Improved and Partially-Tight Lower Bounds for Message-Passing Implementations of Multiplicity Queues

Anh Tran
Bucknell University, Lewisburg, PA, USA

Edward Talmage
Bucknell University, Lewisburg, PA, USA

Abstract

A *multiplicity queue* is a concurrently-defined data type which relaxes the conditions of a linearizable FIFO queue to allow concurrent *Dequeue* instances to return the same value. It would seem that this should allow faster message-passing implementations, as processes should not need to wait as long to learn about concurrent operations at remote processes and previous work has shown that multiplicity queues are computationally less complex than the unrelaxed version. Intriguingly, other work has shown that there is, in fact, not much speedup possible versus an unrelaxed queue implementation. Seeking to understand this difference between intuition and real behavior, we improve the existing lower bound for uniform algorithms. We also give an upper bound for a special case to show that our bound is tight at that point. To achieve our lower bounds, we use extended shifting arguments, which are rarely used. We use these techniques in series of inductive indistinguishability proofs, extending our proofs beyond the usual limitations of traditional shifting arguments. This proof structure is an interesting contribution independently of the main result, as new lower bound proof techniques may have many uses in future work.

2012 ACM Subject Classification Software and its engineering → Abstract data types; Theory of computation → Distributed algorithms; Computing methodologies → Distributed algorithms

Keywords and phrases Distributed Data Structures, ADTs, Lower Bounds, Shifting Arguments, Multiplicity Queues

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.34

Related Version *Short Version*: https://doi.org/10.1145/3583668.3594602 [17]
Preliminary Version: https://arxiv.org/abs/2305.11286 [18]

Funding *Anh Tran*: Bucknell University Emerging Scholars Program.
Edward Talmage: Bucknell University.

1 Introduction

In the search for efficient structured access to shared data, *relaxed data types* [5] have risen as an efficient way to trade off some of the precise guarantees of an ordered data type for more performance [14]. Multiplicity queues are a recently-developed relaxation of queues [4] which allow concurrent *Dequeue* instances to return the same value. Since they cannot have a sequential specification (being defined in terms of concurrency), previous results on relaxed queues do not apply to multiplicity queues.

Multiplicity queues are particularly interesting due to the implications for the computational power of the type. In [4], Castañeda et al. implement multiplicity queues from *Read*/*Write* registers, which is impossible for FIFO queues. This means that it is possible to have queue-like semantics without the cost of strong primitive operations like *Read*-

Modify-Write. Further work showed that this allows interesting applications in areas such as work-stealing [3] and more efficient implementations in shared memory systems with strong primitives than the best known algorithms for FIFO queues [7].

We are interested in message-passing implementations of data types, which provide the simplicity and well-defined semantics of a shared memory system in the message passing models inherent to geographically distributed systems [2]. Specifically, we want to implement shared structures efficiently in terms of the time between when a user invokes an operation and when the algorithm generates the operation's response. In queue implementations, the need for concurrent *Dequeue* instances to wait long enough to learn about each other, so that they can be sure to return different values, is one of the primary reasons that *Dequeue* is expensive to implement [19]. Between the higher performance multiplicity queues achieve in shared memory models and the intuitive notion that concurrent *Dequeue* instances need not learn about each other, it seems intuitive that multiplicity queues should have efficient implementations in message-passing systems.

However, recent work [13] showed the possible performance gains are limited. In a partially-synchronous system with maximum message delay d and delay uncertainty u, that work showed that the worst-case invocation-return delay for *Dequeue* is at least $\min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$, which is within roughly a factor of 2 of a known unrelaxed queue implementation, where $|Dequeue| = d + \epsilon$ [19]. We here extend the work in [13], increasing the lower bound for the return time of *Dequeue* in uniform algorithms (those whose behavior does not depend on the number of participating processes) to $\min\left\{\frac{3d+2u}{5}, \frac{d}{2}+u\right\}$. Intuitively, while a particular *Dequeue* instance may not need to know about other, concurrent instances, determining which instances are concurrent is expensive in its own right.

This suggests new insights into fundamental properties of message passing implementations of shared data structures, showing that differentiating previous from concurrent instances is responsible for much of the time operation instances require. This could help develop more efficient algorithms or new relaxations with minimal weakening while providing maximum performance improvements. The piecewise nature of our bound also provides potential insight into what the optimal lower bound may be. We show that the $\frac{d}{2}+u$ portion of the bound, which has lower slope, is actually tight in the case when all messages have equal delay ($u = 0$). However, for larger uncertainties in message delay ($u > d/6$), the $\frac{3d+2u}{5}$ portion of the bound is higher. In fact, if we could strengthen the base case of our induction, it appears our argument would give larger bounds than $\frac{3d+2u}{5}$ for large values of u. That base case is already the most complex portion of our proof, so such strengthening and finding an optimal lower bound remain as future work. The two parts of the bound we show here relate to different constraints, with $\frac{d}{2}+u$ relating to admissibility of runs in our proof and $\frac{3d+2u}{5}$ relating to how fast information can travel through the system and when processes can make conclusions about the ordering of *Dequeue* instances at other processes.

Except for the edge case of $u = 0$, where they match and which we show is tight, our new bound is larger than the previous state of the art, and further demonstrates better tools for larger lower bound proofs in general. The proofs in [13] relied on shifting and other indistinguishability arguments among three or fewer processes, and the bounds were limited by the number of processes. In this paper, we develop more complex indistinguishability arguments, using inductive definitions of different runs of the algorithm on many processes. This requires using advanced shifting and indistinguishability tools, similar to those developed in [19]. These stronger tools allow us to prove larger bounds, and are interesting in their own right as a hint to how to prove larger lower bounds on other problems, as well.

1.1 Related Work

The idea of relaxing data types grew out of the study of consistency conditions weaker than linearizability. Afek et al. proposed Quasi-Linearizability in [1], which requires linearizations to be within a certain distance of a legal sequence, instead of themselves legal. From another perspective, this is just an expansion of the set of legal sequences on the data type. In [5], Henzinger et al. formalized these relaxations of abstract data type specifications by increasing the set of legal sequences and defined several parameterized ways to do so.

These relaxations and other work which followed [14, 12, 15] concentrated on relaxing sequential data type specifications and showed that they have more efficient implementations in a message passing system than an unrelaxed queue does. This approach cannot consider concurrency, which is simply not defined in the sequential space, so Castañeda et al. [4] defined multiplicity queues, which allow different behavior in the presence of concurrent operations than during sequential operation. They also replaced linearizability with set-linearizability as the consistency condition to accommodate non-sequential data types definitions.

In shared memory models, multiplicity queues have a number of advantages over unrelaxed queues, and even other, sequential relaxations. Castañeda et al. showed that multiplicity queues can be implemented purely from *Read*/*Write* registers, which is impossible for FIFO queues [6] and most previously-considered relaxed queues [12, 16], as they have consensus number 2. This suggests that they may be a practical way to get queue-like behavior cheaply in shared memory systems. Castañeda and Piña [3] use multiplicity queues to provide the first work-stealing algorithms without strong synchronization requirements. Johnen et al. [7] considered the time complexity of shared memory implementations of queues, implementing multiplicity queues in $O(\log n)$ steps for each of *Enqueue* and *Dequeue*, while the best previous algorithm for unrelaxed queues took $O(\sqrt{n})$ steps [8].

2 Model and Definitions

2.1 System Model

To have parameters we can use to prove lower bounds, we work in a partially synchronous model of computation. Lower bounds in this model also apply in asynchronous models, so a high lower bound here is still meaningful. We work in the same system model as [13] and its precedents, a partially-synchronous, message passing model without process failures used in the literature for various shared data type implementation algorithms and lower bounds.[1] There are n processes, $\{p_0, \ldots, p_{n-1}\}$, participating in an algorithm implementing a shared memory object. Each process allows a user to invoke operations on the simulated shared memory object and generates responses to those invocations. Each user can invoke operations at any time when their process does not have a *pending* operation–an invocation which does not yet have a matching response. Processes have local clocks running at the same rate as real time, but each potentially offset from real time, and can use these clocks to set timers.

Processes are state machines, where operation invocations, message arrivals and timer expirations trigger steps, which may perform local computation, set timers, send messages, and generate operation responses. A *run* of an algorithm is a set of sequences of state machine steps, one sequence for each process. Each sequence in a run is a valid state machine history

[1] There are several distinct models which the literature calls partially synchronous. We consider a model that is never totally synchronous or totally asynchronous, but always has some uncertainty in message delay, and thus cannot perfectly synchronize processes [10].

with a real time for each step, and is either infinite or ends in a state with no unexpired timers and no messages sent to that process but not received. A run is *admissible* if every message send step has a uniquely corresponding message receive step with the *delay* between send and receive at least $d - u$ and at most d real time and the skew, or maximum difference between local clocks, is at most $\varepsilon := (1 - 1/n)u$ [10]. We assume d and $u \leq d$ are known system parameters.

An implementation must provide *liveness*: every operation invocation must have a matching response. We call this invocation and corresponding response pair an *operation instance*. We are exploring the time cost of implementations, as measured by the worst-case delay between an instance's invocation and response. For operation OP, $|OP|$ denotes the maximum, over all admissible runs of the algorithm, of the real time between the invocation and response of any instance of OP. We measure communication cost, so we assume local computation is instantaneous. We also restrict ourselves to *eventually quiescent* implementations, requiring that if there are a finite number of operation invocations in a run, there is a finite time after the last invocation by which processes reach and stay in a *quiescent* state with no messages in transit and no timers set. A *uniform* algorithm is independent of the number of processes, with each process' state machine is identical for all n.[2]

A sequential data type specification gives a set of operations the user may invoke, with argument and return types, and the set of legal sequences of instances of those operations. We are interested in data types whose behavior may depend on concurrency in a distributed system, so we consider *set-sequential data type specifications*. A set-sequential data type specification similarly defines the set of operations the user may invoke, but instead of a set of legal sequences of instances, specifies a set of legal sequences of *sets of instances*. Thus, not all instances in a run must be totally ordered relative to each other, but each set of instances must be totally ordered relative to other sets.

To determine whether an algorithm implementing a set-sequential data type is correct, we require it to be *set-linearizable*, as defined in [11] and [4]. Set-linearizability requires that for every admissible run of the algorithm, there is a total order of sets of operation instances which contains every instance in the run, is legal by the set-sequential data type specification, and respects the real-time order of non-overlapping instances. That is, there must be a way to place all operation instances in the run in sets and order those sets into a legal sequence such that for every pair of instances where op_1 returns before op_2's invocation, op_1 is in a set which precedes the set containing op_2. The classic notion of linearizability is a special case of set-linearizability where all sets are required to have cardinality 1.

2.2 Multiplicity Queues

A *queue* is a First-In, First-Out data type providing operations $Enqueue(arg)$ which returns nothing and $Dequeue()$ which returns a data value, where any sequence of operations instances is legal iff each *Dequeue* instance returns the argument of the earliest preceding *Enqueue* instance whose argument has not already been returned by a *Dequeue*, or $\perp$ (which cannot be an *Enqueue* instance's argument) if there is no such *Enqueue* instance. We consider a related data type called a *multiplicity queue*, defined in [4], with the same operations but defined set-sequentially.

[2] It may seem that broadcasting requires knowledge of n, but since each process can simply iterate across all neighbors, no logic changes for different n.

▶ **Definition 1.** *A* multiplicity queue *over value set V is a data type with two operations: $Enqueue(arg)$ takes one parameter $arg \in V$ and returns nothing. $Dequeue()$ takes no parameter and returns one value in $V \cup \{\perp\}$, where $\perp$ is special "empty" value. A sequence of sets of Enqueue and Dequeue instances is legal if (i) every Enqueue instance is in a singleton set, (ii) all Dequeue instances in a set return the same value, and (iii) each Dequeue instance deq returns the argument of the earliest Enqueue instance preceding deq in the sequence, which has not been returned by another Dequeue instance preceding deq. If there is no such Enqueue instance, deq returns $\perp$.*

This definition implies that concurrent *Dequeue* instances may, but do not necessarily, return the same value. If they do, they would set-linearize in the same set. If two *Dequeue* instances are not concurrent, then one must precede the other in the set linearization, so they must return different values. We assume all *Enqueue* arguments are unique, which is easily achieved by a higher abstraction layer timestamping the users' arguments.

2.3 Shifting Proofs

To prove our lower bounds, we will use indistinguishability arguments, where we argue that in a given time range in two runs, one or more processes have the same inputs (invocations, messages, timers) at the same local clock times. Since each process is a (deterministic) state machine, a process receiving the same inputs at the same times performs the same steps in the two runs. We will sometimes argue indistinguishability directly, but in some cases we will use *shifting* [10, 9, 19], a technique which mechanically changes the real time of events at one or more processes, while adjusting message delays and clock offsets such that each event happens at the same local time. Thus, if one run is a shift of another, they are indistinguishable. More formally, given run R and vector $\vec{v}$ of length n, we define $Shift(R, \vec{v})$ as a new run in which each event e at each $p_i, 0 \leq i < n$ which occurs at real time t in R occurs at real time $t + v[i]$. In $Shift(R, \vec{v})$ each local clock offset c_i becomes $c'_i = c_i - v[i]$. Finally, any message from p_i to p_j which had delay x in R has delay $x + v[j] - v[i]$, as the real times when it is sent and received change.

A challenge in using shifting arguments is that the shifted run must be admissible to require the algorithm to behave correctly. Great care is required to define a run's message delays and clock offsets so that the skew and message delays are admissible after shifting. We use an extended shifting technique by Wang et al. [19] which shows that if a shift is too large, making the shifted run inadmissible, it is in some cases possible to chop off each process' sequence of steps before a message arrives after an inadmissible delay, then extend the run from that collection of chop points with different message delays which are admissible. This extended run is not necessarily indistinguishable past the chop, but we can sometimes argue that the runs are indistinguishable long enough to form conclusions about the new run's behavior.

3 Lower Bound Proof Outline

Our primary result is a lower bound on the worst-case time of any uniform set-linearizable implementation of a multiplicity queue. For comparison, a linearizable implementation of an unrelaxed FIFO queue is possible with worst-case *Dequeue* cost $d + \varepsilon = d + (1 - 1/n)u$. Our lower bound is over half of that, indicating a limit on the performance gains of the multiplicity relaxation. Since our lower bound shows the impossibility of a more efficient multiplicity queue implementation in a relatively well-behaved partially synchronous model of computation, it follows that it is similarly impossible in more realistic, and less well-behaved, models, such as those with asynchrony or failures. It is possible that the cost to port a FIFO queue to a less well-behaved model is higher than the cost to port a multiplicity queue, but that remains an open question.

We prove our bound by building up two sets of runs. In both sets, each process invokes a single *Dequeue* instance. In the first set we show that each of these *Dequeue* instances, despite being concurrent with at least one other *Dequeue* instance, returns a unique value. In the second set, we show that there are fewer distinct return values than *Dequeue* instances, so there must be some *Dequeue* instances returning the same value. We then show that, for sufficiently large n, these two sets of runs eventually converge, in the sense that processes cannot distinguish which set they are in until after they choose return values for their *Dequeue* instances. This means they must have the same behavior in both, a contradiction which implies the worst-case cost of *Dequeue* must be higher. We need large n to ensure that the information about all of the *Dequeue* instances cannot reach the last process in time for it to distinguish which run it is in, so the lower bound on n depends on the relationship between d and u, increasing as u approaches d.

Both sets of runs are based on and building towards one simple run, which sequentially enqueues values $1..n$, then once the system is quiescent, has each process dequeue once, with invocation times staggered so that the *Dequeue* instances at different processes overlap slightly (unless $|Dequeue| < u$, in which case processes invoke *Dequeue* simultaneously, which the variable s below handles). Any set linearization of any of our runs will start with n singleton sets, enqueueing the values $1..n$ in order. Further operation instances will set-linearize after those *Enqueue* instances. In general, messages from lower-indexed processes to higher-indexed processes take $d - u$ time, while those from higher-indexed processes to lower-indexed processes take d time. The primary exception is that after a certain point, messages from p_{n-1} to p_n will also take d time. This prevents p_n from collecting complete information on other processes' actions, which is enough uncertainty to cause incorrect behavior. As we develop our proof, we will modify other delays, but start from this pattern.

Let $Q := \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\}$ throughout the paper. To prove that $|Dequeue| \geq Q$, in the following we assume for the sake of contradiction that $|Dequeue| < Q \leq d$.

4 Distinct Return Values

For our first set of runs, we show that each *Dequeue* instance may return a distinct value, despite the fact that each is concurrent with at least one other instance. While this is the easier part of the proof, it is interesting as it shows that, under uncertainty in message delay, processes cannot tell whether their *Dequeue* instances are or are not concurrent, so the relaxation gives no advantage, as processes must spend time to choose distinct return values.

We denote this set of runs by $D_k, 1 \leq k \leq n$, where the first k processes invoke *Dequeue* instances slightly overlapped as discussed, and higher-indexed processes invoke *Dequeue* slightly later. We will inductively show that the *Dequeue* at p_k must return a different value from those at $p_0, \ldots, p_{k-1}$, then shift the run to obtain D_{k+1}, which is indistinguishable. When the inductive chain of shifts is complete, we will see that all n *Dequeue* instances in D_n must return different values.

▶ **Construction 2.** *Define run D_k (D for Distinct) as follows, for each $1 \leq k < n$:*
- *p_0 invokes $Enqueue(1) \cdots Enqueue(n)$ in order. Let t be any arbitrary time after $Enqueue(n)$ returns at which the system is quiescent.*
- *$\forall 0 \leq i < k$, process p_i invokes Dequeue at time $t + i \times s$, where $s = \max\{0, Q - u\}$.*
- *$\forall k \leq j < n$, process p_j invokes Dequeue at time $t + (j-1)s + (s+u)$.*

- *Process p_0 has local clock offset $c_0 = 0$.*
- *$\forall 0 < i < k$, process p_i has local clock offset $c_i = \left(\frac{i}{n}\right) u$.*
- *$\forall k \leq j < n$, process p_j has local clock offset $c_j = \left(\frac{j-n}{n}\right) u$.*
- *$\forall 0 \leq i < k \leq j < n$, messages from p_i to p_j have delay d, from p_j to p_i have delay $d - u$.*
- *$\forall 0 \leq a < b < k$, messages from p_a to p_b have delay $d - u$, from p_b to p_a have delay d.*
- *$\forall k \leq c < d < n$, messages from p_c to p_d have delay $d - u$, from p_d to p_c have delay d.*

Define run D_n similarly, but setting clock offsets, invocations, and message delays only for processes p_i with $i < n$. Since p_n does not exist, it does not invoke Dequeue, send or receive messages, or have a local clock offset.

Since all local clock offsets for processes p_i with $0 < i < k$ are positive and increase with i and all offsets for processes p_j with $k \leq j < n$ are negative and increase with j, the maximum skew between processes is $|c_{k-1} - c_k| = \left|\frac{k-1}{n} - \frac{k-n}{n}u\right| = \frac{n-1}{n}u = \varepsilon$, except when $k = n$, when no such p_j exists and the maximum skew is $\left|0 - \frac{n-1}{n}u\right| = \varepsilon$. With this fact and since all message delays are in the range $[d - u, d]$, we see that each D_k is an admissible run.

Our first step is to note that each D_k is a shifted version of D_{k-1}, which implies that they are all indistinguishable. The proof is a straightforward application of classic shifting, adjusting real times, clock offsets, and message delays, and is included in the appendix. Then we can prove that each *Dequeue* instance in D_n must return a different value.

▶ **Lemma 3.** *For all $2 \leq k < n$, $D_k = Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, where $\overrightarrow{s_{k-1}}$'s only non-zero component is $-u$ at index $k-1$: $\overrightarrow{s_{k-1}} = \langle 0, \ldots, 0, -u, 0, \ldots, 0\rangle$.*

▶ **Lemma 4.** *In run D_k, $1 \leq k \leq n$, every Dequeue instance returns a distinct value. Specifically, for each $0 \leq i < n$, the Dequeue instance at p_i returns $i + 1$.*

Proof. Consider D_1. Here, p_0 invokes *Dequeue* at time t, which must return by time $t + |Dequeue|$. p_1 invokes *Dequeue* at time $t + (1-1)s + (s+u) > t + |Dequeue|$, which is after p_0's *Dequeue* instance returns. Every process p_i with $i \geq 1$ invokes *Dequeue* no earlier than p_1, so no other *Dequeue* instance is concurrent with p_0's, and thus that one must set-linearize before any other. This means that p_0 returns 1 to its *Dequeue* instance and all other processes return values in the set $\{2, \ldots, n\}$ to their *Dequeue* instances.

Assume that for some arbitrary $0 \leq k < n - 1$, each process p_i, $0 \leq i < k$ returns $i + 1$ to its *Dequeue* instance. We will then show that process p_k returns $k + 1$ to its *Dequeue* instance. First, note that in D_k, p_k invokes *Dequeue* at time $t + (k-1)s + (s+u)$, while every $p_i, 0 \leq i < k$ has its *Dequeue* instance return no later than $t + (i-1)s + |Dequeue| \leq t + ((k-1)-1)s + |Dequeue| < t + (k-2)s + (s+u)$. Since $s \geq 0$, this is before p_k invokes *Dequeue*, so p_k's *Dequeue* instance must set-linearize strictly after all of those at any lower-indexed p_i. By the inductive hypothesis, each of those k processes returns $i+1$, so p_k must return a value larger than k.

Now, consider D_{k+1}. Since D_{k+1} is a shifted version of D_k, no process can distinguish the two runs, so all behave the same way in both. Specifically, p_k will return the same value to its *Dequeue* instance. But in D_{k+1}, by an identical argument to that in the previous paragraph, each $p_j, k < j < n$ invokes *Dequeue* after the *Dequeue* instance at p_k returns, so they must all set-linearize strictly after p_k's *Dequeue* instance, and those of each $p_i, 0 \leq i \leq k$. Since there are only $k+1$ *Dequeue* instances set-linearized with or before that at p_k, these must return values from the set $\{1, \ldots, k+1\}$. But we know that those at processes with indices in $\{0, \ldots, k-1\}$ all return values from $\{1, \ldots, k\}$, and the *Dequeue* instance at p_k returns a value distinct from any of these, so it must return $k + 1$, and we have the claim. ◀

5 Repeated Return Values

For the second set of runs, we will show that n processes, each invoking one *Dequeue* instance in our same partially-overlapping pattern, will not return all different values to those *Dequeue* instances. To do this, we first show that if only three processes invoke *Dequeue*, then they will only return two distinct values. We then inductively construct more and more complex runs, with one more process joining the pattern and invoking *Dequeue* in each successive pair of runs. When the induction reaches n, we will show that we have a run indistinguishable from the D_n we constructed in the previous section. Since each of the *Dequeue* instances in that run returns a distinct value, and those in the run we construct here do not all return distinct values, we have a contradiction, proving that the assumed algorithm cannot exist.

First, we define the family of runs S_k, in each of which only $k \leq n$ processes invoke *Dequeue*. We inductively show that each of these has some pair of *Dequeue* instances which return the same value, eventually showing that not all *Dequeue* instances in S_n return distinct values. Then, to show the chain of indistinguishabilities in our induction, we will need another set of runs, which are intermediate steps.

▶ **Construction 5.** *Define run S_k (S for <u>S</u>ame) as follows:*

- *p_0 invokes Enqueue$(1) \cdots$ Enqueue(n) in order. Let t be the same arbitrary time after Enqueue(n) returns at which the system is quiescent as in the definition of D_k.*
- *$\forall 0 \leq i < k$, process p_i invokes Dequeue at time $t + i \times s$, where $s = \max\{0, Q - u\}$.*
- *Process p_0 has local clock offset $c_0 = 0$, and $\forall 0 < i < n$, process p_i has $c_i = \left(\frac{i}{n}\right) u$.*
- *$\forall 0 \leq i < j < n$, messages from p_j to p_i have delay d and from p_i to p_j have delay $d - u$, except for those from p_{k-2} to p_{k-1} sent after $t^*_{k-2} = t + (k-2)(d-u)$, which have delay d.*

▶ **Construction 6.** *For $1 \leq k < n$, define run S'_k from run S_{k-1} by additionally having p_{k-1} invoke Dequeue at time $t + (k-1)s$. Adjust the delay of all messages from p_{k-2} to p_{k-1} sent at or after $t^*_{k-2} = t + (k-2)(d-u)$ to d.*

In S'_k, we have added the next *Dequeue* instance, but have two processes' messages (p_{k-3}'s and p_{k-2}'s) to the next, larger-indexed, process delayed. We can show that processes p_0 through p_{k-2} cannot distinguish S_{k-1} from S'_k before generating return values for their *Dequeue* instances, so they must return the same values, which gives us information about what p_{k-1} must return to its *Dequeue* instance. We then show that S'_k and S_k are indistinguishable to p_{k-1} until after it has generated a return value for its *Dequeue* instance, telling us what values it could return. The prof is by mathematical induction on k, from 3 to n.

▶ **Lemma 7.** *For sufficiently large n, all Dequeue instances in S'_n and S_n return values from $\{1, \ldots, n-1\}$.*

Proof. We proceed by induction on k.

Base Case. We proceed by induction on k, with base case $k = 3$, when only the first three of our n processes invoking *Dequeue*, which is run S_3. We show that all *Dequeue* instances return values from $\{1, 2\}$, for sufficiently large n. Due to higher-indexed processes invoking *Dequeue* later than lower-indexed processes, and the way we will set message delays, the first *Dequeue* instance will behave as if it were running alone, returning 1. We will then shift run S_3, using a technique like that in [19] that allows us to over-shift and break some message delays, then re-insert those messages with new, admissible delays. We can then show that the resulting patched run is still indistinguishable from the starting run for long enough. In this run, we will argue that the second process does not learn about the first process' *Dequeue* instance until after its own returns, and thus cannot distinguish this run from one in which it is running alone, so it must also return 1. Given these two return values, set-linearizability implies that the third process' *Dequeue* instance must return 2. We will then show that the third process cannot distinguish between the original and shifted runs before choosing its return value, so will return 2 in S_3.

First, observe that p_0 cannot learn about the *Dequeue* instances at p_1 and p_2 until after its own *Dequeue* instance has returned. Since all messages from a higher-indexed process to a lower-indexed process have delay d, the earliest p_0 will learn about the other *Dequeue* instances is at time $t+s+d$, since $t+s$ is when p_1 invokes *Dequeue*, and any message indicating that this has happened will take d time to reach p_0. Since p_2 invokes its *Dequeue* instance at time $t+2s \geq t+s$, the same logic will imply that p_0 will also not learn about that instance until after its own *Dequeue* instance has returned. p_0's *Dequeue* instance returns no later than time $t+|Dequeue|$, by definition, which is strictly less than $t+d$. Together, we see that p_0 learns about a remote *Dequeue* invocation no sooner than $t+s+d \geq t+d > t+|Dequeue|$, so through the return of its *Dequeue* instance, p_0 cannot distinguish S_3 from a run in which that is the only *Dequeue* instance. Thus, it returns the same value, which by set-linearization is necessarily 1. Similarly, p_1 must return a value in $\{1,2\}$, since it cannot learn about the *Dequeue* instance at p_2 until time $t+2s+d$, which is larger than when its own *Dequeue* instance returns by $t+s+|Dequeue|$.

Next, we want to show that p_2 will also return a value from $\{1,2\}$ to its *Dequeue* instance. We cannot directly argue this, since if p_2 learns about both the *Dequeue* instances at p_0 and p_1 before it generates a return value for its own, it may decide to return a different value than either. Instead, we will shift events at p_1 earlier, then argue that in this run, the information about p_0's *Dequeue* instance does not arrive at p_1 until after it has generated its *Dequeue* return value, forcing p_1 to return 1 to its *Dequeue* instance. Now, while p_1 may be able to distinguish this new run from S_3, we will argue that p_2 will not be able to distinguish them until after it generates its *Dequeue* return, so must return the same value in both. In the shifted run, p_0 and p_1 will both return 1, which means that p_2 must return either 1 or 2 to satisfy set-linearizability.

We will shift S_3 by the vector $\langle 0, -X, 0, \ldots, 0\rangle$, where X is a value we will determine shortly. Next, we will alter message delays, both to delay p_1 from learning about p_0's *Dequeue* instance and to make the shifted run admissible, yielding run S_3^X.

Our first step is to find what shift amounts X for p_1 will make S_3^X admissible, then argue the behavior of each process. First, note that this shift will increase the local clock offset of p_1 by X. In S_3, $c_1 = \frac{1}{n}u$, the smallest clock offset is $c_0 = 0$ and the largest is $c_{n-1} = \frac{n-1}{n}u$. To keep the run admissible, we must have $X \leq \left(\frac{n-1}{n}u - \frac{1}{n}u\right) = \frac{n-2}{n}u$, since we are not changing the smallest clock offset, so must keep c_1 within ε of that offset.

Next, as shown in Table 1, we see that for a non-negative value of X, we will have some inadmissible message delays in $\mathit{Shift}(S_3, \langle 0, -X, 0, \ldots, 0\rangle)$ (highlighted in the $\mathit{Shift}(S_3)$ column). To correct these, we trim the run before any of the inadmissible messages would arrive, then extend the run with other, admissible message delays (highlighted in the S_3^X column), following the technique introduced in [19]. Unlike a shift, this may change the behavior of the run, so we must argue what each process does in run S_3^X. We chose these new delays to delay processes from learning about remote actions, setting all of the adjusted delays to the maximum, d. Since $X \leq \frac{n-2}{n}u < u$, then the delays not highlighted in the S_3^X column are in the range $[d-u, d]$, and we conclude that if $0 \leq X \leq \frac{n-2}{n}u$, then S_3^X is admissible.

Table 1 Table showing message delays to and from p_1 in runs for base case in repeated *Dequeue* return values proof. Delays for other processes are unchanged across the three runs.

Message Path	S_3	$Shift(S_3)$	Adjusted: S_3^X
$p_0 \to p_1$	$d-u$	$d-u-X$	d
$p_1 \to p_0$	d	$d+X$	d
$p_1 \to p_2$ (initially)	$d-u$	$d-u+X$	$d-u+X$
$p_1 \to p_2$ (after t_1^*)	d	$d+X$	d
$p_1 \to p_{\geq 3}$	$d-u$	$d-u+X$	$d-u+X$
$p_{\geq 2} \to p_1$	d	$d-X$	$d-X$

Now that we know what values of X make S_3^X an admissible run, we will find which of those values of X will make all three *Dequeue* instances return values from $\{1,2\}$ in S_3^X.

p_0 will not learn about p_2's *Dequeue* instance until after its own returns, by the same argument as in S_3. We want p_0 to also not learn about p_1's *Dequeue* instance until after its own returns. p_1 invokes *Dequeue* at $t+s-X$ in S_3^X, since we shifted events at p_1 earlier by X. A message sent at this time will arrive at p_0 at time $t+s-X+d$, and we want to argue that this will be after $t+|Dequeue|$, and thus after p_0's *Dequeue* instance returns. This happens if and only if $d+s-X > |Dequeue|$, or $X < d+s-|Dequeue|$. Since $s \geq 0$, it is sufficient to require that $X < d-|Dequeue|$ to ensure that p_0's *Dequeue* instance returns 1.

To force p_1's *Dequeue* instance to return 1, we want information about p_0's invocation of *Dequeue* to arrive after p_1 generates its *Dequeue* return value. Thus, we want to have time $t+d$ (since messages from p_0 to p_1 have delay d in S_3^X) later than when p_1 generates a return value. p_1 invokes *Dequeue* at time $t+s-X$ and the *Dequeue* instance returns at most $|Dequeue|$ time after invocation, so we want to have $t+d > t+s-X+|Dequeue|$. Solving for X, we find that this is true iff $X > |Dequeue|+s-d$. Here, we split into cases depending on the value of s: If $s=0$, we want $X > |Dequeue|-d$, but we assumed that $|Dequeue| < d$, so any non-negative value of X is sufficient. If $s = Q-u$, we want $X > |Dequeue|+(Q-u)-d = |Dequeue|+Q-(d+u)$.

Similarly to previous arguments, since p_2 invokes *Dequeue* at least X after p_1 does (p_2 invokes *Dequeue* s after p_1 in S_3, which means $s+X$ after in S_3^X), and message delays from p_2 to p_1 are $d-X$, p_1 cannot learn about p_2's *Dequeue* invocation until at least $X+(d-X) = d > |Dequeue|$ after p_1 invokes *Dequeue*. This is after p_1 generates its *Dequeue* return value. Combining this with the previous conclusion that p_1 is unaware of p_0's *Dequeue* invocation until after it chooses a return value, we conclude that p_1 will return the same value as in a run where neither p_0 nor p_2 invoked *Dequeue*. The only legal set-linearization of such a run requires that p_1 return 1.

We can now reason about p_2's behavior. Since both p_0 and p_1 must return 1 to their *Dequeue* instances in S_3^X, we conclude that p_2 must return either 1 or 2, as there is no legal set-linearization of any other return value. We will thus argue that p_2 cannot distinguish S_3^X from S_3 until after it generates its *Dequeue* return value, concluding that p_2 will return either 1 or 2 to its *Dequeue* instance in S_3 as well.

Consider when each process in S_3 can first distinguish that it is not in S_3^X. These differences correspond to the adjusted message delays highlighted in the final column of Table 1. p_0 can distinguish the runs when it does not receive a message p_1 may have sent at its *Dequeue* invocation as soon as it would have received it in S_3^X, since in S_3^X we reduced the delay on messages from p_1 to p_0. This detection would occur at time $t+s+(d-X)$, when that message does not arrive. p_1 can first distinguish the runs at time $t+(d-u)$, when it can receive a message p_0 sent at its *Dequeue* invocation but which arrives later in S_3^X, where we increased the delay on messages from p_0 to p_1. Note $t + s + (d - X) + (d - u) > t + (d - u)$ and $t + s + (d - X) \leq t + (d - u) + d$, so neither process can send a message after it detects the difference which will arrive before the recipient detects the difference itself.

Finally, p_2 can distinguish the runs either by receiving a message p_0 or p_1 sends after distinguishing the runs or directly from adjusted message delays. We argue that each of these must occur after the *Dequeue* instance at p_2 returns, so p_2 cannot distinguish S_3 from S_3^X until after that *Dequeue* instance's return value is set, so the value must be the same in both runs.

Consider when p_2 can receive a forwarded detection of a difference in the runs:

- p_0 can send this information no sooner than $t + s + (d - X)$, and the message would take d time to arrive, meaning that the earliest p_2 could distinguish the runs based on this information is $t + s + 2d - X$. We want to show that this is greater than $t + 2s + |Dequeue|$, and thus after p_2's *Dequeue* instance returns. This is true iff $2d - X - u > s + |Dequeue|$. Consider cases for the value of s:
  - $s = 0$: We want to show that $2d - X - u > |Dequeue|$. This is true if and only if $X < (d - |Dequeue|) + (d - u)$, but we know that $d \geq u$ so this holds if $X < d - |Dequeue|$.
  - $s = Q - u$: We want to show that $2d - X - u > |Dequeue| + Q - u$. This is true if and only if $X < (d - Q) + (d - |Dequeue|)$. Since $Q \leq d$ and we are already assuming $X < d - |Dequeue|$, this inequality holds.
- p_1 can send a message informing p_2 that it is in S_3, not S_3^X, no sooner than $t + (d - u)$. Since $t + (d - u) = t_2^*$, this message will take d time to arrive at p_2. We want to show that this is after p_2's *Dequeue* instance returns, which happens at $t + 2s + |Dequeue|$. Thus, we want $t + (d - u) + d > t + 2s + |Dequeue|$, or $2d - u > 2s + |Dequeue|$. Solving for $|Dequeue|$, this is equivalent to $|Dequeue| < 2d - 2s - u$. Consider the possible values of s:
  - $s = 0$: We want to show that $|Dequeue| < 2d - u$. But we know that $d \geq u$, so $d - u \geq 0$ and $|Dequeue| < d$, so this inequality holds.
  - $s = Q - u$: We want to show that $|Dequeue| < 2d - 2(Q - u) - u = 2d - 2Q + u$. But $|Dequeue| < Q$, so it is sufficient to show that $Q \leq 2d - 2Q + u$. This holds iff $Q \leq \frac{2d+u}{3}$. But we assumed $Q \leq \frac{3d+2u}{5} \leq \frac{2d+u}{3}$, so we have the desired relationship.

  Thus, p_2 cannot learn from p_1 that it is in S_3^X before it generates a return value for its *Dequeue* instance.

Finally, we show that p_2 cannot directly differentiate S_3 from S_3^X based on the altered message delays in S_3^X before its *Dequeue* instance returns. At the earliest, this can happen at $t_2^* + d - X$, when p_2 does not receive a message in S_3 that it may have in S_3^X, since in S_3^X we decreased the delay of messages p_1 sends to p_2 at or after time $t_2^* - X$. We again want to show that this is after p_2's *Dequeue* instance returns which happens no later than $t + 2s + |Dequeue|$. That is, we want $t_2^* + d - X = t + (d - u) + d - X > t + 2s + |Dequeue|$. Equivalently, we want $2d - u - X > 2s + |Dequeue|$. Consider cases for s:

- $s = 0$: In this case, we want $2d - u - X > |Dequeue|$, which is true iff $X \leq (d - |Dequeue|) + (d - u)$. We already have the constraint that $X < d - |Dequeue|$ and $u \leq d$.
- $s = Q - u$: Here, we want $2d - u - X > 2(Q - u) + |Dequeue|$, which is true if $X < 2d + u - 2Q - |Dequeue|$. This is a new constraint on X which we must meet.

Thus, in all cases (if X meets all our constraints simultaneously), p_2 cannot distinguish S_3 from S_3^X until after its *Dequeue* instance returns. This means it returns the same value in both runs, and since we proved it returns a value from $\{1, 2\}$ in S_3^X, it also does in S_3.

Our last step is to verify that our constraints on X are compatible–that there is a value of X which will make S_3^k admissible and give the behavior we want. Our constraints are

- $X \geq 0$ and $X > |Dequeue| + Q - (d+u)$
- $X < d - |Dequeue|$, $X < 2d + u - 2Q - |Dequeue|$, and $X \leq \frac{n-2}{n}u$

These three upper bounds and two lower bounds lead to six cases to check to show that there exists a value of X which satisfies all of our constraints.

- $d - |Dequeue| > 0$: By assumption, $|Dequeue| < d$, so $d - |Dequeue| > 0$.
- $d - |Dequeue| > |Dequeue| + Q - (d+u)$: This is true iff $2d + u > 2|Dequeue| + Q$. Since $|Dequeue| < Q$, it is sufficient to show that $2d + u \geq 3Q$, or $Q \leq \frac{2d+u}{3}$, but we assumed that $Q \leq \frac{3d+2u}{5} \leq \frac{2d+u}{3}$, so this relationship holds.
- $2d + u - 2Q - |Dequeue| > 0$: This is equivalent to the previous case.
- $2d+u-2Q-|Dequeue| > |Dequeue|+Q-(d+u)$: This is true iff $3d+2u > 3Q+2|Dequeue|$, but it is sufficient to show that $3d + 2u \geq 5Q$, and we assumed that $Q \leq \frac{3d+2u}{5}$, so this relationship holds.
- $\frac{n-2}{n}u \geq 0$: $n > 2$ and $u \geq 0$, and a positive fraction of u will thus be non-negative.
- $\frac{n-2}{n}u > |Dequeue| + Q - (d+u)$: Solving for $|Dequeue|$ and Q, this is true iff $Q + |Dequeue| < d + \frac{n-2}{n}u + u$. Since $|Dequeue| < \frac{d}{2} + u$, there is some N_0 s.t. for all $n \geq N_0$, $|Dequeue| < \frac{d}{2} + \frac{n-2}{n}u$. Further $|Q| \leq \frac{d}{2} + u$, so combining these, the inequality holds for $n \geq N_0$.

Thus, since every upper bound is larger than every lower bound, for sufficiently large n ($n \geq N_0$), there exists at least one X such that S_3^X is admissible and p_0, p_1, and p_2 all return values from $\{1, 2\}$ to their *Dequeue* instances, and we have the claim.

Inductive Case. Assume that for some arbitrary $4 \leq k \leq n$, all *Dequeue* instances in S_{k-1} return values from the set $\{1, \ldots, k-2\}$. We will show that in S_k and S'_k, all *Dequeue* instances return values from the set $\{1, \ldots, k-1\}$. First, we will use S_{k-1} to argue the behavior of S'_k, then use that behavior to prove the behavior of S_k.

To show that in S'_k, all processes return values from the set $\{1, \ldots, k-1\}$ to their *Dequeue* instances, we argue that processes $p_0, \ldots, p_{k-1}$ cannot distinguish S_{k-1} from S'_k until after they have all generated their *Dequeue* return values. Thus, they will return the same values as in S_{k-1}, which are all in $\{1, \ldots k-2\}$ by the inductive hypothesis. We can then conclude that p_{k-1}, which invokes a *Dequeue* instance in S'_k but not in S_{k-1} must return a value in the set $\{1, \ldots, k-1\}$ to satisfy set-linearizability.

Recall that S'_k differs from S_{k-1} in two ways: First, p_{k-1} invokes *Dequeue* at time $t + (k-1)s$. Second, messages from p_{k-2} to p_{k-1} sent at or after $t^*_{k-2} = t + (k-2)(d-u)$ have delay d instead of $d - u$. Thus, the first point at which any process can discern that it is in S'_k instead of S_{k-1} is p_{k-1} at whichever of these events happens first. For any other process, the first point where it can distinguish the runs is when it can receive a message p_{k-1} sends after it discerns the difference. We will argue that such a message arrives at any of $p_0, \ldots, p_{k-2}$ after it has chosen a return value for its *Dequeue* instance. Note that we need only prove that such a message arrives at p_{k-2} more than $|Dequeue|$ after it invokes *Dequeue*, since each process with a lower index invokes *Dequeue* at the same time or sooner, and the message delay from p_{k-1} to any lower-index process is the same. We proceed by cases on which distinguishing event at p_{k-1} occurs first.

- p_{k-1} first distinguishes the runs when it invokes *Dequeue*: The message delay from p_{k-1} to p_{k-2} is d, and any indirect path would take even longer, since any such path must have some message from a higher-indexed to lower-indexed process, which has delay d. Thus, the earliest p_{k-2} can distinguish the runs is $t + (k-1)s + d$. We want to show

that this is later than the return time of p_{k-2}'s *Dequeue* instance, which must return by $t+(d-2)s+|Dequeue|$. This inequality is true iff $t+(k-1)s+d > t+(k-2)s+|Dequeue|$, which reduces to $s+d > |Dequeue|$.
Since $d > Q$ and $s \geq 0$, this inequality holds, which means that p_{k-2} (and similarly $p_0, \dots, p_{k-3}$) cannot use the extra *Dequeue* invocation at p_{k-1} to distinguish S'_k from S_{k-1} until after their *Dequeue* instances have returned.

- p_{k-1} first distinguishes the runs when it fails to receive a message whose delay was increased: The earliest possible sending time of such a message is $t^*_{k-2} = t+(k-2)(d-u)$. p_{k-1} can detect that it has not arrived $d-u$ later (when it would have arrived in S_{k-1}), and then the earliest it can get information about the differentiation to a lower-indexed process is another d after that. We similarly want to show that this is after the *Dequeue* instance at p_{k-2} returns, which is true iff $t+(k-2)(d-u)+(d-u)+d > t+(k-2)s+|Dequeue|$, which reduces to $(k-1)(d-u)+d > (k-2)s+|Dequeue|$.
Since $d > |Dequeue|$, $d \geq u$, and $s = \max\{0, Q-u\}$, we see that $d-u \geq s$, so the inequality holds. Thus, in this case no process in $p_0, \dots, p_{k-2}$ can distinguish S'_k from S_{k-1} until after it has generated a return value for its *Dequeue* instance.

Since in neither case can $p_0, \dots, p_{k-2}$ distinguish S'_k from S_{k-1} until after its *Dequeue* instance returns, all their *Dequeue* instances return the same values in both runs. Specifically, by the inductive hypothesis they all return values from the set $\{1, \dots, k-2\}$. The *Dequeue* instance at p_{k-1} must then return a value in the set $\{1, \dots, k-1\}$, as any larger value would violate set-linearizability, since there would be no *Dequeue* instance returning $k-1$.

Now, having determined the behavior of S'_k, we use it to show that S_k will behave similarly. This is another indistinguishability argument, showing that p_{k-1} cannot distinguish S_k from S'_k, until after it has generated a return value for its *Dequeue* instances. Recall that the difference between S_k and S'_k is that in R_k all messages from p_{k-3} to p_{k-2} have delay $d-u$, while in S'_k, those sent at or after t^*_{k-3} have delay d.

Before we start the indistinguishability argument, note that if p_k did not invoke *Dequeue* in S_k, the remaining $k-1$ *Dequeue* instances must return values from the set $\{1, \dots, k-1\}$, since there would only be $k-1$ instances, so there would be no way to set-linearize an instance that returned a larger value. These processes must behave the same way in S_k as in this run, since the first point where any could detect a difference would be d after p_k's invocation, which is after all other *Dequeue* instances have returned, similar to prior arguments. Thus, we need only concern ourselves with showing that p_{k-1} cannot distinguish S_k from S'_k before its *Dequeue* instance returns, so that it will return a value in $\{1, \dots, k-1\}$, as we proved it does in S'_k.

The only process which can directly detect a difference between S_k and S'_k is p_{k-2}, when it receives a message in S_k which arrives sooner than it could in S'_k. This occurs $d-u$ after time t^*_{k-3}, when the message delays changed. The soonest p_{k-1} can learn about the difference is when a message from p_{k-2}, sent after it detected the difference, could arrive. But $t^*_{k-3} + (d-u) = t^*_{k-2}$, so any message p_{k-2} sends to p_{k-1} after this point has delay d. Thus, the soonest p_{k-1} can distinguish S_k from S'_k is $t^*_{k-2} + d = t+(k-2)(d-u)+d$. We argue that this is after p_{k-1} generates its *Dequeue* return value, which occurs no later than $t+(k-1)s+|Dequeue|$. We thus want to show that $t+(k-2)(d-u)+d > t+(k-1)s+|Dequeue|$. Consider the cases for s:

If $s=0$: The inequality holds iff $(k-2)(d-u)+d > |Dequeue|$, which is true because $d \geq u$ and $d > |Dequeue|$. If $s = Q-u$: The inequality holds if $(k-2)(d-u)+d > (k-1)(Q-u)+|Dequeue|$, or $(k-1)d > (k-1)Q - u + |Dequeue|$. Since $|Dequeue| < Q$, it is sufficient to show that $(k-1)d \geq kQ - u$, or $Q \leq \frac{(k-1)d+u}{k}$.

To prove this final inequality, recall that $Q \leq \frac{3d+2u}{5}$ and that $k \geq 4$. For all $k \geq 4$, $\frac{(k-1)d+u}{k} \geq \frac{3d+u}{4}$, so it suffices to show that $\frac{3d+2u}{5} \leq \frac{3d+u}{4}$. This follows because $\frac{3d+2u}{5} = \left(\frac{3d+u}{4}\right)\left(\frac{4}{5}\right) + \frac{u}{5}$, and $\frac{u}{5} \leq \left(\frac{1}{5}\right)\left(\frac{3d+u}{4}\right)$, as that inequality reduces to $u \leq d$, which is true.

We conclude that p_k cannot distinguish S_k from S'_k until after it generates a return value for its *Dequeue* instance, so it must return the same value in both runs, which we previously proved was in the set $\{1, \ldots, k-1\}$. Thus, by mathematical induction, when $k = n$, all *Dequeue* instances in S_n return values from the set $\{1, \ldots, n-1\}$, and we have the claim. ◀

6 Contradiction

Let us quickly recap what we have shown so far. First, we showed that there is a run D_n with n overlapping *Dequeue* instances each returning a different value. Then, we (somewhat laboriously) showed that there is a run S_n with n overlapping *Dequeue* instances in which two *Dequeue* instances return the same value. Now, we want to show that these runs are indistinguishable, a contradiction, as processes must return the same values in indistinguishable runs.

▶ **Theorem 8.** *There is no uniform, set-linearizable implementation of a multiplicity queue with* $|Dequeue| < \min\left\{\frac{d}{2} + u, \frac{3d+2u}{5}\right\}$.

Proof. Assume, in contradiction, that there is such an algorithm. Then the conditions for Lemma 4 and Lemma 7 are satisfied, so we know that S_n and D_n exist, where all *Dequeue* instances in S_n return values from $\{1, \ldots, n-1\}$ and the *Dequeue* instance at p_i in D_n returns $i+1$, for all $0 \leq i < n$. Recall that S_n requires that $n \geq N_0$, defined in Section 5 s.t. for all $n \geq N_0$, $|Dequeue| < \frac{d}{2} + \frac{n-2}{n}u$.

Note that S_n and D_n are nearly identical–they have the same initial sequence of *Enqueue* instances at p_0, the same clock offsets ($c_0 = 0$, $c_i = \frac{i}{n}u, 1 \leq i < n$), and the same *Dequeue* invocations (p_i invokes *Dequeue* at time $t + (i-1)s$). The two runs also have nearly identical message delays, where if $0 \leq i < j < n$, messages from p_j to p_i have delay d and those from p_i to p_j have delay $d-u$, except that in S_n, messages from p_{n-2} to p_{n-1} sent at or after time t^*_{n-2} have delay d. Thus, if we extend those message delays in D_n, we will have the same run. We will argue that we will still have D_n's behavior, which differs from S_n's, in the same run, which is a contradiction.

Suppose first that $u < d$. Construct D^* from D_n by delaying all messages from p_{n-2} to p_{n-1} sent at or after t^*_{n-2} by d. We argue that no process can distinguish that it is in D^* instead of D_n before its *Dequeue* instance returns. The first point where any process could distinguish the two runs is when a message p_{n-2} sends at t^*_{n-2} does not arrive at p_{n-1} at the same time in D^* it would in D_n, because we extended its delay. Thus, the first time a process can distinguish the two runs is $t^*_{n-2} + d - u = t + (n-2)(d-u) + (d-u) = t + (n-1)(d-u)$. We argue that, for sufficiently large n, this is after p_{n-1}'s *Dequeue* instance returns. That happens at or before $t + (n-1)s + Q$. We thus want $t + (n-1)(d-u) > t + (n-1)s + Q$, which is true iff $(n-1)(d-u) > (n-1)s + Q$. Consider the possible values of s by cases:

- $s = 0$: We want to show that $(n-1)(d-u) > Q$. This is true when $n > \frac{Q}{d-u} + 1$. Since $d > u$, this is true for sufficiently large n. Let N_1 be such that for all $n \geq N_1, n > \frac{Q}{d-u} + 1$.
- $s = D - u$: We want to show that $(n-1)(d-u) > (n-1)(Q-u) + Q$. This is true when $(n-1)d - (n-1)u > nQ - (n-1)u$, or $(n-1)d > nQ$. If we solve for n, we have $n(d-Q) - d > 0$, or $n > \frac{d}{d-Q}$, since $d - Q > 0$. Again, this is true for sufficiently large n, so let N_2 be such that for all $n \geq N_2$, $n > \frac{d}{d-Q}$.

Thus, in runs with sufficiently large n (at least $\max\{N_0, N_1, N_2\}$), p_{n-1} cannot distinguish that it is in D^*, not D_n, until after its *Dequeue* instance has returned. Similarly, no other process can distinguish the runs before its *Dequeue* instance returns, as those returns occur by $t + (i)s + Q \leq t + (n-1)s + Q$ for $0 \leq i < n$, so there is not time for p_{n-1} to inform any other process of the discrepancy since by the time p_{n-1} discovers it, all other processes' *Dequeue* instances have already returned.

Next, we have the case where $u = d$. In this case, observe that $t^*_{n-2} = t+(n-2)(d-u) = t$, so all messages from p_{n-2} to p_{n-1} starting at t have delay d. Thus, p_{n-1} can distinguish the runs at $t + (d-u) = t$, which is before its *Dequeue* instance returns.

Instead, we can use a reduction argument to disprove the existence of an algorithm performing better than our bound. Choose a new message uncertainty $u' = \frac{d+|Dequeue|}{2}$, noting that this gives $0 < u' < d$. Now, since our assumed algorithm correctly implements a multiplicity queue in a system with message delays in the range $[0, d]$ with $|Dequeue| < \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\} = d$, it must also correctly implement that multiplicity queue in a system with message delays $[d - u', d]$, since any run possible in that system is possible in the system where $d = u$ since the range of possible message delays is completely contained in $[d - u, d]$. It thus implements multiplicity queues in a system with message uncertainty u' with $|Dequeue| < d = 2u' - |Dequeue|$. Then $|Dequeue| < u' < \min\left\{\frac{3d+2u'}{5}, \frac{d}{2} + u'\right\}$ because $d > u'$. But this contradicts the impossibility of such an algorithm as proved in the $u < d$ case above, so our assumed algorithm cannot exist. ◀

Note that the $n \geq \max\{N_0, N_1, N_2\}$ constraint is the only place we need the assumption of uniform algorithms. This shows that our proof applies not only to uniform algorithms, but to any algorithm on at least that many processes. However, we state the result as for uniform algorithms to get a result applicable to any system, as we have not excluded higher performance of non-uniform algorithms in small systems.

Finally, we note that our result is an improvement over the previously best known bound of $|Dequeue| \geq \min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$ [13], with the added restriction to uniform algorithms. This claim follows from elementary algebra, as $\frac{d+u}{2} = \frac{d}{2} + \frac{u}{2} < \frac{d}{2} + u$ and $\frac{d+u}{2} \leq \frac{2.5d+2.5u}{5} \leq \frac{3d+2u}{5}$, since $u \leq d$.

▶ **Corollary 9.** *Any uniform, set-linearizable implementation of a multiplicity queue must have* $|Dequeue| \geq \frac{d+u}{2} \geq \min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$.

7 Partial Tightness

While it may seem that the $\frac{d}{2} + u$ term in the lower bound is an artifact of our limited proof techniques for lower bounds, and future work may increase the bound to $\frac{3d+2u}{5}$ or better for all values of u, we here outline an algorithm for the special case where $u = 0$ which matches the $\frac{d}{2} + u = \frac{d}{2}$ lower bound, beating $\frac{3d}{5}$. This suggests $\frac{d}{2} + u$ may be somehow fundamental, despite not holding everywhere.

The idea of the algorithm is to have all processes maintain a local copy of the queue, which they update based on messages about operation invocations. For every operation invocation, the invoking process broadcasts operation and arguments immediately, then returns after $d/2$ time. Thus, if two instances are concurrent, neither can learn about the other, since messages take d time to arrive. If they are non-concurrent, then there is more than d time from the invocation of the first instance to the return of the second instance, so at the end of a *Dequeue* instance the invoking process must know about any strictly-preceding instances. Each process will execute every *Enqueue* instance on its local copy, $d/2$ after invocation at the invoking process and d after invocation at every other process, when it receives the message. Non-invoking processes will only locally execute *Dequeue* instances if they have not already seen another *Dequeue* instance concurrent with it. If they have, detected by timestamps, then they have already removed the return value from their local copy, so there is no further work to do. Full pseudocode for the algorithm appears in the appendix, along with the proof of the following theorem.

▶ **Theorem 10.** *If $u = 0$, there is a uniform, set-linearizable implementation of a multiplicity queue with $|Dequeue| = d/2$.*

8 Conclusion

We developed a new combination of shifting and other indistinguishability arguments to prove a larger lower bound of $|Dequeue| \geq \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\}$ in uniform multiplicity queue implementations. This both improves the state of the art and suggests ways to improve the bound further. For example, strengthening the base case for Lemma 7 in Section 5 should improve the $\frac{3d+2u}{5}$ portion of the lower bound. We hypothesize that this may increase to approach a limit of $|Dequeue| \geq d$ for all non-zero values of u, which seems an intuitive value. If that is true, our tightness result that $|Dequeue| = d/2$ is possible when $u = 0$ is more interesting, as it suggests the bounds may be discontinuous. We continue exploring these bounds to understand multiplicity queues, and then use that understanding to design and understand other data type relaxations.

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, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

3 Armando Castañeda and Miguel Piña. Fully read/write fence-free work-stealing with multiplicity. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 16:1–16:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.16.

4 Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. Relaxed queues and stacks from read/write operations. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 13:1–13:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.OPODIS.2020.13.

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 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

7 Colette Johnen, Adnane Khattabi, and Alessia Milani. Efficient wait-free queue algorithms with multiple enqueuers and multiple dequeuers. In Eshcar Hillel, Roberto Palmieri, and Etienne Rivière, editors, *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, volume 253 of *LIPIcs*, pages 4:1–4:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.OPODIS.2022.4.

8 Pankaj Khanchandani and Roger Wattenhofer. On the importance of synchronization primitives with low consensus numbers. In Paolo Bellavista and Vijay K. Garg, editors, *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 18:1–18:10. ACM, 2018. doi:10.1145/3154273.3154306.

9 Martha J. Kosa. Time bounds for strong and hybrid consistency for arbitrary abstract data types. *Chic. J. Theor. Comput. Sci.*, 1999, 1999. URL: http://cjtcs.cs.uchicago.edu/articles/1999/9/contents.html.

10 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.

11 Gil Neiger. Set-linearizability. In James H. Anderson, David Peleg, and Elizabeth Borowsky, editors, *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, page 396. ACM, 1994. doi:10.1145/197917.198176.

12 Nir Shavit and Gadi Taubenfeld. The computability of relaxed data structures: queues and stacks as examples. *Distributed Comput.*, 29(5):395–407, 2016. doi:10.1007/s00446-016-0272-0.

13 Edward Talmage. Lower bounds on message passing implementations of multiplicity-relaxed queues and stacks. In Merav Parter, editor, *Structural Information and Communication Complexity - 29th International Colloquium, SIROCCO 2022, Paderborn, Germany, June 27-29, 2022, Proceedings*, volume 13298 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 2022. doi:10.1007/978-3-031-09993-9_14.

14 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.

15 Edward Talmage and Jennifer L. Welch. Relaxed data types as consistency conditions. *Algorithms*, 11(5):61, 2018. doi:10.3390/a11050061.

16 Edward Talmage and Jennifer L. Welch. Anomalies and similarities among consensus numbers of variously-relaxed queues. *Computing*, 101(9):1349–1368, 2019. doi:10.1007/s00607-018-0661-2.

17 Anh Tran and Edward Talmage. Brief announcement: Improved, partially-tight multiplicity queue lower bounds. In Rotem Oshman, Alexandre Nolin, Magnús M. Halldórsson, and Alkida Balliu, editors, *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023*, pages 370–373. ACM, 2023. doi:10.1145/3583668.3594602.

18 Anh Tran and Edward Talmage. Improved and partially-tight lower bounds for message-passing implementations of multiplicity queues, 2023. doi:10.48550/arXiv.2305.11286.

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

A Appendix

A.1 Proofs Omitted from Paper Body

▶ **Lemma 11.** *For all $2 \leq k < n$, $D_k = Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, where $\overrightarrow{s_{k-1}}$'s only non-zero component is $-u$ at index $k-1$: $\overrightarrow{s_{k-1}} = \langle 0, \dots, 0, -u, 0, \dots, 0 \rangle$.*

Proof. Let k be an arbitrary value with $2 \leq k < n$. Consider what happens when we shift D_{k-1} by $\overrightarrow{s_{k-1}}$. All events at p_{k-1} occur u earlier in real time, so p_{k-1} invokes *Dequeue* at time $t + ((k-1)-1)s + (s+u) - u = t + (k-2)s + s = t + (k-1)s$, which matches the definition of D_k. Let $0 \leq i < k-1 < j < n$. Message delays in D_{k-1} from p_{k-1} to p_i were $d-u$, and from p_i to p_{k-1} were d. In the other direction, messages delays from p_{k-1} to p_j were $d-u$ and from p_j to p_{k-1} were d. When we shift the send and receive events at p_{k-1} earlier, messages from p_{k-1} have a longer delay by u and messages to p_{k-1} have a shorter delay u. We see that this leaves all delays from p_{k-1} to another process at d and all delays to p_{k-1} at $d-u$, which are admissible. Since we only shifted one process, messages between other processes are unchanged.

Finally, we consider clock offsets. c_{k-1} is $\left(\frac{(k-1)-n}{n}\right)u$ in D_{k-1}, and must increase by u to hide the difference in real time when we shift. Thus, in $Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, $c_{i-1} = \left(1 + \frac{(k-1)-n}{n}\right)u = \left(\frac{(k-1)}{n}\right)u$, matching the specification for D_k. ◀

A.2 Partial Tightness: Special Case Upper Bound

The algorithm is event-driven, where each process can react to operation invocations, message receptions, and expiration of local timers it sets. Because $u = 0$, every message takes exactly d time to arrive. Thus, since the algorithm broadcasts every message, when any process receives a message, it knows all other processes receive the same message at the same time. Further, since there is no uncertainty, the maximum clock skew is $(1 - 1/n)0 = 0$, so every process' local clock (read by the function *localClock*()) is equal to real time. We thus let every operation instance take $d/2$ time. By the message delay and operation instance duration, a process learns about an instance at another process before it returns to an instance at itself if and only if that remote instance returned before the local one's invocation, so applying remote operations to the local copy of the structure immediately upon receipt and choosing *Dequeue* return values $d/2$ after invocation together keep the local copies synchronized and choose correct values.

Let R be an arbitrary run of Algorithm 1. Observe that every invocation in R either has a matching response, $d/2$ after invocation. We define a set-linearization of R and prove that π respects real time order and is legal.

▶ **Construction 12.** *Place each Enqueue instance in a singleton set and define the set's timestamp as the pair of the invoking process' local clock read on line 3 plus $d/2$ and the invoking process' id. For each Dequeue return value x, place all Dequeue instances which return x in a set, and define the set's timestamp as the smallest timestamp of any instance in the set, where a Dequeue instance's timestamp is the pair of the local clock read in line 5 and the invoking process' id, with the id breaking ties between clock values. Let π be the sequence of these sets ordered by increasing timestamps (break ties by process id).*

▶ **Lemma 13.** *π respects the order of non-overlapping operation instances.*

Proof. Let op_1 and op_2 be any two non-overlapping operation instances, with op_1 invoked at p_i and returning before op_2's invocation at p_j. Since local clocks are exactly real time, and all instances have duration $d/2$, then op_1's timestamp will be more than $d/2$ smaller than op_2's. Thus, the only way that op_1 would not strictly precede op_2 in π is if they were in the same set, which could happen if they are both *Dequeue* instances which returned the same value x. But in that case, since op_1 returned before op_2's invocation and each of op_1 and op_2 took $d/2$ time between invocation and response, then p_j would receive the message p_i sent on line 5 at op_1's invocation before op_2 returns. This should have removed x from p_j's local copy of the queue, unless there were another element preceding x in p_j's local queue when op_2 returned. By the FIFO ordering of the multiplicity queue, this can only happen if there is a *Dequeue* instance which p_i applied before op_1 returned but p_j did not apply before op_2

Algorithm 1 Set-linearizable implementation of a multiplicity queue with $u = 0$. Code for each p_i.

```
    Initially: $localQueue$ is an empty FIFO queue, $mostRecentDequeue = 0$
 1: HandleInvocation ENQUEUE($arg$)
 2:     send $\langle enq, arg\rangle$ to all other processes
 3:     $setTimer(d/2, \langle enq, arg, \langle localClock(), i\rangle, return\rangle)$
 4: HandleInvocation DEQUEUE
 5:     send $\langle deq, ts = \langle localClock(), i\rangle\rangle$ to all other processes
 6:     $setTimer(d/2, \langle deq, ts\rangle)$
 7: HandleTimer EXPIRE($\langle enq, arg, ts, return\rangle$)
 8:     Generate $Enqueue$ response to user
 9:     $setTimer(d/2, \langle enq, arg, apply\rangle)$
10: HandleTimer EXPIRE($\langle enq, arg, apply\rangle$)
11:     $localQueue.enqueue(arg)$
12: HandleTimer EXPIRE($\langle deq, \langle clockVal, i\rangle\rangle$)
13:     Generate $Dequeue$ response to user with return value $localQueue.dequeue()$
14:     $mostRecentDequeue = clockVal$
15: HandleReceive $\langle enq, arg\rangle$
16:     $localQueue.enqueue(arg)$
17: HandleReceive $\langle deq, \langle clockVal, j\rangle\rangle$
18:     if $clockVal > mostRecentDequeue + d/2$ then
19:         $localQueue.dequeue()$
20:         $mostRecentDequeue = clockVal$
```

returned. Any *Dequeue* instance which p_i has applied before op_1 returns was either delivered to p_j at the same time as to p_i, and thus applied to p_j's local copy or invoked at p_i before op_1, but then by the time op_1 returns, by the fact that every *Dequeue* returns $d/2$ time after invocation, p_j would also receive and apply that *Dequeue* instance before op_2's invocation. Thus, there cannot be an element in p_j's local queue preceding x when it applies op_1, and op_2 cannot return x. ◀

▶ **Lemma 14.** *π is legal by the specification of a multiplicity queue.*

Proof. We proceed by induction on σ, a prefix of π. If σ is empty, then it is legal, as the empty sequence is always legal.

Suppose that $\sigma = \rho \cdot S$, where S is a set of operation instances. Assume that ρ is legal. We will show that σ is also legal by cases on S.

If $S = Enqueue(x)$, then σ is necessarily legal, as *Enqueue* does not return a value, so cannot be illegal.

If S is a set of *Dequeue* instances returning x, then we need to argue that the algorithm chose x correctly. Each invoking process chose the oldest value in its local copy of the queue as a return value, in line 13, so we merely need to argue that the local copy of the queue contains the elements enqueued and not dequeued in ρ, in order. Consider the *Dequeue* instance in S with the smallest timestamp, and call it d and its invoking process p_i. When p_i executes line 13 to generate d's return, it will have received every *Enqueue* invocation in ρ, as those were invoked at least $d/2$ before than this *Dequeue*, and added them to its local queue. The order of *Enqueue* instances in ρ matches their timestamp order, which is the order in which they are locally applied, since every process adds each *Enqueue* argument d

time after its invocation. When any other process p_j which has a *Dequeue* instance return the same value as d executes line 13 for that instance, it will have locally applied all *Enqueue* instances p_i has, and possible more. But any additional *Enqueue* instances will have larger timestamps, and thus follow *Enqueue*(x) in π, so would not be the correct return value for this *Dequeue* instance.

Thus, each process chooses x as the oldest-enqueued value in ρ which it has not already removed for another *Dequeue* instance. Such an instance must be in ρ, as another *Dequeue* instance at the same process would have a smaller timestamp and one at another process would not remove a value from the local queue until d after its invocation, which means it would have a smaller timestamp than this *Dequeue* instance which returns x.

Further, each process only removes values from its local queue when there is a *Dequeue* instance returning it. Suppose this were not so. Then some process p_k must have received a *Dequeue* instance which returned y and executed line 19 when it had already removed y from its local queue. But p_k could only remove y when it either returned y to its own *Dequeue* instance or received a message about another *Dequeue* instance. But either of those cases would update *mostRecentDequeue*, so the check on line 18 means that the two *Dequeue* instances which returned y had timestamps more than $d/2$ apart, which implies they were not concurrent, so they could not have returned the same value as that would imply they are in the same set in π, which is not possible by Lemma 13.

Finally, there cannot be a *Dequeue* instance returning x in ρ, as all instances returning x are in the set S. Thus, x is the argument of the first *Enqueue* instance in ρ which is not returned by a *Dequeue* instance in ρ. ◀

▶ **Theorem 15.** *If $u = 0$, Algorithm 1 is a uniform, set-linearizable implementation of a multiplicity queue with $|Dequeue| = d/2$.*

Proof. By Lemma 13, the sequence π we defined in Construction 12 respects the real-time order of non-overlapping instances. Lemma 14 proves that π is legal, so it is a legal set-linearization, proving by construction that Algorithm 1 is a set-linearizable implementation of a multiplicity queue. By lines 6 and 13, every *Dequeue* instance returns $d/2$ time after invocation, so $|Dequeue| = d/2$. Finally, the code for Algorithm 1 does not depend on n, so it is a uniform algorithm. ◀

Since this matches our lower bound of $|Dequeue| \geq \min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\} = \frac{d}{2}$ when $u = 0$, this algorithm is optimal and proves the bound is tight in this case.