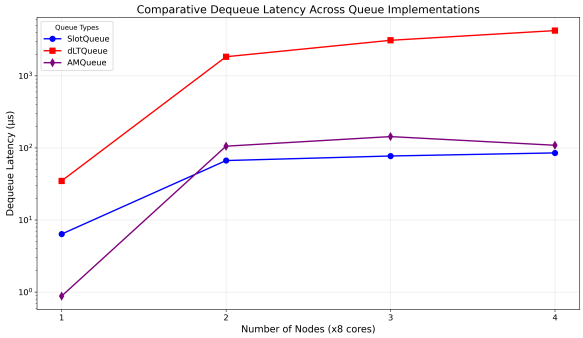


(a) Enqueue latency benchmark results.

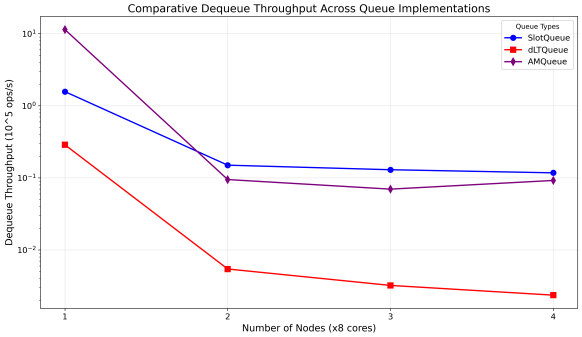


(b) Enqueue throughput benchmark results

Figure 22: Microbenchmark results for enqueue operation.



(a) Dequeue latency benchmark results.



(b) Dequeue throughput benchmark results

Figure 23: Microbenchmark results for dequeue operation.

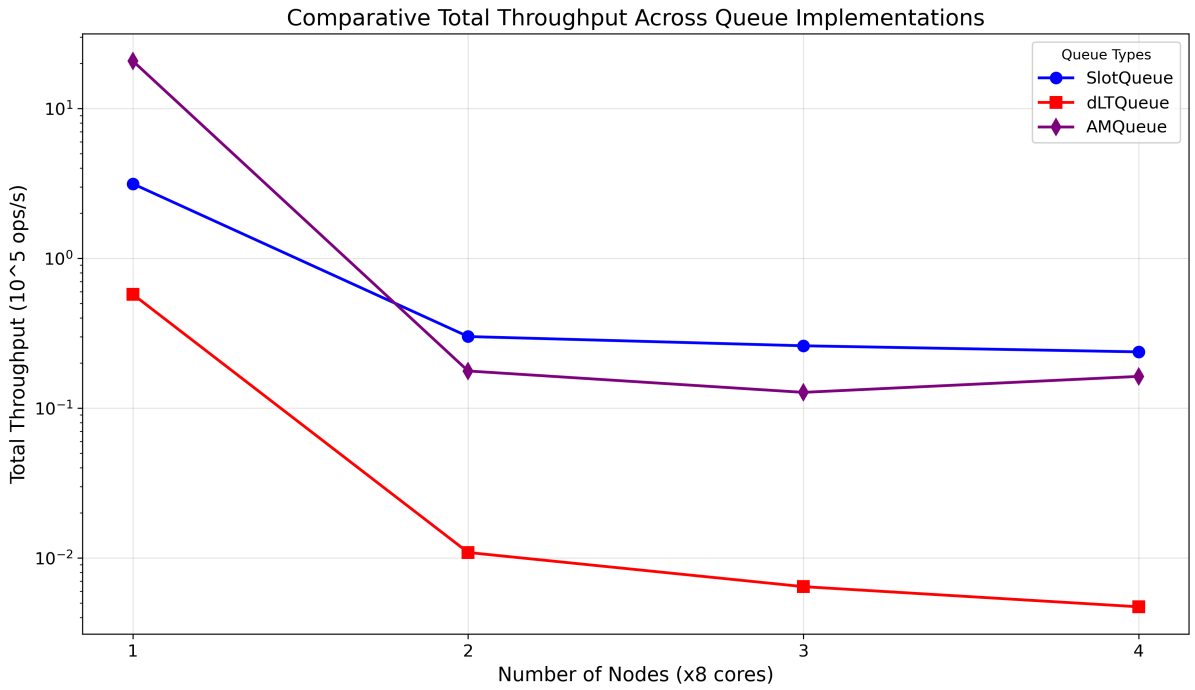


Figure 24: Microbenchmark results for total throughput.

Here is the corrected version of the provided Typst paragraphs with grammatical and spelling mistakes fixed, while keeping line breaks, wordings, and meanings intact:

Chapter VII Benchmarking Results (continued)

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

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

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

In conclusion, based on Figure 24, Slotqueue seems to perform better than dLTQueue and AMQueue in terms of both enqueue and dequeue operations, both latency-wise and throughput-wise. The overhead of a logarithmic-order number of remote

operations in dLTQueue seems to be costly, adversely affecting its performance when the number of nodes increases. Additionally, compared to AMQueue, dLTQueue and Slotqueue also have the advantage of fault tolerance, which, due to the blocking nature of AMQueue, cannot be promised.

Chapter VIII Conclusion

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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