

# Slot-queue - An optimized wait-free distributed MPSC

## 1. Motivation

A good example of a wait-free MPSC has been presented in [1]. In this paper, the authors propose a novel tree-structure and a min-timestamp scheme that allow both enqueue and dequeue to be wait-free and always complete in  $\Theta(\log n)$  where  $n$  is the number of enqueueers.

We have tried to port this algorithm to distributed context using MPI. The most problematic issue was that the original algorithm uses load-link/store-conditional (LL/SC). To adapt to MPI, we have to propose some modification to the original algorithm to make it use only compare-and-swap (CAS). Even though the resulting algorithm pretty much preserve the original algorithm's characteristic, that is wait-freedom and time complexity of  $\Theta(\log n)$ , we have to be aware that this is  $\Theta(\log n)$  remote operations, which is very expensive. We have estimated that for an enqueue or a dequeue operation in our initial LTQueue version, there are about  $2 * \log n$  to  $10 * \log n$  remote operations, depending on data placements and the current state of the LTQueue.

Therefore, to be more suitable for distributed context, we propose a new algorithm that's inspired by LTQueue, in which both enqueue and dequeue only perform a constant number of remote operations, at the cost of dequeue having to perform  $\Theta(n)$  local operations, where  $n$  is the number of enqueueers. Because remote operations are much more expensive, this might be a worthy tradeoff.

## 2. Structure

Each enqueueer will have a local SPSC as in LTQueue [1] that supports dequeue, enqueue and readFront. There's a global array whose entries store the minimum timestamp of the corresponding enqueueer's local SPSC.



Figure 1: Basic structure of slot queue

## 3. Pseudocode

### 3.1. SPSC

The SPSC of [1] is kept in tact, except that we change it into a circular buffer implementation.

#### Types

```

data_t = The type of data stored
spsc_t = The type of the local SPSC
    record
        First: int
        Last: int
        Capacity: int
        Data: an array of data_t of capacity
              Capacity
    end

```

#### Shared variables

```

First: index of the first undequeued entry
Last: index of the first unenqueued entry

```

#### Initialization

```

First = Last = 0
Set Capacity and allocate array.

```

The procedures are given as follows.

---

**Procedure 1:** `spsc_enqueue(v: data_t)` **returns** `bool`

---

```

1 if (Last + 1 == First)
2   | return false
3 Data[Last] = v
4 Last = (Last + 1) % Capacity
5 return true

```

---



---

**Procedure 2:** `spsc_dequeue()` **returns** `data_t`

---

```

6 if (First == Last) return  $\perp$ 
7 res = Data[First]
8 First = (First + 1) % Capacity
9 return res

```

---



---

**Procedure 3:** `spsc_readFront` **returns** `data_t`

---

```

10 if (First == Last)
11   | return  $\perp$ 
12 return Data[First]

```

---

### 3.2. Slot-queue

The slot-queue types and structures are given as follows:

#### Types

data\_t = The type of data stored  
 timestamp\_t = uint64\_t  
 spsc\_t = The type of the local SPSC

#### Shared variables

slots: An array of timestamp\_t with the number of entries equal the number of enqueueers  
 spscs: An array of spsc\_t with the number of entries equal the number of enqueueers  
 counter: uint64\_t

#### Initialization

| Initialize all local SPSCs.

| Initialize slots entries to MAX.

The enqueue operations are given as follows:

---

**Procedure 4:** `enqueue(rank: int, v: data_t)` **returns** `bool`

---

```

1 timestamp = FAA(counter)
2 value = (v, timestamp)
3 res = spsc_enqueue(spsc_s[rank], value)
4 if (!res) return false
5 if (!refreshEnqueue(rank, timestamp))
6   | refreshEnqueue(rank, timestamp)
7 return res

```

---



---

**Procedure 5:** `refreshEnqueue(rank: int, ts: timestamp_t)` **returns** `bool`

---

```

8 old-timestamp = slots[rank]
9 front = spsc_readFront(spsc_s[rank])
10 new-timestamp = front ==  $\perp$  ? MAX : front.timestamp
11 if (new-timestamp != ts)
12   | return true
13 return CAS(&slots[rank], old-timestamp, new-timestamp)

```

---

The dequeue operations are given as follows:

---

**Procedure 6:** `dequeue()` **returns** `data_t`

---

```

14 rank = readMinimumRank()
15 if (rank == DUMMY || slots[rank] == MAX)
16   | return  $\perp$ 
17 res = spsc_dequeue(spsc_s[rank])
18 if (res ==  $\perp$ ) return  $\perp$ 
19 if (!refreshDequeue(rank))
20   | refreshDequeue(rank)
21 return res

```

---

**Procedure 7: readMinimumRank() returns int**


---

```

22 rank = length(slots)
23 min-timestamp = MAX
24 for index in 0..length(slots)
25   timestamp = slots[index]
26   if (min-timestamp < timestamp)
27     rank = index
28     min-timestamp = timestamp
29 old-rank = rank
30 for index in 0..old-rank
31   timestamp = slots[index]
32   if (min-timestamp < timestamp)
33     rank = index
34     min-timestamp = timestamp
35 return rank == length(slots) ? DUMMY :
   rank

```

---

**Procedure 8: refreshDequeue(rank: int) returns bool**


---

```

36 old-timestamp = slots[rank]
37 front = spsc_readFront(spsc[rank])
38 new-timestamp = front ==  $\perp$  ? MAX :
   front.timestamp
39 return CAS(&slots[rank], old-timestamp,
   new-timestamp)

```

---

## 4. Linearizability of the local SPSC

In this section, we prove that the local SPSC is linearizable.

**Lemma 4.1** (*Linearizability of spsc\_enqueue*) The linearization point of spsc\_enqueue is right after line 2 or right after line 4.

**Proof** Notice that only spsc\_enqueue can modify the Last shared variable and only spsc\_dequeue can modify the First shared variable.

If line 2 is executed, that means  $\text{Last} + 1 == \text{First}$  and the enqueue is deemed as failed. This

state can only be exited when an spsc\_dequeue executes line 8. If line 2 is executed before line 8 of spsc\_dequeue then it's safe that we linearize spsc\_enqueue before spsc\_dequeue. Therefore, line 2 is a linearization point of spsc\_enqueue.

Suppose line 2 is not executed.

If line 4 hasn't been executed, the SPSC is as if no element has been enqueued. On the other hand, if line 4 is executed, the enqueue is sure to have completed and other spsc\_dequeues or spsc\_readFront can see this enqueue's effect. Therefore, line 4 is another linearization point of spsc\_enqueue.  $\square$

**Lemma 4.2** (*Linearizability of spsc\_dequeue*) The linearization point of spsc\_dequeue is right after line 6 or right after line 8.

**Proof** Notice that only spsc\_enqueue can modify the Tail shared variable and only spsc\_dequeue can modify the Head shared variable.

If line 6 is executed, that means  $\text{Last} == \text{First}$  and the dequeue is deemed as failed. This state can only be exited when an spsc\_enqueue executes line 4. If line 6 is executed before line 4 of spsc\_dequeue then it's safe that we linearize spsc\_dequeue before spsc\_enqueue. Therefore, line 6 is a linearization point of spsc\_enqueue.

Suppose line 6 is not executed.

If line 8 hasn't been executed, the SPSC is as if no element has been dequeued. On the other hand, if line 8 is executed, the dequeue is sure to have completed and other spsc\_enqueue or spsc\_readFront can see this enqueue's effect. Therefore, line 8 is another linearization point of spsc\_enqueue.  $\square$

**Lemma 4.3** (*Linearizability of spsc\_readFront*) The linearization point spsc\_readFront is right after line 11 or right after line 12.

**Proof** If line 11 is executed, that means spsc\_readFront has just observed and effect of line 8 of spsc\_dequeue or has not observed line 4 of spsc\_enqueue. In this case, spsc\_readFront

is linearized after this `spsc_dequeue` or before `spsc_enqueue`.

If line 12 is executed, that means `spsc_readFront` has just observed and effect of line 4 of `spsc_enqueue` or has not observed line 8 of `spsc_dequeue`. In this case, `spsc_readFront` is linearized after this `spsc_enqueue` or before `spsc_dequeue`.  $\square$

**Theorem 4.4** (*Linearizability of local SPSC*) The local SPSC is linearizable.

**Proof** This directly follows from Lemma 4.1, Lemma 4.2, Lemma 4.3.  $\square$

## 5. ABA problem

Noticeably, we use no scheme to avoid ABA problem in Slot-queue. In actuality, ABA problem does not adversely affect our algorithm's correctness, except in the extreme case that the 64-bit global counter overflows, which is unlikely.

### 5.1. ABA-safety

Not every ABA problem is unsafe. We formalize in this section which ABA problem is safe and which is not.

**Definition 5.1.1** A **modification instruction** on a variable  $v$  is an atomic instruction that may change the value of  $v$  e.g. a store or a CAS.

**Definition 5.1.2** A **successful modification instruction** on a variable  $v$  is an atomic instruction that changes the value of  $v$  e.g. a store or a successful CAS.

**Definition 5.1.3** A **CAS-sequence** on a variable  $v$  is a sequence of instructions of a method  $m$  such that:

- The first instruction is a load  $v_0 = \text{load}(v)$ .
- The last instruction is a CAS( $\&v, v_0, v_1$ ).
- There's no modification instruction on  $v$  between the first and the last instruction.

**Definition 5.1.4** A **successful CAS-sequence** on a variable  $v$  is a **CAS-sequence** on  $v$  that ends with a successful CAS.

**Definition 5.1.5** Consider a method  $m$  on a concurrent object  $S$ .  $m$  is said to be **ABA-safe** if and only if for any history of method calls produced from  $S$ , we can reorder any successful CAS-sequences by an invocation of  $m$  in the following fashion:

- If a successful CAS-sequence is part of an invocation of  $m$ , after reordering, it must still be part of that invocation.
- If a successful CAS-sequence by an invocation of  $m$  precedes another in that same invocation, after reordering, this ordering is still respected.
- Any successful CAS-sequence by an invocation of  $m$  after reordering must not overlap with a successful modification instruction on the same variable.
- After reordering, all method calls' response events on the concurrent object  $S$  stay the same.

### 5.2. Proof of ABA-safety

Notice that we only use CAS on:

- Line 13 of `refreshEnqueue` (Procedure 5), or an enqueue in general (Procedure 4).
- Line 42 of `refreshDequeue` (Procedure 8) or a dequeue in general (Procedure 6).

Both CAS target some slot in the slots array.

We apply some domain knowledge of our algorithm to the above formalism.

**Definition 5.2.1** A **CAS-sequence** on a slot  $s$  of an enqueue that corresponds to  $s$  is the sequence of instructions from line 8 to line 13 of its `refreshEnqueue`.

**Definition 5.2.2** A **slot-modification instruction** on a slot  $s$  of an enqueue that corresponds to  $s$  is line 13 of `refreshEnqueue`.

**Definition 5.2.3** A **CAS-sequence** on a slot  $s$  of a dequeue that corresponds to  $s$  is the sequence of instructions from line 36 to line 42 of its `refreshDequeue`.

**Definition 5.2.4** A **slot-modification instruction** on a slot  $s$  of a dequeue that corresponds to  $s$  is line 40 or line 42 of `refreshDequeue`.

**Definition 5.2.5** A **CAS-sequence** of a dequeue/enqueue is said to **observes a slot value of**  $s_0$  if it loads  $s_0$  at line 8 of refreshEnqueue or line 36 of refreshDequeue.

We can now turn to our interested problem in this section.

**Lemma 5.2.1** (*Concurrent accesses on a local SPSC and a slot*) Only one dequeuer and one enqueue can concurrently modify a local SPSC and a slot in the slots array.

**Proof** This is trivial to prove based on the algorithm's definition.  $\square$

**Lemma 5.2.2** (*Monotonicity of local SPSC timestamps*) Each local SPSC in Slot-queue contains elements with increasing timestamps.

**Proof** Each enqueue would FAA the global counter (line 1 in Procedure 4) and enqueue into the local SPSC an item with the timestamp obtained from the counter. Applying Lemma 5.2.1, we know that items are enqueued one at a time into the SPSC. Therefore, later items are enqueued by later enqueues, which obtain increasing values by FAA-ing the shared counter. The theorem holds.  $\square$

**Lemma 5.2.3** A refreshEnqueue (Procedure 5) can only changes a slot to a value other than MAX.

**Proof** For refreshEnqueue to change the slot's value, the condition on line 11 must be false. Then new-timestamp must equal to  $t_s$ , which is not MAX. It's obvious that the CAS on line 13 changes the slot to a value other than MAX.  $\square$

**Theorem 5.2.4** (*ABA safety of dequeue*) Assume that the 64-bit global counter never overflows, dequeue (Procedure 6) is ABA-safe.

**Proof** Consider a **successful CAS-sequence** on slot  $s$  by a dequeue  $d$ .

Denote  $t_d$  as the value this CAS-sequence observes.

Due to Lemma 5.2.1, there can only be at most one enqueue at one point in time within  $d$ .

If there's no **successful slot-modification instruction** on slot  $s$  by an enqueue  $e$  within  $d$ 's **successful CAS-sequence**, then this dequeue is ABA-safe.

Suppose the enqueue  $e$  executes the **last successful slot-modification instruction** on slot  $s$  within  $d$ 's **successful CAS-sequence**. Denote  $t_e$  to be the value that  $e$  sets  $s$ .

If  $t_e \neq t_d$ , this CAS-sequence of  $d$  cannot be successful, which is a contradiction.

Therefore,  $t_e = t_d$ .

Note that  $e$  can only set  $s$  to the timestamp of the item it enqueues. That means,  $e$  must have enqueued a value with timestamp  $t_d$ . However, by definition,  $t_d$  is read before  $e$  executes the CAS. This means another process (dequeuer/enqueueer) has seen the value  $e$  enqueued and CAS  $s$  for  $e$  before  $t_d$ . By Lemma 5.2.1, this "another process" must be another dequeuer  $d'$  that precedes  $d$  because it overlaps with  $e$ .

Because  $d'$  and  $d$  cannot overlap, while  $e$  overlaps with both  $d'$  and  $d$ ,  $e$  must be the *first* enqueue on  $s$  that overlaps with  $d$ . Combining with Lemma 5.2.1 and the fact that  $e$  executes the **last successful slot-modification instruction** on slot  $s$  within  $d$ 's **successful CAS-sequence**,  $e$  must be the only enqueue that executes a **successful slot-modification instruction** on  $s$  within  $d$ 's **successful CAS-sequence**.

During the start of  $d$ 's successful CAS-sequence till the end of  $e$ , `spsc_readFront` on the local SPSC must return the same element, because:

- There's no other dequeues running during this time.
- There's no enqueue other than  $e$  running.
- The `spsc_enqueue` of  $e$  must have completed before the start of  $d$ 's successful CAS sequence, because a previous dequeuer  $d'$  can see its effect.

Therefore, if we were to move the starting time of  $d$ 's successful CAS-sequence right after  $e$  has

ended, we still retain the output of the program because:

- The CAS sequence only reads two shared values: `slots[rank]` and `spsc_readFront()`, but we have proven that these two values remain the same if we were to move the starting time of  $d$ 's successful CAS-sequence this way.
- The CAS sequence does not modify any values except for the last CAS instruction, and the ending time of the CAS sequence is still the same.
- The CAS sequence modifies `slots[rank]` at the CAS but the target value is the same because inputs and shared values are the same in both cases.

We have proved that if we move  $d$ 's successful CAS-sequence to start after the *last successful slot-modification instruction* on slot  $s$  within  $d$ 's **successful CAS-sequence**, we still retain the program's output.

If we apply the reordering for every dequeue, the theorem directly follows.  $\square$

**Theorem 5.2.5 (ABA safety of enqueue)** Assume that the 64-bit global counter never overflows, enqueue (Procedure 4) is ABA-safe.

**Proof** Consider a **successful CAS-sequence** on slot  $s$  by an enqueue  $e$ .

Denote  $t_e$  as the value this CAS-sequence observes.

Due to Lemma 5.2.1, there can only be at most one enqueue at one point in time within  $e$ .

If there's no **successful slot-modification instruction** on slot  $s$  by a dequeue  $d$  within  $e$ 's **successful CAS-sequence**, then this enqueue is ABA-safe.

Suppose the dequeue  $d$  executes the *last successful slot-modification instruction* on slot  $s$  within  $e$ 's **successful CAS-sequence**. Denote  $t_d$  to be the value that  $d$  sets  $s$ .

If  $t_d \neq t_e$ , this CAS-sequence of  $e$  cannot be successful, which is a contradiction.

Therefore,  $t_d = t_e$ .

If  $t_d = t_e = \text{MAX}$ , this means  $e$  observes a value of MAX before  $d$  even sets  $s$  to MAX. If this MAX value is the initialized value of  $s$ , it's a contradiction, as  $s$  must be non-MAX at some point for a dequeue such as  $d$  to run. If this MAX value is set by an enqueue, it's also a contradiction, as `refreshEnqueue` cannot set a slot to MAX. Therefore, this MAX value is set by a dequeue  $d'$ . If  $d' \neq d$  then it's a contradiction, because between  $d'$  and  $d$ ,  $s$  must be set to be a non-MAX value before  $d$  can be run. Therefore,  $d' = d$ . But, this means  $e$  observes a value set by  $d$ , which violates our assumption.

Therefore  $t_d = t_e = t' \neq \text{MAX}$ .  $e$  cannot observe the value  $t'$  set by  $d$  due to our assumption. Suppose  $e$  observes the value  $t'$  from  $s$  set by another enqueue/dequeue call other than  $d$ .

If this "another call" is a dequeue  $d'$  other than  $d$ ,  $d'$  precedes  $d$ . By Lemma 5.2.2, after each dequeue, the front element's timestamp will be increasing, therefore,  $d'$  must have set  $s$  to a timestamp smaller than  $t_d$ . However,  $e$  observes  $t_e = t_d$ . This is a contradiction.

Therefore, this "another call" is an enqueue  $e'$  other than  $e$  and  $e'$  precedes  $e$ . We know that an enqueue only sets  $s$  to the timestamp it obtains.

Suppose  $e'$  does not overlap with  $d$ .  $e'$  can only set  $s$  to  $t'$  if  $e'$  sees that the local SPSC has the front element as the element it enqueues. Due to Lemma 5.2.1, this means  $e'$  must observe a local SPSC with only the element it enqueues. Then, when  $d$  executes `readFront`, the item  $e'$  enqueues must have been dequeued out already, thus,  $d$  cannot set  $s$  to  $t'$ . This is a contradiction.

Therefore,  $e'$  overlaps with  $d$ .

For  $e'$  to set  $s$  to the same value as  $d$ ,  $e'$ 's `spsc_readFront` must serialize after  $d$ 's `spsc_dequeue`.

Because  $e'$  and  $e$  cannot overlap, while  $d$  overlaps with both  $e'$  and  $e$ ,  $d$  must be the *first* dequeue on  $s$  that overlaps with  $e$ . Combining with Lemma 5.2.1

and the fact that  $d$  executes the *last successful slot-modification instruction* on slot  $s$  within  $e$ 's *successful CAS-sequence*,  $d$  must be the only dequeue that executes a *successful slot-modification instruction* within  $e$ 's *successful CAS-sequence*.

During the start of  $e$ 's successful CAS-sequence till the end of  $d$ , `spsc_readFront` on the local SPSC must return the same element, because:

- There's no other enqueues running during this time.
- There's no dequeue other than  $d$  running.
- The `spsc_dequeue` of  $d$  must have completed before the start of  $e$ 's successful CAS sequence, because a previous enqueue  $e'$  can see its effect.

Therefore, if we were to move the starting time of  $e$ 's successful CAS-sequence right after  $d$  has ended, we still retain the output of the program because:

- The CAS sequence only reads two shared values: `slots[rank]` and `spsc_readFront()`, but we have proven that these two values remain the same if we were to move the starting time of  $e$ 's successful CAS-sequence this way.
- The CAS sequence does not modify any values except for the last CAS/store instruction, and the ending time of the CAS sequence is still the same.
- The CAS sequence modifies `slots[rank]` at the CAS but the target value is the same because inputs and shared values are the same in both cases.

We have proved that if we move  $e$ 's successful CAS-sequence to start after the *last successful slot-modification instruction* on slot  $s$  within  $e$ 's *successful CAS-sequence*, we still retain the program's output.

If we apply the reordering for every enqueue, the theorem directly follows.  $\square$

**Theorem 5.2.6 (ABA safety)** Assume that the 64-bit global counter never overflows, Slot-queue is ABA-safe.

**Proof** This follows from Theorem 5.2.5 and Theorem 5.2.4.  $\square$

## 6. Linearizability of Slot-queue

**Definition 6.1** For an enqueue or dequeue  $op$ ,  $rank(op)$  is the rank of the enqueueer whose local SPSC is affected by  $op$ .

**Definition 6.2** For an enqueueer whose rank is  $r$ , the value stored in its corresponding slot at time  $t$  is denoted as  $slot(r, t)$ .

**Definition 6.3** For an enqueueer with rank  $r$ , the minimum timestamp among the elements between First and Last in its local SPSC at time  $t$  is denoted as  $min\_spsc\_ts(r, t)$ .

**Definition 6.4** For an enqueue, **slot-refresh phase** refer to its execution of line 5-6 of Procedure 4.

**Definition 6.5** For a dequeue, **slot-refresh phase** refer to its execution of line 19-20 of Procedure 6.

**Definition 6.6** For a dequeue, **slot-scan phase** refer to its execution of line 24-34 of Procedure 7.

**Definition 6.7** An enqueue operation  $e$  is said to **match** a dequeue operation  $d$  if  $d$  returns a timestamp that  $e$  enqueues. Similarly,  $d$  is said to **match**  $e$ . In this case, both  $e$  and  $d$  are said to be **matched**.

**Definition 6.8** An enqueue operation  $e$  is said to be **unmatched** if no dequeue operation **matches** it.

**Definition 6.9** A dequeue operation  $d$  is said to be **unmatched** if no enqueue operation **matches** it, in other word,  $d$  returns  $\perp$ .

We prove some algorithm-specific results first, which will form the basis for the more fundamental results.

**Lemma 6.1** If an enqueue  $e$  begins its **slot-refresh phase** at time  $t_0$  and finishes at time  $t_1$ , there's always at least one successful

refreshEnqueue or refreshDequeue on  $rank(e)$  starting and ending its **CAS-sequence** between  $t_0$  and  $t_1$ .

**Proof** If one of the two refreshEnqueues succeeds, then the lemma obviously holds.

Consider the case where both fail.

The first refreshEnqueue fails because there's another refreshDequeue executing its **slot-modification instruction** successfully after  $t_0$  but before the end of the first refreshEnqueue's **CAS-sequence**.

The second refreshEnqueue fails because there's another refreshDequeue executing its **slot-modification instruction** successfully after  $t_0$  but before the end of the second refreshEnqueue's **CAS-sequence**. This another refreshDequeue must start its **CAS-sequence** after the end of the first successful refreshDequeue, due to Lemma 5.2.1. In other words, this another refreshDequeue starts and successfully ends its **CAS-sequence** between  $t_0$  and  $t_1$ .

We have proved the theorem.  $\square$

**Lemma 6.2** If a dequeue  $d$  begins its **slot-refresh phase** at time  $t_0$  and finishes at time  $t_1$ , there's always at least one successful refreshEnqueue or refreshDequeue on  $rank(d)$  starting and ending its **CAS-sequence** between  $t_0$  and  $t_1$ .

**Proof** This is similar to the above lemma.  $\square$

**Lemma 6.3** Given a rank  $r$ , if an enqueue  $e$  on  $r$  that obtains the timestamp  $c$  completes at  $t_0$  and is still unmatched by  $t_1$ , then  $slot(r, t) \leq c$  for any  $t \in [t_0, t_1]$ .

**Proof** Take  $t'$  to be the time  $e$ 's `spsc_enqueue` takes effect.

By Lemma 6.1, there must be a successful refresh call that observes the effect of `spsc_enqueue` happening at  $t''$ ,  $t'' \in [t', t_0]$ .

By the same reasoning as in Theorem 5.2.6, any successful slot-modification instructions happening after  $t''$  must observe the effect of

`spsc_enqueue`. However, because  $e$  is never matched between  $t''$  and  $t_1$ , the timestamp  $c$  is in the local SPSC the whole timespan  $[t'', t_1]$ . Therefore, any slot-modification instructions during  $[t'', t_1]$  must set the slot's value to some value not greater than  $c$ .  $\square$

We now look at the more fundamental results.

**Theorem 6.4** If an enqueue  $e$  precedes another dequeue  $d$ , then either:

- $d$  isn't matched.
- $d$  matches  $e$ .
- $e$  matches  $d'$  and  $d'$  precedes  $d$ .
- $d$  matches  $e'$  and  $e'$  precedes  $e$ .
- $d$  matches  $e'$  and  $e'$  overlaps with  $e$ .

**Proof** If  $d$  doesn't match anything, the theorem holds. If  $d$  matches  $e$ , the theorem also holds. Suppose  $d$  matches  $e'$ ,  $e' \neq e$ .

If  $e$  matches  $d'$  and  $d'$  precedes  $d$ , the theorem also holds. Suppose  $e$  matches  $d'$  such that  $d$  precedes  $d'$  or is unmatched. (1)

Suppose  $e$  obtains a timestamp of  $c$  and  $e'$  obtains a timestamp of  $c'$ .

Due to (1), at the time  $d$  starts,  $e$  has finished but it is still unmatched. By the way Procedure 7 is defined and by Lemma 6.3,  $d$  would find a slot that stores a timestamp that is not greater than the one  $e$  enqueues. In other word,  $c' \leq c$ . But  $c' \neq c$ , then  $c' < c$ . Therefore,  $e$  cannot precede  $e'$ , otherwise,  $c < c'$ .

So, either  $e'$  precedes or overlaps with  $e$ . The theorem holds.  $\square$

**Lemma 6.5** If  $d$  matches  $e$ , then either  $e$  precedes or overlaps with  $d$ .

**Proof** If  $d$  precedes  $e$ , none of the local SPSCs can contain an item with the timestamp of  $e$ . Therefore,  $d$  cannot return an item with a timestamp of  $e$ . Thus  $d$  cannot match  $e$ .

Therefore,  $e$  either precedes or overlaps with  $d$ .  $\square$

**Theorem 6.6** If a dequeue  $d$  precedes another enqueue  $e$ , then either:



- $d$  isn't matched.
- $d$  matches  $e'$  such that  $e'$  precedes or overlaps with  $e$  and  $e' \neq e$ .

**Proof** If  $d$  isn't matched, the theorem holds.

Suppose  $d$  matches  $e'$ . By Lemma 6.5, either  $e'$  precedes or overlaps with  $d$ . Therefore,  $e' \neq e$ . Furthermore,  $e$  cannot precede  $e'$ , because then  $d$  would precede  $e'$ .

We have proved the theorem.  $\square$

**Theorem 6.7** If an enqueue  $e_0$  precedes another enqueue  $e_1$ , then either:

- Both  $e_0$  and  $e_1$  aren't matched.
- $e_0$  is matched but  $e_1$  is not matched.
- $e_0$  matches  $d_0$  and  $e_1$  matches  $d_1$  such that  $d_0$  precedes  $d_1$ .

**Proof** if  $e_1$  is not matched, the theorem holds.

Suppose  $e_1$  matches  $d_1$ . By Lemma 6.5, either  $e_1$  precedes or overlaps with  $d_1$ .

Suppose the contrary,  $e_0$  is unmatched or  $e_0$  matches  $d_0$  such that  $d_1$  precedes  $d_0$ , then when  $d_1$  starts,  $e_0$  is still unmatched.

If  $e_0$  and  $e_1$  targets the same rank, it's obvious that  $d_1$  must prioritize  $e_0$  over  $e_1$ . Thus  $d_1$  cannot match  $e_1$ .

If  $e_0$  targets a later rank than  $e_1$ ,  $d_1$  cannot find  $e_1$  in the first scan, because the scan is left-to-right, and if it finds  $e_1$  it would later find  $e_0$  that has a lower timestamp. Suppose  $d_1$  finds  $e_1$  in the second scan, that means  $d_1$  finds  $e' \neq e_1$  and  $e'$ 's timestamp is larger than  $e_1$ 's, which is larger than  $e_0$ 's. Due to the scan being left-to-right,  $e'$  must target a later rank than  $e_1$ . If  $e'$  also targets a later rank than  $e_0$ , then in the second scan,  $d_1$  would have prioritized  $e_0$  that has a lower timestamp. Suppose  $e'$  targets an earlier rank than  $e_0$  but later than  $e_1$ . Because  $e_0$ 's timestamp is larger than  $e'$ 's, it must precede or overlap with  $e$ . Similarly,  $e_1$  must precede or overlap with  $e$ . Because  $e'$  targets an earlier rank than  $e_0$ ,  $e_0$ 's **slot-refresh phase** must finish after  $e'$ 's. That means  $e_1$  must start

after  $e'$ 's **slot-refresh phase**, because  $e_0$  precedes  $e_1$ . But then,  $e_1$  must obtain a timestamp larger than  $e'$ , which is a contradiction.

Suppose  $e_0$  targets an earlier rank than  $e_1$ . If  $d_1$  finds  $e_1$  in the first scan, then in the second scan,  $d_1$  would have prioritize  $e_0$ 's timestamp. Suppose  $d_1$  finds  $e_1$  in the second scan and during the first scan, it finds  $e' \neq e_1$  and  $e'$ 's timestamp is larger than  $e_1$ 's, which is larger than  $e_0$ 's. Due to how the second scan is defined,  $e'$  targets a later rank than  $e_1$ , which targets a later rank than  $e_0$ . Because during the second scan,  $e_0$  is not chosen, its **slot-refresh phase** must finish after  $e'$ 's. Because  $e_0$  precedes  $e_1$ ,  $e_1$  must start after  $e'$ 's **slot-refresh phase**, so it must obtain a larger timestamp than  $e'$ , which is a contradiction.

Therefore, by contradiction,  $e_0$  must be matched and  $e_0$  matches  $d_0$  such that  $d_0$  precedes  $d_1$ .  $\square$

**Theorem 6.8** If a dequeue  $d_0$  precedes another dequeue  $d_1$ , then either:

- $d_0$  isn't matched.
- $d_1$  isn't matched.
- $d_0$  matches  $e_0$  and  $d_1$  matches  $e_1$  such that  $e_0$  precedes or overlaps with  $e_1$ .

**Proof** If either  $d_0$  isn't matched or  $d_1$  isn't matched, the theorem holds.

Suppose  $d_0$  matches  $e_0$  and  $d_1$  matches  $e_1$ .

If  $e_1$  precedes  $e_0$ , applying Theorem 6.7, we have  $e_1$  matches  $d_1$  and  $e_0$  matches  $d_0$  such that  $d_1$  precedes  $d_0$ . This is a contradiction.

Therefore,  $e_0$  either precedes or overlaps with  $e_1$ .  $\square$

**Theorem 6.9** (*Linearizability of Slot-queue*) Slot-queue is linearizable.

**Proof** Suppose some history  $H$  produced from the Slot-queue algorithm.

If  $H$  contains some pending method calls, we can just wait for them to complete (because the algorithm is wait-free, which we will prove later). Therefore, now we consider all  $H$  to contain only

completed method calls. So, we know that if a dequeue or an enqueue in  $H$  is matched or not.

If there are some unmatched enqueues, we can append dequeues sequentially to the end of  $H$  until there's no unmatched enqueues. Consider one such  $H'$ .

We already have a strict partial order  $\rightarrow_{H'}$  on  $H'$ .

Because the queue is MPSC, there's already a total order among the dequeues.

We will extend  $\rightarrow_{H'}$  to a strict total order  $\Rightarrow_{H'}$  on  $H'$  as follows:

- If  $X \rightarrow_{H'} Y$  then  $X \Rightarrow_{H'} Y$ . (1)
- If a dequeue  $d$  matches  $e$  then  $e \Rightarrow_{H'} d$ . (2)
- If a dequeue  $d_0$  matches  $e_0$  and another dequeue matches  $e_1$  such that  $d_0 \Rightarrow_{H'} d_1$  then  $e_0 \Rightarrow_{H'} e_1$ . (3)
- If a dequeue  $d$  overlaps with an enqueue  $e$  but does not match  $e$ ,  $d \Rightarrow_{H'} e$ . (4)

We will prove that  $\Rightarrow_{H'}$  is a strict total order on  $H'$ . That is, for every pair of different method calls  $X$  and  $Y$ , either exactly one of these is true  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$  and for any  $X$ ,  $X \not\Rightarrow_{H'} X$ .

It's obvious that  $X \not\Rightarrow_{H'} X$ .

If  $X$  and  $Y$  are dequeues, because there's a total order among the dequeues, either exactly one of these is true:  $X \rightarrow_{H'} Y$  or  $Y \rightarrow_{H'} X$ . Then due to (1), either  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$ . Notice that we cannot obtain  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$  from (2), (3), or (4).

Therefore, exactly one of  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$  is true. (\*)

If  $X$  is dequeue and  $Y$  is enqueue, in this case (3) cannot help us obtain either  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$ , so we can disregard it.

- If  $X \rightarrow_{H'} Y$ , then due to (1),  $X \Rightarrow_{H'} Y$ . By definition,  $X$  precedes  $Y$ , so (4) cannot apply. Applying Theorem 6.6, either
  - $X$  isn't matched, (2) cannot apply. Therefore,  $Y \not\Rightarrow_{H'} X$ .
  - $X$  matches  $e'$  and  $e' \neq Y$ . Therefore,  $X$  does not match  $Y$ , or (2) cannot apply. Therefore,  $Y \not\Rightarrow_{H'} X$ .

Therefore, in this case,  $X \Rightarrow_{H'} Y$  and  $Y \not\Rightarrow_{H'} X$ .

- If  $Y \rightarrow_{H'} X$ , then due to (1),  $Y \Rightarrow_{H'} X$ . By definition,  $Y$  precedes  $X$ , so (4) cannot apply. Even if (2) applies, it can only help us obtain  $Y \Rightarrow_{H'} X$ . Therefore, in this case,  $Y \Rightarrow_{H'} X$  and  $X \not\Rightarrow_{H'} Y$ .
- If  $X$  overlaps with  $Y$ :
  - If  $X$  matches  $Y$ , then due to (2),  $Y \Rightarrow_{H'} X$ . Because  $X$  matches  $Y$ , (4) cannot apply. Therefore, in this case  $Y \Rightarrow_{H'} X$  but  $X \not\Rightarrow_{H'} Y$ .
  - If  $X$  does not match  $Y$ , then due to (4),  $X \Rightarrow_{H'} Y$ . Because  $X$  doesn't match  $Y$ , (2) cannot apply. Therefore, in this case  $X \Rightarrow_{H'} Y$  but  $Y \not\Rightarrow_{H'} X$ .

Therefore, exactly one of  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$  is true. (\*\*)

If  $X$  is enqueue and  $Y$  is enqueue, in this case (2) and (4) are irrelevant:

- If  $X \rightarrow_{H'} Y$ , then due to (1),  $X \Rightarrow_{H'} Y$ . By definition,  $X$  precedes  $Y$ . Applying Theorem 6.7,
  - Both  $X$  and  $Y$  aren't matched, then (3) cannot apply. Therefore, in this case,  $Y \not\Rightarrow_{H'} X$ .
  - $X$  is matched but  $Y$  is not matched, then (3) cannot apply. Therefore, in this case,  $Y \not\Rightarrow_{H'} X$ .
  - $X$  matches  $d_x$  and  $Y$  matches  $d_y$  such that  $d_x$  precedes  $d_y$ , then (3) applies and we obtain  $X \Rightarrow_{H'} Y$ .

Therefore, in this case,  $X \Rightarrow_{H'} Y$  but  $Y \not\Rightarrow_{H'} X$ .

- If  $Y \rightarrow_{H'} X$ , this case is symmetric to the first case. We obtain  $Y \Rightarrow_{H'} X$  but  $X \not\Rightarrow_{H'} Y$ .
- If  $X$  overlaps with  $Y$ , because in  $H'$ , all enqueues are matched, then,  $X$  matches  $d_x$  and  $d_y$ . Because  $d_x$  either precedes or succeeds  $d_y$ , Applying (3), we obtain either  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$  and there's no way to obtain the other.

Therefore, exactly one of  $X \Rightarrow_{H'} Y$  or  $Y \Rightarrow_{H'} X$  is true. (\*\*\*)

From (\*), (\*\*), (\*\*\*), we have proved that  $\Rightarrow_{H'}$  is a strict total ordering that is consistent with  $\rightarrow_{H'}$ . In other words, we can order method calls in  $H'$  in a sequential manner. We will prove that this sequential order is consistent with FIFO semantics:

- An item can only be dequeued once: This is trivial as a dequeue can only match one enqueue.
- Items are dequeued in the order they are enqueued: Suppose there are two enqueues  $e_1, e_2$  such that  $e_1 \Rightarrow_{H'} e_2$  and suppose they match  $d_1$  and  $d_2$ . Then we have obtained  $e_1 \Rightarrow_{H'} e_2$  either because:
  - (3) applies, in this case  $d_1 \Rightarrow_{H'} d_2$  is a condition for it to apply.
  - (1) applies, then  $e_1$  precedes  $e_2$ , by Theorem 6.7,  $d_1$  must precede  $d_2$ , thus  $d_1 \Rightarrow_{H'} d_2$ .

Therefore, if  $e_1 \Rightarrow_{H'} e_2$  then  $d_1 \Rightarrow_{H'} d_2$ .

- An item can only be dequeued after it's enqueued: Suppose there is an enqueue  $e$  matched by  $d$ . By (2), obviously  $e \Rightarrow_{H'} d$ .
- If the queue is empty, dequeues return nothing. Suppose a dequeue  $d$  such that any  $e \Rightarrow_{H'} d$  is all matched by some  $d'$  and  $d' \Rightarrow_{H'} d$ , we will prove that  $d$  is unmatched. By Lemma 6.5,  $d$  can only match an enqueue  $e_0$  that precedes or overlaps with  $d$ .
  - If  $e_0$  precedes  $d$ , by our assumption, it's already matched by another dequeue.
  - If  $e_0$  overlaps with  $d$ , by our assumption,  $d \Rightarrow_{H'} e_0$  because if  $e_0 \Rightarrow_{H'} d$ ,  $e_0$  is already matched by another  $d'$ . Then, we can only obtain this because (4) applies, but then  $d$  does not match  $e_0$ .

Therefore,  $d$  is unmatched.

In conclusion,  $\Rightarrow_{H'}$  is a way we can order method calls in  $H'$  sequentially that conforms to FIFO semantics. Therefore, we can also order method calls in  $H$  sequentially that conforms to FIFO semantics as we only append dequeues sequentially to the end of  $H$  to obtain  $H'$ .

We have proved the theorem.  $\square$

## 7. Wait-freedom

The algorithm is trivially wait-free as there is no possibility of infinite loops.

## 8. Memory-safety

The algorithm is memory-safe: No memory deallocation happens and accesses are only made on allocated memory.

## References

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