6-2023

Brief Announcement: Improved, Partially-Tight Multiplicity Queue Lower Bounds

Anh Tran
Bucknell University

Edward Talmage
Bucknell University, elt006@bucknell.edu

Recommended Citation

Anh Tran and Edward Talmage. 2023. Brief Announcement: Improved, Partially-Tight Multiplicity Queue Lower Bounds. In ACM Symposium on Principles of Distributed Computing (PODC '23), June 19–23, 2023, Orlando, FL, USA. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583668. 3594602

Brief Announcement: Improved, Partially-Tight Multiplicity Queue Lower Bounds

Anh Tran
adt008@bucknell.edu
Bucknell University
Lewisburg, PA, USA

Edward Talmage
elt006@bucknell.edu
Bucknell University
Lewisburg, PA, USA

ABSTRACT

A *multiplicity queue* is a concurrently-defined data type which relaxes the conditions of a linearizable FIFO queue by allowing 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 and previous work has shown that multiplicity queues are computationally less complex than the unrelaxed version. Intriguingly, recent work has shown that there is, in fact, little possible speedup versus an unrelaxed queue. Seeking to understand this difference between intuition and real behavior, we increase the lower bound for uniform algorithms. Further, we outline a path toward building proofs for even higher lower bounds, hypothesizing that the worst-case time to *Dequeue* approaches maximum message delay, which is similar to the time required for an unrelaxed *Dequeue*. We also give an upper bound for a special case to show that our bounds are tight at that point. To achieve our lower bounds, we use extended shifting arguments, which have been rarely used but allow larger lower bounds than traditional shifting arguments. We use these in series of inductive indistinguishability proofs which allow us to extend our proofs beyond the usual limitations of shifting arguments. This proof structure is an interesting contribution independently of the main result, as developing new lower bound proof techniques may have many uses in future work.

CCS CONCEPTS

• **Software and its engineering → Abstract data types**; • **Theory of computation → Distributed algorithms**.

KEYWORDS

Distributed Data Structures, ADTs, Lower Bounds, Shifting Arguments, Multiplicity Queues

ACM Reference Format:
Anh Tran and Edward Talmage. 2023. Brief Announcement: Improved, Partially-Tight Multiplicity Queue Lower Bounds. In *ACM Symposium on Principles of Distributed Computing (PODC '23), June 19–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3583668.3594602

1 INTRODUCTION AND RELATED WORK

Relaxed data types [4] have risen as an efficient way to trade off some of the precise guarantees of an ordered data type for improved performance [11]. Multiplicity queues are a recent relaxation of queues [3] 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 their reduced computational complexity. In [3], Castañeda et al. implement multiplicity queues from $Read/Write$ registers, which is impossible for FIFO queues and most previous, sequential relaxations of queues [9, 12]. This means that it is possible to have queue-like semantics without the cost of strong primitive operations like $Read\text{-}Modify\text{-}Write$. Further work has shown that this allows interesting application in work-stealing [2] and that in shared memory systems with strong primitives, multiplicity queues are more efficient to implement than the best known algorithm for FIFO queues [5].

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 model inherent to geographically distributed systems [1]. In queue implementations, the need for concurrent *Dequeue* instances to wait long enough to learn about each other, so that they can return different values, is one of the primary reasons that *Dequeue* is expensive to implement, in terms of time delay from operation invocation to response [14]. 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 to return different values, it seems intuitive that multiplicity queues should be very efficient to implement in a message-passing system.

To the contrary, recent work [10] showed that there are limited performance gains possible in this model. In a partially-synchronous system with maximum message delay d and delay uncertainty u, that work showed that the worst-case delay from invocation to return for *Dequeue* is at least $\min\left\{\frac{2d}{3}, \frac{d+u}{2}\right\}$, at most a factor of 2 speedup over unrelaxed queues, where *Dequeue* can return after $d + (1 - 1/n)u$ time [14].

We here extend the work in [10], improving 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 $\min\left\{\frac{3d+2u}{5}, \frac{d}{2} + u\right\}$. Our bound is larger everywhere except for matching at the edge case of $u = 0$, which we show is tight. Intuitively, a *Dequeue* instance may not need to know about concurrent instances, but determining which instances are concurrent is inherently expensive. This insight could help develop more efficient algorithms or precise relaxations to provide minimal weakening while achieving performance improvements.

The proofs in [10] used shifting and indistinguishability arguments among three processes, which limited their results. We develop more complex indistinguishability arguments, using inductive definitions of different runs of the algorithm with all processes participating. This requires more advanced shifting tools, as developed in [14]. These allow us to prove larger bounds and are independently interesting as they may also enable larger lower bounds on other problems. The piecewise nature of our bound may also provide insight into the optimal lower bound. We show that the $\frac{d}{2}+u$ portion of the bound, which seems weaker, is tight in the edge case when all messages have delay d ($u = 0$). However, for larger uncertainties in message delay ($u > d/6$), the $\frac{3d+2u}{5}$ portion of the bound is higher. In fact, if one can strengthen our induction's base case, our proof structure may give bounds for larger u larger than $\frac{3d+2u}{5}$, perhaps even increasing more than linearly with u. This direction suggests that the optimal lower bound may in fact be as high as d for $u > 0$, and may be discontinuous at that point, which would be interesting. The base case is already the most complex portion of the proof, so such strengthening and finding an optimal lower bound remain as future work.

2 SYSTEM MODEL AND DEFINITIONS

We work in the same partially synchronous, message-passing, failure free model of computation as [10]. Lower bounds in this model apply in harder models, so a high lower bound here is still meaningful. There are n processes, $\{p_0, \ldots, p_{n-1}\}$, which participate in an algorithm implementing a shared memory object. Each process is a state machine with a local clock running at the same rate as real time, but potentially offset from real time and can set timers based on this clock. Users invoke data type operations at any process at any time, as long as the previous invocation at that process has received a response. Invocations, message arrivals, and timer expirations trigger state machine steps, which perform local computation, set timers, send messages, and generate operation responses. A *run* is a set of sequences of state machine steps, one for each process, each a valid state machine history with a real time for each step and either infinite or ending in a state with no unexpired timers and no messages sent to that process but not received. A run is *admissible* if there is a bijection between message send and receive steps with the *delay* between mapped sends and receives at least $d - u$ and at most d real time and the *skew*, or maximum difference between local clocks, is at most $\varepsilon := (1 - 1/n)u$ [7]. d and $u \leq d$ are known system parameters. A *uniform* algorithm is independent of the number of processes, using the same logic for all values of n.

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 invocation-response pairs, or *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 replaces the set of legal sequences of instances with 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 others. We are interested in *set-linearizable* implementations of set-sequential data types, as defined in [8] and [3], which requires that every admissible run of the algorithm has a total order of sets containing all operation instances in the run which 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 put those sets in 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 preceding the set containing op_2. The time cost of operation OP, denoted $|OP|$, is the largest time in any admissible run from invocation to response of any instance of OP.

We use *shifting* [6, 7, 14] in our proofs, a mechanism for creating indistinguishable runs. 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 the shifted run. To ensure that the runs are indistinguishable to the processes, local clock offsets c_i are changed to $c'_i = c_i - v[i]$. 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 also be admissible. Too large a shift vector causes message delays that are too long or short, preventing us from drawing conclusions about algorithm behavior. Wang et al. [14] extended the classic idea of shifting by showing 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, admissible delays. This new run is not, by default, indistinguishable from the original, but in some cases we can argue it is, to a certain point. We use this technique to exceed previous lower bounds based on classic shifting.

We consider a set-sequential data type based on a traditional FIFO queue called a *multiplicity queue*, defined in [3].

Definition 2.1. A *multiplicity queue* over value set V has 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 $\perp \notin V$ is a special value. A sequence of sets of *Enqueue* and *Dequeue* instances is legal if (i) every *Enqueue* instance is in a singleton set, (ii) *Dequeue* instances in the same 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 in a set-linearizable implementation of a multiplicity queue, any two concurrent *Dequeue* instances may, but do not necessarily, return the same value. Such instances would be placed 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. Note that we assume that all *Enqueue* arguments are unique, which is easily achieved by a higher abstraction layer timestamping the user's arguments.

3 LOWER BOUND PROOF OUTLINE

We prove our lower bound of $|Dequeue| \geq \min\{\frac{3d+2u}{5}, \frac{d}{2} + u\}$ by contradiction (see [13] for details), building up two sets of runs of an assumed algorithm which beats the bound. 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, returns a unique value. In the second set, we show that there are fewer distinct return values than *Dequeue* instances, so there must be *Dequeue* instances returning the same value. We then show that, for sufficiently large n, these two sets of runs converge, in that processes cannot distinguish which set they are in until after they choose return values for their *Dequeue* instances. Thus, they must have the same behavior in both, contradicting their different return values and proving our bound.

Every run we use starts with process p_0 sequentially executing the sequence $Enqueue(1) \cdot Enqueue(2) \cdots Enqueue(n)$. We then fix a time t_1 after the algorithm completes those. Thus, any set linearization of any of our runs will start with n singleton sets, enqueueing the values $1..n$ in order. All further operation instances will set-linearize after those *Enqueue* instances. In general in our runs, 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 in some runs, after a certain point, messages from p_{n-1} to p_n will also take d time. For sufficiently large n, this prevents p_n from collecting complete history information, which allows us to complete our indistinguishability proof.

We denote our first set of runs by D_k, $1 \le k \le n$ (D for "Distinct) and will show that all *Dequeue* instances return distinct values. In each D_k, the first k processes invoke *Dequeue* instances overlapping by u real time, and higher-indexed processes invoke *Dequeue* later. All messages from p_i to p_j have delay $d-u$ if $j \ge k > i$, $i < j < k$, or $k < i < j$ and d otherwise. We show that the *Dequeue* at p_k must return a different value from those at $p_0, \dots, p_{k-1}$, then shift the run to obtain D_{k+1}, which is indistinguishable. By induction, this is true for all $1 \le k \le n$, so all n *Dequeue* instances in D_n must return different values.

Lemma 3.1. *For all $2 \le k < n$, $D_k = Shift(D_{k-1}, \overrightarrow{s_{k-1}})$, where $\overrightarrow{s_{k-1}} = \langle 0, \dots, -u, \dots, 0 \rangle$, with $-u$ at index $k-1$.*

Lemma 3.2. *$\forall 0 \le i < n$, p_i's Dequeue instance in D_n returns $i+1$.*

For the second set of runs, we 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 pairs of more complex runs, with one more process joining the pattern and invoking *Dequeue* in each successive pair. When the induction reaches n, we show that we have a run, S_n, with n *Dequeue* instances returning only $n-1$ values.

We define the families of runs S_k, $0 \le k \le n$ and S'_k, $1 \le k < n$, in each of which only $k \le n$ processes invoke *Dequeue*. We inductively show that each of these has two *Dequeue* instances which return the same value, eventually showing that not all *Dequeue* instances in S_n return distinct values. In S_k, the first k processes invoke *Dequeue* exactly as in D_k, and all delays from lower-indexed processes to higher indexed are $d-u$, while those in the other direction are d, except that after a certain point, the earliest time information about p_0's *Dequeue* invocation could travel through all $p_1, .., p_{k-2}$, messages from p_{k-2} to p_{k-1} have delay d instead. We then define S'_k from S_{k-1} by having p_{k-1} additionally invoke *Dequeue* u before p_{k-2}'s *Dequeue* instance returns, and delaying messages from p_{k-2} to p_{k-1} similarly to those from p_{k-3} to p_{k-2}. Thus, S'_k is similar to S_k, except that messages from p_{k-3} to p_{k-2} and from p_{k-2} to p_{k-1} are both eventually delayed. We show that, despite this difference, S'_k and S_k are indistinguishable until after all *Dequeue* instances return, so they return the same values in both. The base case, $k = 3$, is the core of our overall proof, using the extended shifting technique from [14] to prove that the three *Dequeue* instances return only two distinct values.

Lemma 3.3. *In S'_n and S_n, all Dequeue instances return values from the set $\{1, \dots, n-1\}$, for sufficiently large n.*

Now, we want to show that D_n and S_n are indistinguishable, which leads to a contradiction, as processes must return the same values in indistinguishable runs. D_n and S_n are nearly identical: They have the same initial sequence of *Enqueue* instances, the same *Dequeue* invocations, and nearly identical message delays. The only difference is that past a certain point in time, messages in S_n from p_{n-2} to p_{n-1} have delay d, while such messages in D_n have delay $d-u$. We show that delaying those messages in D_n, for sufficiently large n, gives a run indistinguishable from S_n, a contradiction.

Theorem 3.4. *There is no uniform, set-linearizable implementation of a multiplicity queue with $|Dequeue| < \min\left\{\frac{d}{2}+u, \frac{3d+2u}{5}\right\}$.*

Finally, we note that our result is an improvement over the previously best-known bound from [10], with the added restriction to uniform algorithms. This follows because $\frac{d+u}{2} = \frac{d}{2} + \frac{u}{2} < \frac{d}{2} + u$ and $\frac{d+u}{2} \le \frac{2.5d+2.5u}{5} \le \frac{3d+2u}{5}$, since $u \le d$.

4 TIGHTNESS: EDGE CASE UPPER BOUND

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 developed 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 that all processes maintain a local copy of the queue, which they update based on messages about operation invocations. Each invoking process broadcasts operations and arguments on invocation, 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 when $u = 0$. If instances are non-concurrent, then there is more than d time from the first instance's invocation to the second's return, so by that return, the second process must know about any preceding instances. Each process will execute every *Enqueue* instance on its local copy, An *Enqueue* instance's invoking process will execute it after $d/2$ time, and all other processes after d time, when they receive the message. Non-invoking processes only locally execute *Dequeue* instances if they have not already seen a concurrent *Dequeue*, which is where multiplicity benefits us. If they have seen a concurrent *Dequeue*, detected by timestamps, they have already removed the return value, so there is no further work to do.

Theorem 4.1. *If $u = 0$, there is an optimal uniform, set-linearizable implementation of a multiplicity queue with $|Dequeue| = d/2$.*

REFERENCES

[1] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (1995), 124–142. https://doi.org/10.1145/200836.200869

[2] Armando Castañeda and Miguel Piña. 2021. Fully Read/Write Fence-Free Work-Stealing with Multiplicity. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference) (LIPIcs, Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 16:1–16:20. https://doi.org/10.4230/LIPIcs.DISC.2021.16

[3] Armando Castañeda, Sergio Rajsbaum, and Michel Raynal. 2020. Relaxed Queues and Stacks from Read/Write Operations. In *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference) (LIPIcs, Vol. 184)*, Quentin Bramas, Rotem Oshman, and Paolo Romano (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:19. https://doi.org/10.4230/LIPIcs.OPODIS.2020.13

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

[5] Colette Johnen, Adnane Khattabi, and Alessia Milani. 2022. Efficient Wait-Free Queue Algorithms with Multiple Enqueuers and Multiple Dequeuers. In *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium (LIPIcs, Vol. 253)*, Eshcar Hillel, Roberto Palmieri, and Etienne Rivière (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:19. https://doi.org/10.4230/LIPIcs.OPODIS.2022.4

[6] Martha J. Kosa. 1999. Time Bounds for Strong and Hybrid Consistency for Arbitrary Abstract Data Types. *Chic. J. Theor. Comput. Sci.* 1999 (1999). http://cjtcs.cs.uchicago.edu/articles/1999/9/contents.html

[7] Jennifer Lundelius and Nancy A. Lynch. 1984. An Upper and Lower Bound for Clock Synchronization. *Information and Control* 62, 2/3 (1984), 190–204.

[8] Gil Neiger. 1994. Set-Linearizability. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, California, USA, August 14-17, 1994*, James H. Anderson, David Peleg, and Elizabeth Borowsky (Eds.). ACM, 396. https://doi.org/10.1145/197917.198176

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

[10] Edward Talmage. 2022. Lower Bounds on Message Passing Implementations of Multiplicity-Relaxed Queues and Stacks. In *Structural Information and Communication Complexity - 29th International Colloquium, SIROCCO 2022, Paderborn, Germany, June 27-29, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13298)*, Merav Parter (Ed.). Springer, 253–264. https://doi.org/10.1007/978-3-031-09993-9_14

[11] Edward Talmage and Jennifer L. Welch. 2014. Improving Average Performance by Relaxing Distributed Data Structures. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8784)*, Fabian Kuhn (Ed.). Springer, 421–438. https://doi.org/10.1007/978-3-662-45174-8_29

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

[13] Anh Tran and Edward Talmage. 2023. Improved and Partially-Tight Lower Bounds for Message-Passing Implementations of Multiplicity Queues. https://doi.org/10.48550/arXiv.2305.11286 arXiv:2305.11286 [cs.DC]

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