Fast and Space-Efficient Queues via Relaxation

Dempsey Wade
Bucknell University, Lewisburg, PA, USA

Edward Talmage[1]
Computer Science Department, Bucknell University, Lewisburg, PA, USA
edward.talmage@bucknell.edu

Abstract

Efficient message-passing implementations of shared data types are a vital component of practical distributed systems, enabling them to work on shared data in predictable ways, but there is a long history of results showing that many of the most useful types of access to shared data are necessarily slow. A variety of approaches attempt to circumvent these bounds, notably weakening consistency guarantees and relaxing the sequential specification of the provided data type. These trade behavioral guarantees for performance. We focus on relaxing the sequential specification of a first-in, first-out queue type, which has been shown to allow faster linearizable implementations than are possible for traditional FIFO queues without relaxation.

The algorithms which showed these improvements in operation time tracked a complete execution history, storing complete object state at all n processes in the system, leading to n copies of every stored data element. In this paper, we consider the question of reducing the space complexity of linearizable implementations of shared data types, which provide intuitive behavior through strong consistency guarantees. We improve the existing algorithm for a relaxed queue, showing that it is possible to store only one copy of each element in a shared queue, while still having a low amortized time cost. This is one of several important steps towards making these data types practical in real world systems.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms

Keywords and phrases Shared Data Structures, Message Passing, Relaxed Data Types, Space Complexity

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2020.14

Acknowledgements We would like to thank Anh Kieu, Shane Staret, and Jimmy Wei for helping find references.

1 Introduction & Related Work

Because they present the same interface as sequential data types, shared memory objects are a relatively intuitive way to program access to shared data by many processors. Unfortunately, in a distributed computation setting, physical shared memory is usually not possible and processes communicate by sending messages. Programming in a message passing system is more difficult, since there tend to be many messages in transit at once, on many communication links, and their causal and temporal relationships may be masked by variable delays. To hide this difficulty and make distributed programming easier and less error-prone, there is much work on implementing shared memory objects as an abstraction layer on top of message passing systems. As more and more computing moves to distributed and cloud systems, the ability to write programs that interact with shared data in predictable and efficient ways continues to grow, so it is vital that we work to provide the best shared data structure implementations possible.

[1] Corresponding author

Existing work on shared data types has given implementations for both specific and arbitrary data types, but it has also shown that all operations with certain often-desirable properties are inherently slow, requiring a delay proportional to the maximum time a message may take in transit to ensure knowledge of preceding and concurrent operation invocations [8, 11, 6]. In a widely distributed system, such a delay could easily be on the order of hundreds of milliseconds, which is more than enough to negatively impact a human user's experience, and is extremely costly to a computation using such a shared data object.

There are several approaches to circumvent this lower bound, some of which the community has explored for decades and some of which are newer. The lower bounds mentioned before generally apply to *linearizable* implementations, in which it is possible to reduce a concurrent execution to an equivalent sequential one without reordering non-concurrent operations. Weaker consistency conditions, such as *sequential consistency* or *eventual consistency*, do not have the same lower bounds, but they also do not provide the same guarantees [2, 10] or intuitive correspondence to sequential structures. Eventual consistency is widely used in commercial applications but gives very weak guarantees, making it difficult to reason about the expected behavior of interactions with shared data. Even for stronger conditions like sequential consistency, the practical effects of weakened guarantees can be hard to anticipate and seem counter-intuitive, making them less attractive in practice.

Another approach called *relaxation*, developed in [1] and formalized in [5], allows better performance in a linearizable system [5, 9]. By weakening the guarantees of the data type's sequential specification, more possible responses to a particular operation are allowed, and this limited non-determinism can be exploited to eliminate the need for processes to synchronize in every operation instance. Instead, updates can be sent in the background, allowing quick responses and high throughput. Occasional synchronization is necessary, keeping the worst-case time complexity high, but relaxed data types can have a much lower amortized cost per operation than is possible for unrelaxed types.

While [9] proved the possibility of these performance gains, it did so in a theoretical model without many of the difficulties present in real systems. Assumptions of known message delays, free storage, and always-correct processes do not translate well to practical implementations. In this paper, we seek to take a first step towards removing these assumptions by reducing the space required to implement relaxed queues, while still maintaining good time performance. Future work is still necessary to remove other idealized model assumptions and build practical implementations of these types.

The performance gains possible with relaxed data structures, particularly queues, come at the cost of weakened guarantees on the order of data retrieval. For example, the relaxation we primarily consider in this work merely guarantees that some old element in the structure, not necessarily the single oldest, is returned by each *Dequeue* instance. Such a weakening reduces the usefulness of the data type in many cases, since we can no longer be confident which element we get when we retrieve one from the structure. But because the type still provides some ordering, these objects are still of use in applications where response time is more important than exact ordering. For example, consider a distributed job queue, where the primary goal is to execute a large computation as quickly as possible. While we intuitively want to send tasks in exact FIFO order, if they are being completed concurrently, their completion order may not exactly match their start order, so relaxing the start order will not adversely affect the computation. Similarly, online shopping applications, such as for high-demand, limited-run items like concert tickets, demand very quick response time, or customers are left frustrated while they wait to find out whether they were among the first to request a product. By relaxing the order of customers in some cases, the average wait

time may decrease, without any customer waiting longer than they would in an unrelaxed system. Care is necessary to avoid disadvantaging customers who made their request earlier, but this is possible by counting requests. As a tangible example, consider the problem of selling 1000 identical tickets for lawn seating at a concert. The exact order in which requests are processed has no bearing on correctness, as long as the first 1000 customers are those who get the tickets. Since we do not relax *Enqueue* (which already has good performance), we know that we store requests in the order they arrive, and merely need to mark which are the first 1000. With a fast relaxed *Dequeue*, we can then process all the requests more quickly, leading to happier customers and lower load on the ticket servers.

Our Contribution

The algorithm for arbitrary data types in [11], on which the algorithm in [9] for relaxed types is based, keeps a complete copy of the shared data locally on every process and updates these copies based on operations invoked throughout the system. This is highly inefficient, especially for data types like Queues, Stacks, and Heaps in which reading an element also removes it, so only a single process will ever need the value of each stored data object. The space overhead of full replication was necessary for those algorithms to achieve their low time complexity, as they avoided waiting for round-trip messages by having every process simulate the shared object by executing all operation instances on its local copy of the structure.

We consider only linearizable implementations of data types, since they provide the strongest, most intuitive restrictions on concurrent behavior. The idea of only partially replicating data elements has been more thoroughly explored in the context of causal consistency [4, 12, 3] where, despite the weaker consistency condition it is difficult or impossible to store fewer than n copies of the data and maintain consistency.

We show that by exploiting the same properties of relaxation that allow a structure to have lower time cost, we can also reduce storage to only a single copy of each data element in the system. This gives a reduction in space complexity by a factor of n over the existing work. We here present this solution for one particular relaxation of FIFO queues as a proof of concept. For this relaxation, we increase the amortized time complexity of the costly *Dequeue* operation by approximately a factor of 2 over the relaxed queue implementation in [9], but with reasonable levels of relaxation still achieve amortized time below the lower bound for unrelaxed queues. We also show that this is better than is possible in an unrelaxed queue implementation. In the future, we intend to explore other relaxations, where we expect to match the time complexity of the best-known algorithm while still reducing the total space complexity by a factor of n.

Our solution is still somewhat idealized, as we keep assumptions about known message delay bounds and correct processes. We are working separately on fault-tolerant implementations, with the aim of eventually combining improvements along different dimensions. In fact, a real-world solution will probably not want to reduce space complexity quite as far as we do here, since some replication is necessary to prevent data loss in the presence of faults. However, we feel it worthwhile to explore the bounds of possible space savings and the tradeoff of space versus time on their own merits. This helps demonstrate the essential parts of efficient implementations and educates our ongoing work to build practically useful structures.

2 Model & Definitions

We consider sequential data type specifications consisting of two parts: a list of operations, with argument and return types, and a list of legal sequences of invocation-response pairs of those operations. In the sequential setting, an operation's invocation must be immediately followed by its response. In a concurrent setting, the argument and response of an operation may occur at different times. An operation instance is an invocation of an operation, which specifies an argument, together with a corresponding response. We require that the set of legal sequences in a data type specification be prefix-closed and complete, meaning that any prefix of a legal sequence is legal and after any sequence ρ, for any invocation i there must be a response r forming an operation instance (i, r) such that $\rho \cdot (i, r)$ is legal.

We focus on the queue data type, since its ordering properties lend themselves to intuitive relaxations. Specifically, in this paper we implement queues with Out-of-Order k-relaxed *Dequeue*. Informally, this is a FIFO queue in which each *Dequeue*, instead of being required to return and remove the oldest element in the queue, may return and remove any of the k oldest elements. For analysis, we will also refer to the derived parameter $\ell := \lfloor k/n \rfloor$.

▶ **Definition 1.** *A queue with* Out-of-Order k-relaxed *Dequeue provides two operations:*

1. *Enqueue*$(x, -)$ *takes one value* x *as its argument and returns nothing.*
2. *Dequeue*$(-, r)$ *takes no argument and returns one value* r*.*

Let $\perp$ *be a special symbol to indicate an empty queue. The empty sequence is legal and, if* ρ *is a legal sequence,*

- $\rho \cdot$ *Enqueue*$(x, -)$ *is legal for any* $x \neq \perp$ *which is not the argument of an Enqueue in* ρ.[2]
- $\rho \cdot$ *Dequeue*$(-, r), r \neq \perp$*, is legal if* r *is the argument of one of the first* k *Enqueue*(y) *instances in* ρ *s.t. Dequeue*$(-, y)$ *is not in* ρ*.*
- $\rho \cdot$ *Dequeue*$(-, \perp)$ *is legal if there are fewer than* k *Enqueue*(y) *instances in* ρ *s.t. Dequeue*$(-, y)$ *is not in* ρ*.*

We adopt the model of [9]: We consider a system of n processes which can communicate by sending point-to-point messages to each other. This is a partially-synchronous model, where each process has a local clock running at the same rate as real time, but with an unknown offset, and processes know that every message takes between $d - u$ and d real time in transit. We assume that local computation is instantaneous to focus on the communication costs which arise in the algorithm. Each process interacts with a user by allowing them to invoke operations and by providing return values to those invocations. We thus model each process with a state machine whose transitions are triggered by three types of events: message arrival, timer expiration, and operation invocation, and which can set timers, send messages, and/or generate operation responses in each step.

A schedule for each process describes the sequence of states and transitions of its state machine. A run of an algorithm consists of a schedule for each process, where each transition has an associated real time. A run is admissible if the times associated with each process' transitions are monotonically non-decreasing and interaction with each process starts with an operation invocation and then alternates responses and invocations. This prevents a user from invoking an operation until its previous invocation has finished. A run is complete if every message sent is received and each process' schedule is infinite or ends with no timers set. Note that this assumes that all processes are correct and do not crash.

[2] We assume that arguments to *Enqueue* are unique. This can be achieved by another abstraction layer adding tags such as timestamps to elements.

We assume that local clocks have previously been synchronized by an algorithm such as that of [7], which yields an optimal bound of $\epsilon \leq (1 - 1/n)u$ on clock skew, the difference between any two local clocks.

We consider algorithms implementing data type specifications in this message passing model which satisfy a liveness condition, that every operation invocation has a matching response and vice versa, and linearizability, which says that for every complete, admissible run of the algorithm, there is a permutation, called a linearization, of the operation instances in the run which is legal by the data type specification and respects the real-time order of instances which do not overlap in real time. We require algorithms to be eventually quiescent, which means that if users stop invoking operations, every process' schedule will be finite–processes eventually stop setting timers and sending messages.

The time complexity of an operation OP in this model, denoted $|OP|$, is the maximum over all instances in all complete, admissible runs of the real time between the invocation and response of a single instance of that operation. We are also interested in the amortized time complexity of OP, which is the maximum over all complete, admissible runs of the average real time between invocation and response of every instance of OP. We assume that local computation is instantaneous, partly because it is practically much faster than communication time and partly because we are focused on minimizing the cost of communication-related delays.

To measure space complexity of our queue implementations, we introduce the parameter T, which represents the maximum number of data elements concurrently in the queue. That is, in a sequence π of operation instances, T is the maximum over all prefixes ρ of π of the number of *Enqueue* instances in ρ minus the number of *Dequeue* instances in ρ which return a non-$\perp$ value. To focus on the principles of shared data objects, we only measure the amount of data stored, not local variables used for the algorithm or buffers holding unprocessed messages.

3 Lower Bound on Unrelaxed Queues

We begin with a brief argument for the worst-case time complexity of any algorithm for an unrelaxed queue which stores only one copy of each data element. Our algorithm will match this space complexity and worst-case time bound, which shows that our algorithm is not a step backward from an unrelaxed queue with the same space complexity.

▶ **Theorem 2.** *Any algorithm linearizably implementing an unrelaxed queue which stores only one copy of each element must have* $|Dequeue| \geq 2d$.

Proof. Suppose that some algorithm A linearizably implements a queue and only stores one copy of each element. Consider a run in which one process enqueues several elements, then nothing happens until the system is quiescent. Since A stores only one copy of each element, the oldest element *head* is stored at a single process, which we'll call p_h. Suppose that some other process p_d then invokes a *Dequeue*. Since there are no concurrent operation instances, p_d must return *head*. But p_d doesn't know what *head* is, so must retrieve it from p_h. Since the system was in a quiescent state, p_h must wait to hear from p_d before sending *head*, and the upper bound on message delay implies that p_d will not have *head* available to return until up to $2d$ after invocation. ◀

In a system that does not satisfy eventual quiescence, we could prove the same result by showing that if *head* is in transit, a process that is not the recipient of a message carrying *head* can invoke a *Dequeue* and it will still need to wait up to $2d$ time before receiving *head*.

The exception, and why we restrict ourselves to the domain of eventually quiescent algorithms, are algorithms that effectively use the message channels for memory. By broadcasting all elements immediately upon reception, instead of storing them, such an algorithm could prevent a *Dequeue* instance from needing a long delay. This would lead to an unconscionably large message complexity, even in the absence of activity on the data structure.

4 Algorithm

Our algorithm is a modification of that in [9], as we want to maintain its improvements in time complexity. That algorithm uses a system of timers to ensure that all processes execute all invocations on their local copies of the queue in the same order. This, coupled with deterministic execution, ensures that all processes maintain the same state and can provide consistent and correct return values. To enable the majority of *Dequeue* instances to return after only local computation, the algorithm used an element-claiming system to divide the k values which were possibilities for a legal *Dequeue* return value among the n processes. As long as a process had a claimed element when a *Dequeue* invocation arrived, the algorithm responded quickly to the user and coordinated with other processes in the background. When a process ran out of claimed elements, the next *Dequeue* invocation was forced to wait until the process was sure its local copy of the queue was up to date to claim ownership of more elements and generate a *Dequeue* response.

We want to avoid having a complete copy of the shared state at every process, so we add mechanisms for determining which process stores each element. This is a two-stage system, with each element initially stored at one process, but then moved to a (potentially) different process which claims it. This transfer is necessary for a process to be able to return its claimed elements without waiting for communication with other processes, enabling the common case of most *Dequeue* instances returning without waiting for communication.

4.1 Description

We first give an intuitive description of our algorithm's behavior. The algorithm is event-driven, where possible events are operation invocation, message arrival, and timer expiration. Recall that we assume local computation is instantaneous, so events cannot interrupt other event handlers.[3]

When a user invokes an operation at a particular process p_i, the appropriate handler (lines 1-4 for *Enqueue*, 5-11 for *Dequeue*) will announce the invocation to every process, including p_i. When each process receives such an announcement of an invocation *op*, the message handler in lines 15-20 sets a timer to wait $u + \epsilon$ time (u to account for variation in message delay and ϵ to correct for clock skew) ensuring that it receives all invocations with smaller timestamps. The process will then locally execute, in increasing timestamp order, all invocations with smaller timestamps, ending with *op*, via the while loop in lines 21-25. In [9], this guaranteed that all processes follow the same sequence of local operation executions, allowing them to keep their local views of the shared queue synchronized. In our algorithm, we do not store the full state of the shared queue at each process, so cannot make as strong a claim. Instead, we track the number of each type of operation (modulo n) and current size of the simulated queue, which enables us to determine which process should store and retrieve the data elements involved in each *Enqueue* or *Dequeue* instance, respectively.

[3] We name functions in the pseudocode based on how they may be called: **HandleEvent**s respond to external events, while **Function**s are called internally.

We can respond to certain operation invocations before their execution is complete at all processes, as soon as we know the correct return value. The algorithm will propagate the instance's effects in the background to ensure correct state. All *Enqueue* instances can thus return quickly, since they have no return value. Similarly, *Dequeue* instances can quickly return one of a process' claimed elements, if there are any. Lines 12-14 generate these fast responses. When a process runs out of claimed elements, however, quick returns to *Dequeue* invocations are not possible, dividing *Dequeue* instances into two types, fast and slow. For slow *Dequeue*s, the invoking process must claim new elements, which requires waiting until it knows about all preceding operation instances so that all processes agree which elements it claims. This occurs when processes locally execute the slow *Dequeue* instance, which sends the newly-claimed element to the invoking process as a *restock* message. Once the invoking process receives a *restock*, it knows it has an element that is a correct return value for the *Dequeue* instance and will not be returned by any other process, so the slow *Dequeue* can return to the user. The algorithm does this in lines 39-53 by a similar logical structure as that which ensures local execution of all instances in timestamp order. Here, we ensure that *restock* elements are claimed, or returned by slow *Dequeue* instances, in the correct order.

There are two primary improvements in this algorithm over that of [9]. First, and central to this paper's result, we note that when processes receive an announcement of a new *Enqueue* instance, they do not all need to store a copy of the argument. Instead, we separate *stored* elements from those which processes have *claimed*. Only one process saves the new element, putting it in a local *stored* queue. When a process claims that element, the storing process can send and delete it, since it will be saved in the claiming process' *claimed* queue.

Using the number of *Enqueue* instances which have happened so far, the algorithm distributes enqueued elements in a round-robin fashion to achieve balanced storage. When processes claim elements and remove them from storage, we similarly remove them in round-robin order using the saved number of *Dequeue* instances, which guarantees a FIFO ordering of *stored* elements across all processes. Note that this order does not hold for *Dequeue* return values, since a process can claim elements, then sit idle while other processes remove elements added to the queue since its claimed elements.

Our second improvement is the restocking procedure: when a process invokes a *Dequeue*, its announcement of that invocation also serves as a request to claim a new element. If the invoking process has no claimed elements, it must wait for the new element to arrive from the process storing it. If the invoking process has claimed elements, restocking occurs in the background, with the effect that if *Dequeue* invocations are not too frequent at any one process, all *Dequeue* instances in a run could be fast. This restocking system increases the worst-case time of a *Dequeue* to approximately $2d$, where the original algorithm had a worst-case time of approximately d, but this tradeoff is necessary to reduce our storage requirements. As detailed later in the paper, we still have a lower amortized time complexity than is possible without relaxation.

Pseudocode for our relaxed queue is in Algorithms 1 and 2. It uses local FIFO queues *claimed* and *stored* and min-priority queues *Pending* and *Restocks*, which are keyed on the timestamps (lexicographically-ordered pairs containing local clock values and process ids) of the instances they store.

4.2 Correctness

To prove our algorithm is a correct, linearizable implementation of a queue with Out-of-Order k relaxed *Dequeue*, we will show that every invocation has a response, then construct a linearization of those instances based on the timestamps assigned when they are invoked, and show that every return value is legal by the data type specification.

Algorithm 1 Pseudocode for each p_i implementing a Queue with Out-of-Order k-relaxed *Dequeue*.

```
1:  HandleEvent ENQUEUE(val)
2:      ts = ⟨localtime, i⟩
3:      send (enq, val, ts) to all
4:      setTimer(ϵ, ⟨enq, val, ts⟩, respond)
5:  HandleEvent DEQUEUE
6:      ts = ⟨localtime, i⟩
7:      val = claimed.dequeue()
8:      if val ≠ ⊥ then
9:          send (fastDeq, val, ts) to all
10:         setTimer(ϵ, ⟨fastDeq, val, ts⟩, respond)
11:     else send (slowDeq, ⊥, ts) to all
12: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, respond)
13:     if op == fastDeq then Generate Dequeue response with return value val
14:     else  Generate Enqueue response with no return value
15: HandleEvent RECEIVE (op, val, ts) FROM p_j
16:     if op ∈ {fastDeq, slowDeq} then
17:         Restocks.insert(⟨op, val, ts⟩)
18:         setTimer(d + 2u + ϵ, ⟨op, val, ts⟩, restock)
19:     Pending.insert(⟨op, val, ts⟩)
20:     setTimer(u + ϵ, ⟨op, val, ts⟩, execute)
21: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, execute)
22:     while ts ≥ Pending.min() do
23:         ⟨op′, val′, ts′⟩ = Pending.extractMin()
24:         executeLocally(op′, val′, ts′)
25:         cancelTimer(⟨op′, arg′, ⟨t, j⟩⟩, execute)
26: Function EXECUTELOCALLY(op, val, ts)
27:     if op == enq then
28:         if enqueueCount == i then
29:             if clean and size < k then
30:                 claimed.enqueue(val)
31:             else stored.enqueue(val)
32:         enqueueCount += 1 (mod n)
33:         size += 1
34:     else
35:         if restockCount == i then  send (restock, stored.dequeue(), ⟨op, val, ts⟩) to p_j
36:         restockCount += 1 (mod n)
37:         size −= 1
38:         clean = (size == 0)
39: HandleEvent RECEIVE (restock, restockVal, ⟨op, val, ⟨t, i⟩⟩) FROM p_j
40:     Restocks.update(⟨op, val, ⟨t, i⟩⟩, restockVal)
41: Function EXPIRETIMER(⟨op, val, ts, restockVal⟩, restock)
42:     while ts ≥ Restocks.min() do
43:         ⟨op′, val′, ts′, restockVal′⟩ = Restocks.extractMin()
44:         EXECUTERESTOCK(op′, val′, ts′, restockVal′)
45:         cancelTimer(⟨op′, arg′, ts′, restockVal′⟩, restock)
```

Algorithm 2 Algorithm 1, continued.

```
46: Function ExecuteRestock(op, val, ⟨∗, j⟩, restockVal)
47:     if op == fastDeq and j == i then
48:         claimed.Enqueue(restockVal)
49:     if op == slowDeq and j == i then
50:         returnVal = claimed.Dequeue()
51:         if returnVal == ⊥ then returnVal == restockVal
52:         else claimed.Enqueue(restockVal)
53:         Generate Dequeue response with return value returnVal
```

Let R be an arbitrary complete, admissible run of the algorithm. We assume that if multiple events happen at the same process at exactly the same real time, message receptions occur before timer expirations, but events of the same type may occur in any order. An operation invocation's *timestamp* is the value of the variable ts defined in line 2 or 6 for *Enqueue* or *Dequeue* invocations, respectively.

We omit the proofs for Lemmas 3 and 4 for the sake of space, since they are fundamentally the same as proofs in [9].

▶ **Lemma 3.** *Each operation invocation in R causes exactly one response.*

This defines the set of operation instances in R, by pairing each invocation with the resultant response. We say that an operation instance's timestamp is that of its invocation.

▶ **Lemma 4.** *Every process locally executes every operation instance exactly once, in timestamp order.*

▶ **Construction 1.** Let π be the sequence of all operations instance in R, sorted by timestamp order.

▶ **Lemma 5.** *π respects the order of non-overlapping operation instances in R.*

Proof. Suppose in contradiction that op_2 responds before op_1's invocation, but $ts(op_1) < ts(op_2)$, so op_1 precedes op_2 in π. Every operation instance takes at least ϵ time to respond, by the timers in lines 4 and 10. Thus, op_2's invocation must be at least ϵ real time before op_1's. But local clocks are skewed by at most ϵ, so $ts(op_2)$ must be less than or equal to $ts(op_1)$, contradicting our assumption and proving the claim. ◀

▶ **Lemma 6.** *At any time, there are no more than k elements in the union of all processes' claimed queues and the set of restock messages in transit.*

Proof. We observe that there are only two ways that elements can be added to a *claimed* queue. First, in a *clean* state, which means that there have been no *Dequeue*s since the queue was last empty, *Enqueue* instances can add their arguments directly to *claimed* queues in line 30. This cannot cause there to be more than k elements in all processes' *claimed* queues, by the check in line 29.

Second, we add elements to *claimed* when restocking after a *Dequeue* instance, in lines 48 and 52. Elements are only added to *claimed* after removing an element from that process' *claimed*, either in line 7 or line 50, no *Dequeue* instance can increase the size of any process' *claimed* queue above the maximum size of that queue set by *Enqueue* instances.

The only time an element is sent in a *restock* message is after a *Dequeue* instance. If that instance was fast, then it removed a *claimed* element, so sending the *restock* message does not increase the number of *claimed* or *restock*ing elements. If the *Dequeue* instance

was slow, then the invoking process had no *claimed* elements. Sending the *restock* message could increase the total number of elements claimed or in transit, but since the invoking process' *claimed* queue was empty, $k \geq n$, and *Enqueue* instances which add to *claimed* do so in round-robin fashion, this means that there were previously fewer than k elements in the union of all *claimed* queues and in-transit *restock* messages, so there are still fewer than k.

Thus, the total number of elements in all processes' *claimed* queues and all in-transit *restock* messages will be less than or equal to k. ◀

▶ **Lemma 7.** *For any prefix ρ of the sequence π defined in Construction 1, after locally executing ρ, every process' size and clean variables will have the same values.*

Proof. We first note that both *size* and *clean* are only edited in the function executeLocally, so we restrict our attention to that function and prove this lemma by induction on $|\rho|$, the length of the prefix ρ. When $|\rho| = 0$, all processes' variables hold their initial value of $clean = true$ and $size = 0$.

Assume that after locally executing a prefix ρ' of length k, the claim holds. Then when any process locally executes the next operation instance *op* in π, it will follow the same logic, since executeLocally is deterministic and all processes have the same parameters, since they are executing the same operation instance, they will set *clean* to the same value and change *size* in the same way. The only differences in behavior that may occur at different processes are the results of the process id checks in lines 28 and 35, which do not have any effect on the values of *clean* or *size*. Thus, after executing a prefix of length $k + 1$, every process will have the same values for its *clean* and *size* variables. ◀

▶ **Lemma 8.** *At any time, for any element c in any process' claimed queue or in-transit restock message and any element s in any process' stored queue, c was the argument of an Enqueue instance which appears in the sequence π defined in Construction 1 before the Enqueue instance with s as argument.*

Proof. Suppose in contradiction that an element x in some process p_i's *claimed* queue was the argument of an *Enqueue* instance $enq = Enqueue(x)$ that appears in π after another instance $enq' = Enqueue(y)$, where y is in some process p_j's *stored* queue (note that i and j may not be distinct). As before, there are two possible ways the algorithm may have put x in p_i's *claimed* queue: directly by *enq* in line 30 or as a restock for a *Dequeue* instance in line 48 or 52.

Suppose first that x was added directly to p_i's *claimed* queue by *enq*. Then when each of p_i and p_j locally executed *enq*, by line 28 and Lemma 7 we know that *clean* was true. Thus, p_j would only have put y in *stored* if either *clean* was false when p_j locally executed enq' and changed to true before p_j locally executed *enq*, or if *size* was at least k when p_j locally executed enq' but less than k when p_i locally executed *enq* (or both). *clean* could not have been false when processes locally executed enq' and true when they locally executed *enq* without y having been removed from *stored* and returned by a *Dequeue* in between, since *clean* is only set to true when there are no elements left in the queue, by Line 38. On the other hand, if *size* was at least k when p_j locally executed enq', but was not when p_i locally executed *enq*, there must have been a *Dequeue* instance between enq' and *enq* in π, since *size* only decreases in line 37. But when each process locally executed that *Dequeue* instance, they would have set *clean* to false in line 38, so when p_i locally executed *enq*, it would not have stored x in *claimed*, unless *clean* was reset to true between the local execution of the *Dequeue* instance, which we have already argued could not happen. Thus, x cannot have been added to p_i's *claimed* by *enq*.

The other possible way for x to be in p_i's *claimed* queue is for it to have been put in some process p_k's *stored* queue and passed to p_i as a restock element for a *Dequeue* instance *deq*. But, since all processes locally execute all instances in the same order, restocks are taken from storage in the order in which they were added (proving the claim for elements in in-transit *restock* messages), and y would be added to *stored* before x, by Lemma 4, we can conclude that y was removed from p_j's *stored* queue by an instance deq' with a lower timestamp than that which caused x to be removed. Because the instance deq' removing y from *stored* had a lower timestamp than *deq*, it must have been locally executed within $d + u + \epsilon$ time after the invocation of *deq*, at the latest when the timer set in line 18 (upon arrival of the message containing *deq*) expires. However, no process can call executeRestock for *deq* and add x to claimed until the timer on line 18 expires for *deq* or another instance with larger timestamp. Such an instance can be invoked at most ϵ real time before *deq*, and the message sent at its invocation must take at least $d - u$ times. Thus, this instance's *restock* timer can expire no earlier than $(d - u) + (d + 2u + \epsilon) - \epsilon = 2d + u$ real time after the invocation of *deq*. Thus, p_j must have locally executed deq' and removed y from *stored* at least $(2d + u) - (d + u + \epsilon) = d - \epsilon > 0$ real time before p_i added x to *claimed*, contradicting our assumption.

Thus, for any x enqueued by an instance with smaller timestamp than that enqueueing y, x cannot be in any process' *claimed* queue or an in–transit *restock* message while y is in another process' *stored* queue, and we have the claim. ◀

▶ **Lemma 9.** *The sequence π defined in Construction 1 is legal by the specification of a queue with Out-of-Order k-relaxed Dequeue.*

Proof. We prove this by induction on the length of a prefix ρ of π. The empty sequence is legal, proving the base case. Suppose now that $\rho = \sigma \cdot op$ and σ is a legal sequence. Denote the process invoking *op* as p_h.

- **Case 1:** $op = Enqueue(arg, -)$**.** ρ is legal by the type specification.
- **Case 2:** $op = Dequeue(-, retVal), retVal \in V$**.** We consider the cases of fast and slow *Dequeue* instances separately:
  - Suppose *op* is a $fastDeq$ instance. Then its return value is chosen from p_h's *claimed* queue. By Lemmas 6, there are no more than k elements in all processes' *claimed* queues and in-transit *restock* messages. By Lemma 8, all claimed elements and those carried by in-transit *restock* messages are the arguments of the earliest unmatched *Enqueue* instances in the prefix of π before *op*. Thus, at *op*'s invocation $retVal$ is the argument of one of the first k unmatched *Enqueue* instances.

    Every *Dequeue* instance's return value in a *claimed* queue is removed as soon as it is chosen (lines 7, 50) and p_h removes $retVal$ from its *claimed* queue during *op*'s invocation. Also, no element is ever in more than one *claimed* queue, by the checks on lines 48 and 52 and the fact that elements are removed from *stored* when sent to claimed (line 35). Thus, no other *Dequeue* will return $retVal$. Further, once an *Enqueue* instance is among the first k unmatched, it will continue to be until it is matched, so ρ is legal.
  - Suppose *op* is a *slowDeq* instance. If *op* chooses its return value in line 50, then Lemmas 6 and 8 show that $retVal$ is the argument of one of the first k unmatched *Enqueue* instances in σ, so ρ is legal. If *op* chooses its return value in line 51, then that value was carried by a *restock* message. As discussed before, that means it was the argument of one of the first k unmatched *Enqueue* instances, and thus ρ is legal.

- **Case 3:** $op = Dequeue(-, \perp)$**.** For a *Dequeue* instance to return $\perp$, then it must have chosen its return value in line 51, and have received a $\perp$ as a restock element. For the *restock* message to have been carrying $\perp$, then it must have come from a process with an empty *stored* queue, in line 35. This means that when processes locally executed this *Dequeue* instance, there were no elements in any process' *stored* queue, since elements are removed from *stored* queues in the order in which they were added. This means there were fewer than k unmatched *Enqueue*s, since processes locally execute all instances in the order given by π, by Lemma 4 and there are fewer than k elements in *claimed* elements and *restock* messages–which are the only other places values can be–by Lemma 6. Thus, by the specification of a queue with Out-of-Order k-relaxed *Dequeue*, ρ is legal. ◀

▶ **Theorem 10.** *Algorithm 1 is a correct, linearizable implementation of a queue with Out-of-Order k-relaxed Dequeue.*

Proof. By Lemma 5, π is an ordering of all operation instances which respects the real time order of non-overlapping instances and by Lemma 9, π is legal. Thus, for any run R of Algorithm 1, there is a linearization of R, and we have the claim. ◀

4.3 Complexity

4.3.1 Time

When discussing time complexity, we are interested in the time the algorithm takes to respond to operation invocations, in terms of the system's message timing parameters. With relaxation, we can have many *Dequeue* instances return much faster than the worst-case, leading to a low average cost for *Dequeue*. Thus, we also measure the amortized, or worst-case of the average, time required for *Dequeue*. We do not consider the amortized cost of *Enqueue* since every instance takes the (low) worst-case time.

One additional wrinkle in measuring the time complexity is that the mechanism for accelerating fast *Dequeue* instances depends on having a significant number of elements in the queue at all times. This would be the most common use case, and the number of fast *Dequeue*s, and thus average performance, scales cleanly with the size of the queue, but makes general analysis difficult. We thus present bounds for the heavily-loaded case, where there are consistently at least k elements in the queue. In more lightly loaded scenarios, where there are fewer than k elements to distribute, the algorithm behaves as if k was decreased–the structure is less relaxed. Practically, the enqueue elements are distributed evenly among all processes, and they can dequeue those quickly before trying to claim more. Since *Enqueue* instances only claim elements while the structure is clean, this is the same as if k was the size of the queue at the first *Dequeue* instance until the queue is clean again. This means that relaxation scales cleanly with the queue's size when the first *Dequeue* instance occurs, up to k.

Finally, we observe that our restocking mechanism gives the possibility of much better average performance than the worst case represented in the amortized cost. A *Dequeue* instance is fast if its invoking process has claimed elements available, so slow *Dequeue* instance occur when many fast *Dequeue*s in a row deplete the process' stock of claimed elements. Because we restock in the background, if *Dequeue* invocations are sufficiently infrequent, then a process will never run out of claimed elements, and all *Dequeue* instances will be fast. Exactly how infrequent *Dequeue* invocations must be for this to occur depends on the system parameters, but the average time for each would be the same as the amortized cost of the mix of fast and slow *Dequeue*s which results from invoking them continuously. Thus, for the common case when many *Dequeue*s are not invoked immediately one after another, a process will experience only *Dequeue* instances with low response times.

▶ **Theorem 11.** *The worst-case operation times for Algorithm 1 are ϵ for Enqueue and $2d + u + \epsilon$ for Dequeue.*

Proof. *Enqueue* instances and *Dequeue* instances at a process which currently has claimed elements are fast operations, and respond ϵ time after invocation, by the timers set in lines 4 and 10 and handled in lines 12-14, while their effects are propagated through the system in the background. *Dequeue* instances at processes which do not have any claimed elements are slow, and cannot respond until the process can coordinate with other processes to claim an element. Such a *slowDeq* instance must wait for its announcement message to arrive at the process which holds the next stored element (in FIFO order), up to $u + \epsilon$ time (line 20) for that process to ensure that it is executing instances in timestamp order, and a second message delay for that process to send the newly-claimed element back (line 35). The *slowDeq* instance can then return either the newly claimed element or one its invoking process claimed in the background execution of another operation instance while it was waiting. This delay is managed by the timer set in line 18, which starts after the $d - u$ delay for a message from the invoking process to reach itself. Thus, the total worst-case time complexity for *Dequeue* is $2d + u + \epsilon$. ◀

This worst-case cost is higher than the $d + \epsilon$ achievable in unrelaxed queues [11], but slow *Dequeue* instances are relatively infrequent, so we still obtain a low amortized cost. More concerning is the fact that we have more than doubled the worst-case cost from the algorithm for queues with out-of-order k-relaxed *Dequeue* in [9]. This is the tradeoff for reducing space complexity, and is unavoidable since at least some *Dequeue* instance must retrieve its return value from another process, taking minimum of $2d$ time.

For amortized response time, consider a *heavily-loaded* run, defined as a run which starts with at least k *Enqueue* instances, after which there are never fewer than k unmatched *Enqueue* instances. Because there have been no *Dequeue* instances so far, when processes locally execute each of the initial k *Enqueue* instances, the check in line 29 will pass, and one process will the enqueued element. Thus, there will be k claimed elements, evenly distributed among the processes, when the first *Dequeue* is invoked. Since we assume $k \geq n$, this means that the first *Dequeue* will be fast, as will subsequent *Dequeue* instances until some process fast *Dequeue*s ℓ elements and empties its *claimed* queue. If that process invokes another *Dequeue* before restocking, it will be a slow *Dequeue* instance, and take the worst-case time of $2d + u + \epsilon$. By the time that slow *Dequeue* instance returns, all restocking for previous fast *Dequeue* instances will be complete, and the process will again have ℓ claimed elements. In a heavily-loaded run, this pattern is the worst-case for every process, since there will always be elements to restock processes' *claimed* queues. Thus, at most one in every ℓ *Dequeue* instances will be slow (recall $\ell = \lfloor k/n \rfloor$) and the amortized time complexity of *Dequeue* is at most $\frac{2d+u+\epsilon+(\ell-1)\epsilon}{\ell} = \frac{2d+u}{\ell} + \epsilon$.

▶ **Theorem 12.** *The amortized time complexity of Dequeue in Algorithm 1 in a heavily-loaded run is $\frac{2d+u}{\ell} + \epsilon$.*

[9] gives a lower bound of $d(1 - 1/n)$ for the amortized complexity of unrelaxed *Dequeue*, so for $\ell > 3$, our algorithm is faster than an unrelaxed queue, while using a factor of n less space. We also note that existing algorithms for more complex relaxations already have a similar $2d$ term in their amortized cost and expect that we will be able to extend the benefits of this paper's work to such relaxations without significantly increasing the time cost.

4.3.2 Space

For space complexity, because we use a round-robin method to evenly distribute stored elements across the various processes, each process only has to hold $(1/n)^{th}$ of the elements stored at any time. To present this in a way useful for building a system or determining whether one has the capability necessary to participate in this algorithm, we can phrase this in terms of the maximum size of the queue at any point in a run, T.

▶ **Theorem 13.** *Our algorithm requires memory for at most $T/n + k/n + O(1)$ elements at any process at a time, excluding message buffers.*

Proof. No more than k elements are in the union of all processes' *claimed* queues at a time, by Lemma 6. These elements are evenly distributed by the round-robin procedure for claiming elements at enqueue time, or restocking elements removed from that balanced state. Thus, each process stores up to $\lfloor k/n \rfloor + 1$ claimed elements. The restocking procedure for *Dequeue* instances attempts to keep each process at this level, but in the event of many *Dequeue* invocations in close succession, the claimed elements could be depleted, leaving all elements in storage. Because elements are stored and removed from storage in a FIFO, round-robin fashion, the number of elements stored at each process will differ by at most 1. This follows from the fact that if there have been E *Enqueue* instances and D *Dequeue* instances, each process will have stored at most $\lfloor E/n \rfloor + 1$ elements and removed at least $\lfloor D/n \rfloor$ of those, leaving it with less than or equal to $\frac{E-D}{n} + 1$ elements in *stored*. Since $T = E - D$, we have at most $\lfloor T/n \rfloor + 1$ stored elements at each process for a total of at most $T/n + k/n + 2$ elements in memory.

This analysis is slightly oversimplified since processes, while they locally execute all operation instances in the same order, may not do so at the same time. It is thus possible that some process has not yet locally executed a *Dequeue* instance and removed an element from storage at the real time when an *Enqueue* instance with later timestamp adds its argument to another process' storage. This leads to at most a constant number of additional stored elements per process, however, as any one process cannot store elements from *Enqueue*s with later timestamps until it has completed locally executing all previous operations. Thus, there can be at most $n - 1$ of these extra elements, each stored at a different process, before the delayed local execution of a *Dequeue* removes an element from storage. This is less than 1 extra element stored at each process, preserving the bound. ◀

This reduction by a factor of n makes shared queues far more practical for computing devices with limited memory, as well as much more attractive to users of larger systems since a process needs only store elements which it expects to return and a share of elements which it may yet use. In a particular system, it may be possible to use a more finely-tuned storage strategy to demand more storage from processes with more resources, but we here treat the general case by balancing the load evenly.

Extending this algorithm to keep more than one copy of each data element, which would be an important component of fault-tolerance, would be simple, as we would just alter the logic in line 28 so that more than one process stored each element and in line 35 to make sure that all copies were removed from storage. The logic could be tuned to provide as much or little replication as desired, but we leave the details to ongoing work that treats all the concerns of failure tolerance.

5 Conclusion and Ongoing Work

We have shown that relaxing a data type can not only enable faster implementations of that type, but also reduce the type's space complexity. This is a large step towards making these data types practical in real-world systems. Now that we have shown the possibility of reducing the space complexity to a single stored copy of each data element, we know we have the flexibility to replicate them as many times as we may need for fault tolerance or resilience to poor message delays.

One other approach which could provide this level of space efficiency is a system with a centralized storage server, which coordinates requests and allocates elements to all processes. Traditionally, the two primary drawbacks to such an approach are increased communication time, since a round trip is required, and lack of fault tolerance, as a loss of the central server is a loss of all data. The first complaint does not apply, since we also require a round-trip delay for slow *Dequeue*s. Since we assume no failures, we do not directly need fault tolerance. This work is not our end goal, though, as we are working towards fault-tolerant implementations, so we want to avoid structures which will make that extension more difficult.

Our end goal is to obviate all of the model assumptions that are unrealistic, eventually yielding a practical implementation of our shared queue implementation. This will involve not only space efficiency, but fault-tolerance, independence from exact knowledge of message bounds, and accounting for local computation time, at a minimum. We have addressed one of these dimensions here, and are working on the others independently. We also hope to generalize and extend results to more relaxations of more data types, creating a large collection of efficient shared data structures to aid developers of distributed systems.

Specifically, the *lateness* and *restricted Out-of-Order* relaxations of *Dequeue* are natural targets. Lateness corrects the issue with Out-of-Order relaxations that a single element at the head of the queue may be starved indefinitely by requiring that one in every k *Dequeue* instances returns the oldest element in the queue, but places no restrictions on the return values of other *Dequeue* instances. Restricted Out-of-Order combines the lateness and Out-of-Order relaxations, requiring that every *Dequeue* returns one of the k elements which were oldest when the queue's head was last returned, which means every removed element is near the head and at least one in every k *Dequeue* instances returns the head. A space-efficient implementation of a queue with a lateness k-relaxed *Dequeue* does not need to claim elements, only to coordinate removal of the head. An implementation of a queue with restricted Out-of-Order k-relaxed *Dequeue* would combine that coordination system with the claiming system of this paper's algorithm. Both implementations should be possible with the same time complexity as we achieve for the Out-of-Order relaxation.

References

1 Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings*, volume 6490 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2010. doi:10.1007/978-3-642-17653-1_29.

2 Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, 1994. doi:10.1145/176575.176576.

3 Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. In *7th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2015, Vancouver, BC, Canada, November 30 - December 3, 2015*, pages 282–289. IEEE Computer Society, 2015. doi:10.1109/CloudCom.2015.81.

4 Jean-Michel Hélary and Alessia Milani. About the efficiency of partial replication to implement distributed shared memory. In *2006 International Conference on Parallel Processing (ICPP 2006), 14-18 August 2006, Columbus, Ohio, USA*, pages 263–270. IEEE Computer Society, 2006. doi:10.1109/ICPP.2006.15.

5 Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 317–328. ACM, 2013. doi:10.1145/2429069.2429109.

6 Martha J. Kosa. Time bounds for strong and hybrid consistency for arbitrary abstract data types. *Chicago Journal of Theoretical Computer Science*, 1999, 1999. URL: http://cjtcs.cs.uchicago.edu/articles/1999/9/contents.html.

7 Jennifer Lundelius and Nancy A. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2/3):190–204, 1984. doi:10.1016/S0019-9958(84)80033-9.

8 Marios Mavronicolas and Dan Roth. Linearizable read/write objects. *Theoretical Computer Science*, 220(1):267–319, 1999. doi:10.1016/S0304-3975(98)90244-4.

9 Edward Talmage and Jennifer L. Welch. Improving average performance by relaxing distributed data structures. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 421–438. Springer, 2014. doi:10.1007/978-3-662-45174-8_29.

10 Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009. doi:10.1145/1435417.1435432.

11 Jiaqi Wang, Edward Talmage, Hyunyoung Lee, and Jennifer L. Welch. Improved time bounds for linearizable implementations of abstract data types. *Information and Computation*, 263:1–30, 2018. doi:10.1016/j.ic.2018.08.004.

12 Zhuolun Xiang and Nitin H. Vaidya. Partially replicated causally consistent shared memory: Lower bounds and an algorithm. In Peter Robinson and Faith Ellen, editors, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, pages 425–434. ACM, 2019. doi:10.1145/3293611.3331600.