DREU Research

Alyssa Byrnes, Graduate Mentor: Edward Talmage, Faculty Mentor: Jennifer Welch

August 26, 2015

Abstract

Cloud computing has increased interest in implementing shared data objects in message-passing distributed environments. An important feature of these systems is Linearizability. Linearizability can be an expensive consistency condition to implement. Previously, linearizable message-passing implementations of relaxed queues have been considered and proven to improve the average time complexity from that of a standard FIFO queue. This paper expands on this idea and presents an algorithm that implements a k-relaxed lateness queue in a message-passing system of n processes that tightens the upper bound for the average time complexity of operations in such a system.

1 Introduction

As cloud computing has gained more ground in the field of computer science, researchers have an increased interest in finding accurate, efficient ways to implement shared data objects in message-passing distributed environments. Linearizability provides a valuable correctness condition for such systems by expressing potentially concurrent operations as discrete, linear events that occur in the same order at all processors. However, linearizability can be expensive to implement, in both message-passing [10, 4] and shared memory [3], as processors must communicate somewhat frequently in order to maintain synchronization of events.

Our approach reduces the average cost of operations in a message-passing system while still maintaining linearizability. This is done by considering "relaxed" versions of classic data types. [1] and [6], introduced and formalized this concept.

In [13], Talmage and Welch reiterated four different kinds of relaxed queues introduced by Henzinger, Kirsch, Payer, Sezgin, and Sokolova ([6]): an Out-of-Order queue, a Restricted Out-of-Order queue, a Lateness queue, and a Stuttering Queue and discuss the simulation of such queues in a message-passing system. Talmage and Welch then provided algorithms for both the Out-of-Order queue and the Restricted Out-of-Order queue in a message-passing system. Finally, they compared runtimes of dequeue operations in these queues to the traditional FIFO queue in the following table:

Bounds on *Dequeue* Time Complexity from [13]

	Worst Case Cost		Average Cost	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
FIFO Queue	$d + \min\{\epsilon, u, \frac{d}{3}\}$ [12]	$d + \epsilon$ [12]	$d(1 - \frac{1}{n})$	$d + \epsilon$ [12]
Lateness	d	$d + \epsilon$ [12]	?	$\frac{d}{\lfloor k/n \rfloor} + \epsilon$
Restricted-Out-of-Order	d	$d + \epsilon$ [12]	$\frac{d}{\lfloor k/n \rfloor}$	$\frac{2d+\epsilon}{\lfloor k/n \rfloor} + \epsilon$

This paper mainly focuses on the upper bound for the average cost of the dequeue operation of the lateness queue. In [13], the upper bound is said to be $\frac{d}{\lfloor k/n \rfloor} + \epsilon$, but this in fact was not proven in the paper. This paper will prove that this Upper Bound is in fact correct.

First, we reiterate the conditions of a k-relaxed Lateness Queue. Next, we provide an algorithm that implements this data structure in a message-passing system that is linearizable, and finally we prove that this algorithm is correct and the upper bound for the average cost of the dequeue operation is in fact $\frac{d}{\lfloor k/n \rfloor} + \epsilon$.

1.1 k-relaxed Lateness Queues

The definition of a k-relaxed lateness queue builds off of the definition of a FIFO queue. In a FIFO queue, the dequeue operation removes and returns the element that was enqueued at the earliest time and has yet to be dequeued, called the *head*. For a k-relaxed lateness queue, the head does not need to be removed every time dequeue is called, but rather it must be removed in no more than k dequeues. The following are the formal conditions of a k-relaxed lateness queue.

(C1) Every argument to an instance of *Enqueue* is unique.

(C2) Every non-$\perp$ return value of an instance of *Dequeue* is unique.

(C3) Every non-$\perp$ value which an instance of *Dequeue* returns is the argument to a previous instance of *Enqueue*.

(C4) If ρ is a legal sequence of operation instances, then $\rho \cdot Dequeue(-, val)$ is legal iff every $Enqueue(val', -)$ preceding $Enqueue(val, -)$ has a matching $Dequeue(-, val')$ in ρ or there are fewer than k instances $Dequeue(-, val')$ that follow the first $Enqueue(val'', -)$ which does not have a matching $Dequeue(-, val'')$ in ρ.

Further, $\rho \cdot Dequeue(-, \perp)$ is legal iff there are fewer than k instances $Dequeue(-, val')$ that follow the first $Enqueue(val'', -)$ without a matching $Dequeue(-, val'')$ in ρ or every val' such that $Enqueue(val', -)$ is in ρ has a matching $Dequeue(-, val')$ in ρ.

Condition (C4) specifies the lateness condition. The head must always be dequeued in k dequeue operations. This means that any other value or $\perp$ can be dequeued as long as there haven't been k dequeues since the removal of the head.

2 k-relaxed Algorithm for Lateness Queues

2.1 System Model

This is the same system model used in [13]. We have a set of processes $p_0, \ldots, p_{n-1}$ acting as individual state machines. We have three types of events in our system: operation invocations, message receipts, and timer expirations.

A *view* of a process is a sequence of steps in which, as defined in [13],

- the old state of the first step is an initial state of the state machine;
- the old state of each step after the first one equals the new state of the previous step;
- each timer in the old state of each step has a value that does not exceed the clock time of the step;

- if the trigger of a step is a timer going off, then the old state of the step has a timer whose value is equal to the clock time for the step
- clock times of steps are increasing, and if the sequence is infinite then they increase without bound;
- at most one operation instance is pending at a time

A *timed view* is a view with a "real time" number associated with every step. A *run* is a set of n timed views. A run is *admissible* if all messages are delivered between $d-u$ and d real time after sending and each process has a local clock that runs at the same rate as real time, and all local clocks are within ϵ of each other. A run is *complete* when every sent message has been received and if each timed view either ends at a state where there are no timers set or is infinite.

Every message sent is assumed to take the minimum time of $d-u$, where d is the message delay and u is the uncertainty.

2.2 Algorithm Summary

We provide an algorithm for a Lateness k-Relaxed Queue. This algorithm assumes $k > 3n$ where n is the number of processes in our system. This algorithm is inspired by the algorithm for Out-of-Order k-relaxed queues in [13]. To increase average performance, we allow any element of the queue to be dequeued at any time as long as the the earliest enqueued element that has yet to be dequeued (the *head* of the queue) is removed at least every k dequeue operations. This is maintained by each process being able to quickly return the elements that are assigned to it through the labeling process as long as it hasn't completed $l-1 = \lfloor \frac{k}{n} \rfloor - 1$ dequeues since the head has been removed. Every element is labeled for a particular process when it is enqueued. The process with the least number of elements labeled for it in the current state of the queue is assigned the enqueued element.

When a process p_i calls the dequeue operation, it checks to see if the number of local dequeues it has called since the head was last removed is $< l-1$. If this holds, then p_i's local count of dequeues is increased by 1 and the process calls a "fast dequeue" by sending $(deq\_f, x, \langle localTime, i\rangle)$ to all processes including itself. "deq_f" informs the processes that a fast dequeue is being called, x is the element labeled for p_i that is being dequeued, and $\langle localTime, i\rangle$ is the *timestamp* of the operation. The timestamp is used to determine the order in which operations execute locally. At the time of execution at each process, x will be removed from the process' local queue.

If the number of dequeues p_i has called since the head was last removed is $\geq l-1$, then the process calls a "slow dequeue" by sending $(deq\_s, lQueue.head(), \langle localTime, i\rangle)$ to all processes including itself. $lQueue.head()$ represents the head of the local queue, which is the element that p_i is attempting to remove. At the time of execution, if $lQueue.head()$ is still the head of the queue, then it will be removed and its value returned to p_i. The count of dequeues that have been called since the head was last removed is reset to 0, and the new head will be determined so that it can be marked as unavailable for fast dequeues to return. If $lQueue.head()$ is not still the head of the queue at the time of execution, then some other element labeled for p_i will be removed and its value returned to p_i.

3 Local Variables

We specify the local variables our algorithm will use at each process.

- *count*: Count of how many dequeues the process has executed since the head was last dequeued, initially 0.
- *lQueue*: Local copy of the simulated data structure, initially empty. Each entry has three associated fields: a *label* field which is initially *null* and can hold a process id, a *value* field that contains a unique integer value for the element, and a boolean *available*, initially true. Behavior is an extension of a (sequential) FIFO Queue. Operations:
  - $enq(val)$: inserts val
  - $head()$: returns, without removing, the earliest enqueued element that has yet to be dequeued in $lQueue$, $\perp$ if none exists
  - $deqByLabel(p_j)$: removes and returns arbitrary element labeled p_j, with $available == true$, $\perp$ if none exists
  - $peekByLabel(p_j)$: returns, without removing, arbitrary element labeled p_j, with $available == true$, $\perp$ if none exists
  - $deqHead()$: removes and returns the earliest enqueued element that has yet to be dequeued element from $lQueue$, $\perp$ if none exists
  - $size()$: returns current number of elements in $lQueue$
  - $sizeByLabel(p_j)$: Returns number of elements in $lQueue$ with label p_j
  - $remove(val)$: removes element from $lQueue$
  - $label(p_j, val)$: label val with p_j
- *Pending*: Priority queue to hold operation instances, keyed by timestamp; initially empty. Supports standard operations $insert(op, val, ts)$, $min()$, and $extractMin()$.

4 Lateness Queue Pseudocode

Algorithm 1 Code for each process p_i to implement a Queue with k-lateness for $k \geq n$, where $l = \lfloor k/n \rfloor$.

```
1: HandleEvent ENQUEUE(val)
2:     send (enq, val, ⟨localTime, i⟩) to all
3:     setTimer(ϵ, ⟨enq, val, ⟨localTime, i⟩⟩, respond)

4: HandleEvent R-DEQUEUE
5:     if count < l − 1 then                          ▷ Lateness requirement met, Fast dequeue
6:         count + +
7:         x = lQueue.peekByLabel(p_i)
8:         x.available == false
9:         send (deq_f, x, ⟨localTime, i⟩) to all
10:        setTimer(ϵ, ⟨deq_f, x, ⟨localTime, i⟩⟩, respond)
11:    else                                           ▷ Slow Dequeue
12:        send (deq_s, lQueue.head(), ⟨localTime, i⟩) to all

13: HandleEvent RECEIVE (op, val, ts) FROM p_j
14:    Pending.insert(⟨op, val, ts⟩)
15:    setTimer(u + ϵ, ⟨op, val, ts⟩, execute)

16: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, respond)
17:    if op == deq_f then return val.value           ▷ R-DEQUEUE returns
18:    else return ACK                                ▷ ENQUEUE returns

19: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, execute)
20:    while ts ≥ Pending.min() do    ▷ While Pending contains operations with earlier timestamps
21:        ⟨op′, val′, ts′⟩ = Pending.extractMin()
22:        executeLocally(op′, val′, ts′)
23:        cancelTimer(⟨op′, val′, ts′⟩, execute)

24: function EXECUTELOCALLY(op, val, ⟨∗, j⟩)
25:    if op == enq then
26:        lQueue.enq(val)
27:        let m = min_h{lQueue.sizeByLabel(p_h)}     ▷ Process with least number of elements labeled.
28:        lQueue.label(p_m, lQueue.tail)
29:        if lQueue.size() == 1 then
30:            lQueue.head().available = false
31:    else if op == deq_f then
32:        lQueue.remove(val)
33:    else                                           ▷ Slow dequeue
34:        if val.value == lQueue.head().value then   ▷ If val is still the head, dequeue head
35:            count = 0                              ▷ Reset local count
36:            ret = lQueue.deqHead().value
37:            lQueue.head().available = false
38:        else                                       ▷ Don't dequeue head
39:            ret = lQueue.deqByLabel(p_j).value
40:            if j == i then count + +               ▷ Increase local count
41:        if j == i then return ret
```

5 Proof

Lemma 1. *All invoked operation instances return in at least ϵ time.*

Proof. There are three possible outcomes of the ENQUEUE and R-DEQUEUE events: *enq*, *deq_f*, and *deq_s*.

- When *enq* is invoked, a response timer is set for ϵ time (Line 3). When the response timer expires, **ACK** is returned (Line 18).
- When *deq_f* is invoked, a response timer is set for ϵ time (Line 10). When the response timer expires, the value being dequeued is returned (Line 17).
- When *deq_s* is invoked, say by process p_j, $(deq\_s, lQueue.head(), \langle localTime, j\rangle)$ is sent to all processes including p_j. When p_j receives the message, the operation is added to the local $Pending_j$ (Line 14) and the timer for execution is set for $u+\epsilon$ time (Line 15). Either the timer for this operation expires in $u+\epsilon$ time or another timer for an operation with a larger timestamp expires, and $executeLocally(\ldots)$ is called (Line 22). The value *ret* is returned (Line 41) to the invoking process p_j.

  If another timer for an operation op' with a larger timestamp expires, then the slow dequeue might return before $u+\epsilon$ time passes. op' has a larger timestamp which means that it is invoked at most ϵ earlier the slow dequeue in real time, so the soonest op''s timer can expire is ϵ before the slow dequeue's own timer at its invoking process would go off, giving d as the minimum time for a slow dequeue. It is assumed that $d > u$ and $u > \epsilon$, so $d > \epsilon$.

□

Construction 1. *Define the permutation π of operation instances in a complete, admissible run of Algorithm 1 as the order given by sorting by the timestamp of op, $ts(op)$, for each instance op.*

We want to show that this construction respects timestamp order.

Lemma 2. *Each process locally executes all invoked enqueues and dequeues in timestamp order, i.e., in the same order the operations occur in π.*

Proof. Since the invoker of each operation sends a message with the information about the operation to all the processes (including itself), and since no messages are lost, every process p_i puts every operation in $Pending_i$.

We now argue that every operation op in $Pending_i$ is eventually locally executed by p_i. When p_i gets the message about op, it inserts op into $Pending_i$ and sets a time for $u+\epsilon$ time in the future. If the timer goes off, then op is locally executed at p_i. The timer is cancelled only if op is locally executed as a result of an earlier timer going off (cf. Lines 22-23).

Now we show that the operations are locally executed at p_i in timestamp order. Suppose in contradiction there is an operation op_1 that is locally executed after operation op_2 but $ts(op_1) < ts(op_2)$. Furthermore, assume that op_2 has the largest timestamp of all operations that are locally executed at p_i before op_1. Since operations are removed from $Pending_i$ in timestamp order, it must be that op_1 is not in $Pending_i$ when op_2 is extracted for local execution.

Let t_1 be the real time when op_1 is invoked (at some process q_1) and t_2 be the real time when op_2 is invoked (at some process q_2). Since the clocks are synchronized to within ϵ and op_1's timestamp is less than op_2's timestamp, it follows that $t_1 < t_2 + \epsilon$.

The earliest that p_i can receive the message about op_2 and put it in $Pending_i$ is $t_2 + d - u$, while the latest that p_i can receive the message about op_1 and put it in $Pending_i$ is $t_1 + d$.

- Case 1: Suppose p_i locally executes op_2 because op_2's own timer goes off. The earliest this can happen is $t_2 + d - u + u + \epsilon = t_2 + d + \epsilon$. From above, we have $t_2 + d + \epsilon > t_1 + d$, i.e., this happens *after* p_i has put op_1 in $Pending_i$, contradiction.
- Case 2: Suppose p_i locally executes op_2 because the timer of another operation, say op_3, expires. Then it must be that $ts(op_3) > ts(op_2)$, and thus op_3 also has the property of being locally executed before op_1. This contradicts our choice of op_2 as the operation with the largest timestamp that is locally executed before op_1.

□

Notation 1. *For each prefix π' of π and each i, define $t(\pi', i)$ as the real time when process p_i has just finished locally executing the last operation in π'.*

Lemma 3. *For all processes p_i and p_j and every prefix π' of π, $lQueue_i$ at time $t(\pi', i)$ is the same as $lQueue_j$ at time $t(\pi', j)$, ignoring the values of the available fields of the elements.*

Proof. By Lemma 2, p_i and p_j locally execute operations in the same order. The only aspect of the *lQueue* local variables that is changed outside of the *executeLocally* function is the available flag (Line 8). Requiring that the min function accessed in Line 27 is deterministic gives the result. □

Lemma 4. *The sequence π produced by Construction 1 respects the real-time order of non-overlapping operations in the execution.*

Proof. Suppose in contradiction there is an operation op_1 that returns before operation op_2 is invoked, but op_1 follows op_2 in π. By Construction 1, it follows that $ts(op_1) > ts(op_2)$. Since each operation takes at least ϵ time to return, it can be concluded that op_1 starts at least ϵ time after op_2 starts. But since the clock skew is at most ϵ, it cannot be that $ts(op_1) < ts(op_2)$. □

5.1 Legality

The following lemmas assist in proving the legality of the queue in every execution of the algorithm.

Lemma 5. *At $t(\pi' i)$, if $lQueue_i.size() \geq 1$, $lQueue_i.head.available == false$ for every prefix π' of π and every $i \in [0, n-1]$.*

Proof. This proof utilizes induction on the sequence of operation instances in permutation π created by Construction 1.

- *Base Case:* Initially, *lQueue* is empty across all processes ($lQueue.size() == 0$). Therefore, the lemma is vacuously true.
- *Inductive Step:* Assume for all prefixes of π', $lQueue_i.head.available == false$ and $lQueue_i.head$ is the same for all processes p_i, $i \in [0, n-1]$. Both events "R-DEQUEUE" and "ENQUEUE" must be considered.
  - ENQUEUE(val): Due to Lemma 2, it is known that the first enqueue invoked by a process will be the first enqueue to be locally executed by each process. First, $(enq, val, \langle localTime, i \rangle)$ is sent to all and the timer is set (Line 3). Then the operation is added to $Pending_j$ for all processes p_j (Line 14) and the timer for execution is set (Line 15). Either the timer expires for this operation or another timer for an operation with a larger time stamp expires, and $executeLocally(\ldots)$ is called for the

enqueue (Line 22). Because $op == enq$, val is enqueued into $lQueue_j$ (Line 26). If $lQueue_j$ was empty immediately before the enqueue, $lQueue_j.size() == 1$, so $lQueue_j.head().available$ is declared $false$ (Line 30). Every process will have locally executes the enqueue by $t(\pi', j)$, where the enqueue operation is the last operation in pi', and every $lQueue_j$ will have the same elements by Lemma 2, so the head will be consistent across all processes. If $lQueue_j$ wasn't empty, then $lQueue_j.size()$ increases by 1, so $lQueue_j.size() > 1$ (Line 29). Therefore, the availability of the head hasn't changed.

- R-DEQUEUE(): If $lQueue.peekByLabel(p_i) \neq \perp$ and $count_i < l$ then a fast dequeue is invoked. First, $(deq\_f, x, \langle localTime, i\rangle)$, where x is an element being dequeued determined by the $peekByLabel(p_i)$ function, is sent to all and the timer is set (Line 10). Then the operation is added to $Pending_j$ for all processes p_j (Line 14) and the timer for execution is set (Line 15). Either the timer expires for this operation or another timer for an operation with a larger time stamp expires, and $executeLocally(\ldots)$ is called to execute the fast dequeue (Line 22). Because $op == deq\_f$, some item that is available is dequeued (Line 32), which, by the inductive hypothesis, means the head isn't dequeued and therefore hasn't changed and will remain unavailable.

  Else, a slow dequeue is invoked. First, $(deq\_s, lQueue.head(), \langle localTime, i\rangle)$, is sent to all. Then the operation is added to $Pending_j$ for each process p_j (Line 14) and the timer for execution is set (Line 15). Either the timer expires for this operation or another timer for an operation with a larger time stamp expires, and $executeLocally(\ldots)$ is called to execute the slow dequeue (Line 22). Because $op == deq\_s$, then the process checks if the head is still present. If the head is present (Line 34), then it will be dequeued (Line 36). In this case, the head now must change. The new head is determined, which by Lemma 2 would be the same element for every process, and set to be unavailable (Line 37).

  Else if the head is not present (Line 38), then a $deqByLabel(\ldots)$ will be called (Line 39), which can not remove the head and therefore the head will stay the same and remain unavailable.

□

Lemma 6. *Two slow dequeues will not both return the same value.*

Proof. For two slow dequeue operations, deq_1 and deq_2, let $ts(deq_1) < ts(deq_2)$. It has been proven that operations will execute in time stamp order, so deq_1 will locally execute before deq_2. If the head isn't available when deq_1 executes, then another resource labeled for the process invoking deq_1 is dequeued (Line 39). When deq_2 is executed, the process checks if the head is still present (Line 34). If it isn't, then another resource labeled for the process invoking deq_2 is dequeued (Line 39). By Lemma 2, labeling is consistent across all processes, so deq_1 and deq_2 will not return the same value given that they are invoked by different processes. If the head is still available when deq_1 locally executes, the head is removed from $lQueue$ (Line 36). By Lemma 5, the head won't be available and therefore can't be returned by deq_2. □

Lemma 7. *A slow dequeue deq from some process p_i not return the value x identified as the local head in Line 12 on behalf of another slow dequeue with an earlier timestamp removing x from $lQueue_i$ in between deq's invocation and local execution.*

Proof. A slow dequeue will result in the returning of ret (Line 41). If the value val identified by the process invoking the slow dequeue p_i as the head at the time of invocation equals the value

identified by p_i as the head at the time of execution, then *ret* is set to equal that value (Line 36). Therefore, the case where the head won't be returned is when *val* does not equal the global head.
There are two instances in which the head of $lQueue_i$, x_1 could change. Firstly, a new head, x_2 could be chosen without x_1 being removed. Secondly, x_1 could be removed from $lQueue_i$. However, if a new head is chosen, then that means that x_2 was enqueued before x_1, which, by Lemma 2 implies that $ts(x_2) < ts(x_1)$. If this were the case, then x_2 would already be the head, which gives us a contradiction.

Therefore, the head must be removed from $lQueue_i$. A fast dequeue can not remove the head by Lemma 5, so it must be removed by a slow dequeue.

This change must happen between local invocation and execution of the slow dequeue *deq* because if the head was locally dequeued before invocation, then the dequeue would have locally executed and the process would know that the head is new. If the head was dequeued after execution, then *deq* would have already executed.

□

Theorem 1. *For any execution of the algorithm, the permutation π given by Construction 1 is legal by the specification of a lateness k-relaxed queue.*

Proof. [1]

After every operation in π, the queue is legal if the following properties, which were previously defined, are met.

(C1) **Every argument to an instance of *Enqueue* is unique.** The property of uniqueness holds because it is assumed that every item being enqueued is unique.

(C2) **Every return value of an instance of *Dequeue* is unique.** Assuming every dequeued item has been previously enqueued and every enqueued item is unique, then every dequeued item will be unique as well. However it must also be shown that every item that is enqueued can only be dequeued once. Therefore, we must examine every combination of dequeues and make sure they can not possibly return the same value.

- **Two fast dequeues.** For a fast dequeue, every resource in the queue can only be removed by the process it is labeled for, and this labeling scheme is consistent across all processes by Lemma 2. This prevents one item being dequeued by two different processes. One process will not dequeue an item labeled to it more than once because the item is immediately removed from *lQueue* when the fast dequeue is locally executed (Line 32).
- **Two slow dequeues.** Lemma 6 proves that two slow dequeues can not both return the same value.
- **A fast dequeue and a slow dequeue.** Proof by contradiction. Assume that some fast dequeue deq_f invoked by process p_i and some slow dequeue deq_s invoked by process p_j both return the same value *val*. Fast dequeues only return elements labeled to their invoking process so *val* must have label i in $lQueue_i$. Labels are consistent across processes (Lemma 2), so by the time p_j invokes deq_s, *val* will also have label i in $lQueue_j$. It must be taken into consideration if deq_f and deq_s are being invoked by the same process, or in other words $i = j$.

[1] Some aspects of this proof are inspired by the proof of similar properties of unrestricted out-of-order k-relaxed queues in [13].

– $i = j$: If p_i and p_j are in fact the same process, then deq_f and deq_s can not both be pending at the same time. This would mean that whichever operation with an earlier timestamp would invoke and execute before the other process could invoke and execute. Every execution of a dequeue results in the removal of the dequeued element from the local queue, so the element dequeued by the earlier operation will not be in the local queue at the time of the second operation's invocation.
– $i \neq j$: If deq_f has a smaller timestamp than deq_s, then deq_f would execute before deq_s executes, and val would no longer be in $lQueue_j$ at the time deq_s executes. Therefore, deq_s must have a smaller timestamp than deq_f.
If p_i and p_j are two different processes, then deq_s would have to be returning an item labeled for p_i. The only time a slow dequeue returns a value not labeled for its invoking process is when it dequeues the head. Therefore, val must be the head of $lQueue_j$ at the time of invocation and execution of deq_s. However, a fast dequeue can not return the head, so val must not be the head of $lQueue_i$ at the time of invocation of deq_f, which is at most ϵ time after deq_f's invocation. All processes execute every operation in the same order and once an element is the head of a local queue, it can only cease to be the head after its removal from the queue, so we can conclude that p_j has executed some operation deq'_s that has removed the head of $lQueue_j$ by the time of deq_s's invocation and p_i has not yet executed deq'_s by the time of deq_f's invocation.
An operation in locally executed at all processes within $u + \epsilon$ time, and deq_f was invoked between the local execution of deq'_s at p_j and the local execution of deq'_s at p_i. Therefore, deq_f must have been invoked at most $u + \epsilon$ real time after deq'_s was locally executed at p_j.
Since deq_s is a slow dequeue that removes the head of $lQueue_j$, then we can conclude that $count_j$ has increased from 0 at the time of the execution of deq'_s to at least l at the time of the execution of deq_s. Every fast dequeue takes at least ϵ time to return, so at least $l * \epsilon$ time has passed between the p_j's local execution of deq'_s and invocation of deq_s. Therefore, the execution of deq'_s at p_i is at most $(u + \epsilon) - (l * \epsilon) = u + \epsilon(1 - l)$ real time after the invocation of deq_s. The invocation of deq_f has a larger timestamp than the invocation of deq_s and occurs before the local execution of deq'_s at p_i, which means that deq_f was invoked at most $u + \epsilon(1 - l)$ real time after deq_s. If we assume a reasonably large relaxation of $k \geq 3n$ then $l \geq 3$ and $u + \epsilon(1 - l) \leq u - 2\epsilon$. In reference [4], it is determined that $\epsilon \geq u(1 - \frac{1}{n})$, making $u - 2\epsilon \leq u(-2 + \frac{2}{n}) < -u < 0$. This tells us that deq_f was invoked $-u$ after deq_s which is a contradiction.

(C3) **Every non-$\perp$ value which an instance of *Dequeue* returns is the argument to a previous instance of *Enqueue*.** The only possible value that can be returned by a dequeue function is a value from each process p_i's local queue $lQueue_i$. The only way an item can be added to $lQueue$ is through an enqueue. Therefore, only enqueued items can be dequeued.

(C4) **If ρ is a legal sequence of operation instances, then $\rho \cdot Dequeue(-, val)$ is legal iff every $Enqueue(val', -)$ preceding $Enqueue(val, -)$ has a matching $Dequeue(-, val')$ in ρ or there are fewer than $k - 1$ instances $Dequeue(-, val')$ that follow the first $Enqueue(val'', -)$ which does not have a matching $Dequeue(-, val'')$ in ρ.**

Further, $\rho \cdot Dequeue(-, \perp)$ is legal iff there are fewer than $k-1$ instances $Dequeue(-, val')$ that follow the first $Enqueue(val'', -)$ without a matching $Dequeue(-, val'')$ in ρ

or every val' **such that** $Enqueue(val', -)$ **is in** ρ **has a matching** $Dequeue(-, val')$ **in** ρ**.**

This is the property that states that at the end of the local execution of each operation in π the amount of values or $\perp$s that have been dequeued by all processes since the head was dequeued, j, is less than k.

There are two cases to consider: slow dequeues that don't return the head and fast dequeues.

By Lemma 7, it is known that the only time a slow dequeue, deq_1 doesn't return the head is when another slow dequeue deq_2 executes between its invocation and local execution. If process p_i is invoking deq_1, no fast dequeues are going to be invoked or executed by p_i in between the invocation and execution of deq_1. So, when deq_2 executes, $count_i = 0$, and when deq_1 executes, a resource labeled to p_i is dequeued and $count_i$ is increased by 1 (Line 6), making $count_i = 1$, which maintains that $count_i < \lfloor \frac{k}{n} \rfloor$ for $k > n$.

For all processes p_i, a fast dequeue only invokes under the condition that $count_i < \lfloor \frac{k}{n} \rfloor$ (Line 5), where $count_i$ is the local count of dequeues that have occurred since the head was last removed. Therefore the most items that can be removed by fast dequeues is some integer $j < n * \lfloor \frac{k}{n} \rfloor \leq k$.

□

5.2 Average Runtime

The following definition is from [13].

Definition 1. *We will call a run* heavily loaded *if every Dequeue is linearized after a prefix of π in which there are at least k more instances of Enqueue than instances of Dequeue.*

Theorem 2. *The average time complexity per Dequeue in any heavily loaded, complete admissible run for Algorithm 1 is no more than $\frac{d}{\lfloor k/n \rfloor} + \epsilon$.*

Proof. Consider the view of a single process p_i. $count_i$ is initially zero, and p_i will invoke only fast dequeues until $count_i \geq l - 1$, so at least $l - 1$ fast dequeues will take place, each taking ϵ time, before a slow dequeue occurs. The worst case scenario is the one in which slow dequeues occur most frequently. At most, a slow dequeue will be invoked every $l - 1$ dequeues. A slow dequeue will $d - u + \epsilon + u = d + \epsilon$ time from the time of its invocation to execute at p_i. Then, $count_i$ will be set to 0, and l more fast dequeues can occur. Thus the following pattern emerges: every l operations, $l - 1$ fast dequeues occur and 1 slow dequeue occurs, resulting in the average runtime: $\frac{\epsilon(l-1)+d+\epsilon}{l} = \frac{d}{l} + \epsilon$.

□

6 Conclusion

We have provided an example of another relaxed data type that achieves higher performance in message-passing systems. First, we reviewed the definition of a k-relaxed lateness queue and gave an example algorithm for a message passing system to implement it. Then we proved the correctness of this algorithm.

Moving forward, we would like to see if this algorithm can be proven to work for a less stringent condition than $k > 3n$. Additionally, it would be ideal to modify the algorithm so that $\perp$ is returned less frequently.

References

[1] Afek, Y., Korland, G., Yanovsky, E.: Quasi-linearizability: Relaxed consistency for improved concurrency. In: Principles of Distributed Systems - 14th International Conference (OPODIS), 395-410 (2010)

[2] Aspnes, J., Herlihy, M., Shavit, N.: Counting networks. J. ACM 41(5), 1020-1048 (1994)

[3] Attiya H., Guerraoui, R., Hendler, D., Kuznetsov, P., Michael, M.M., Vechev, M.T.: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In: Ball, T., Sagiv, M. (eds.) Principles Of Programming Languages (POPL), 487-498. ACM (2011)

[4] Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. ACM Transactions on Computer Systems, 12(2), 91-122 (1994)

[5] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., ul Haq, M.F., ul Haq, M.I., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., Rigas, L.: Windows Azure Storage: a highly available cloud storage service with strong consistency. In: Proceedings of the 23rd ACM Symposium on Operating Systems Principles, 143-157 (2011)

[6] Henzinger, T.A., Kirsch, C.M., Payer, H., Sezgin, A., Sokolova, A.: Quantitative relaxation of concurrent data structures. In: Giacobazzi, R., Cousot, R. (eds) Principles Of Programming Languages (POPL), 317-328. ACM (2013)

[7] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12(3), 463-492 (1990)

[8] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers, 28(9), 690-691 (1979)

[9] Lamport, L.: On interprocess communication. Distributed Computing, 1(2), 77-101 (1986)

[10] Lipton, R.J., Sandberg, J.D.: PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Department of Computer Science, (1988)

[11] Vogels, W.: Eventually consistent. Communications of the ACM, 52(1), 40-44 (2009)

[12] Wang, J., Talmage, E., Lee, H., Welch, J.L.: Improved time bounds for linearizable implementations of abstract data types. In: International Parallel and Distributed Processing Symposium (IPDPS), 691-701 (2014)

[13] Talmage, E., Welch, J.L.: Improving Average Performance by Relaxing Distributed Data Structures. In: Distributed Computing Lecture Notes in Computer Science, 421-438 (2014).