# 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 enqueuers.

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 enqueuers. Because remote operations are much more expensive, this might be a worthy tradeoff.

#### 2. Structure

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

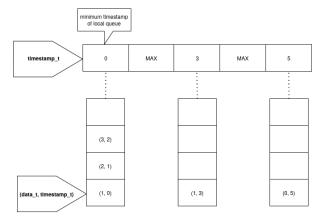


Figure 1: Basic structure of slot queue

### 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 enqueuers spscs: An array of spsc\_t with the number of entries equal the number of enqueuers 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 = FFA(counter)
- 2 value = (v, timestamp)
- 3 res = spsc\_enqueue(spscs[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(spscs[rank])
- new-timestamp = front ==  $\perp$  ? MAX : front.timestamp
- 11 if (new-timestamp != ts)
- 12 | return true
- 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 ⊥
- 17 res = spsc\_dequeue(spscs[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 \text{ min-timestamp} = MAX
24 for index in 0..length(slots)
     timestamp = slots[index]
25
     if (min-timestamp < timestamp)</pre>
26
       rank = index
27
      min-timestamp = timestamp
28
29 old-rank = rank
30 for index in 0..old-rank
     timestamp = slots[index]
31
     if (min-timestamp < timestamp)</pre>
32
       rank = index
33
      min-timestamp = timestamp
   return rank == length(slots) ? DUMMY :
   rank
```

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

```
36 old-timestamp = slots[rank]
37 front = spsc_readFront(spscs[rank])
38 new-timestamp = front == \( \triangle \)? MAX :
front.timestamp
39 if (front != \( \triangle \))
40 | slots[rank] = new-timestamp
41 | return true
42 return CAS(&slots[rank], old-timestamp,
new-timestamp)
```

# 4. Linearizability

# 5. Wait-freedom

# 6. Memory-safety

## References

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