Relaxation for Efficient Asynchronous Queues

Samuel Baldwin
Bucknell University, USA

Cole Hausman
Bucknell University, USA

Mohamed Bakr
Bucknell University, USA

Edward Talmage
Bucknell Univserity, USA

1 Introduction

This paper proposes a new algorithm for implementing a standard FIFO queue in a fully asynchronous message passing mode. To our knowledge we have not encountered any previous work that suggested a queue algorithm for this specific model. Our algorithm utilizes vector clock timestamps that allow individual processes to hold some view of the time steps of other processes have taken at points of communication. Using this technique to timestamp invocations of the two operations of the queue, then linearizing all of the queue invocations based on the ascending lexicographic order of these timestmps creates a valid permutation that meets the specifications of a FIFO Queue. The goal of establishing this queue method is to design a system with full replication. There are existing systems that handle the asynchronous queue (server client model), but are incapable of replication. With replication, and (INSERT THE CITATION TO THE PAPER HERE) we hope that this queue algorithm can be used in fault tolerant systems in the future.

1.1 Related Work

2 Model and Definitions

2.1 Asynchronous System Model

We assume a fully asynchronous message passing model, with a set of n processes $\Pi = [p_0, \ldots, p_{n-1}]$ modeled as state machines. These state machines are time-free, meaning that their output is only described through their input and state transitions without any specific time bound. All inter-process communication is assumed to be reliable i.e any message that is sent will always be received at it's destination process. Additionally, communication channels between processes are considered to be treated as FIFO, in that any message sent from process A to process B will be processed prior to any other message from the same pair sent later in an execution. This can be accomplished by assuming that each process has an incoming buffer for each other process, and that inter-process messages are marked numerically, and out of order messages are stored in the buffer until they can be retrieved in order. We also assume no processes crash, and none of the processes have access to a hardware clock.

2.2 Queue Definitions

We introduce the definition for a standard FIFO Queue abstract data type. We will use the special character $\perp$ to represent an empty queue.

▶ **Definition 1.** *A* Queue *over a set of values V is a data type with two operations:*

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- *Enqueue*$(val, -)$ *adds the value X to the queue*
- *Dequeue*$(-, val)$ *returns the oldest value in the queue.*

A sequence of queue operations is legal iff it satisfies the following conditions. The empty sequence is a legal sequence. Each Enqueue value is unique. Once a value has been enqueued to the queue once, it can not be enqueued again. If ρ *is a legal sequence of operation instances, then* $\rho \cdot$*Dequeue*$(-, val), val \neq \perp$ *is legal iff Enqueue(val, -) is the first Enqueue instance in* ρ *that does not already have a matching Dequeue*$(-, val)$ *in* ρ*. Furthermore,* $\rho \cdot$*Dequeue*$(-, \perp)$ *is legal iff every Enqueue*$(val, -)$ *in* ρ *has a matching Dequeue*$(-, val)$ *in* ρ*.*

3 Asynchronous FIFO Queues

3.1 Description

Each process stores a vector clock timestamp that holds its local view of the clocks at all other processes. Thus the vector clock timestamp at vi, dictating the vector clock view of process i is an array of size n (number of processes in this system) that is initially 0 at all indices. At the point when a process starts an *Enqueue*() or *Dequeue*() invocation, that process will increment their own index in the local clock timestamp, which marks a step or clock tick. Processes also update the local view of the vector clocks when they receive a message containing a timestamp from another process. This occurs each Enqueue and Dequeue invocation and response, as as a part of the payload of each of these messages, the timestamp at the point in which it was invoked is included. This results in processes regularly updating the local views of their clocks, resulting in processes being as updated as possible. To update the local vector clock, values at corresponding indexes in the local clock and message clock are compared, and the larger of the two values is set to the local view. By adjusting each of these indices, this guarantees that a local clock will have the largest of the two indices at all indices in the local clock, and therefore has the most advanced possible clock at that point in the execution. For any two vector timestamps $v_i i$ and v_j , such that $v_i \prec v_j$, states that v_i is a strictly smaller timestamp than v_j, if $v_i[x] < v_j[x], \forall x \in [0, \ldots, n-1]$. If this is not true, there is some index k, the first element that is different between the two vector timestamps v_i and v_j i.e. for $x = 0$ to $k-1, v_i[k] < v_j[x]$ but at index k, $v_i[k] \geq v_j[k]$. Then we say that v_i is lexicographically smaller than v_j, $v_i << v_j$ if $v_i[i] < v_j[i]$. Notice that $v_i \prec v_j$ implies that $v_i << v_j$ but not the opposite.

Within each process there is a local version of an augmented minimum priority queue keyed on lexicographic timestamp order. This priority queue can perform three operations: $insert(value, v_{clock})$, $get(position)$, $remove(value)$. The insert operation adds the value to the queue based on the vclock as a priority. The $get(position)$ function allows the user to peek into the element at the passed position without removing a value from the queue. The $remove(value)$ function removes the specific value passed to it from the queue which can be at any location in the queue. Ordering elements in FIFO order is not a straight-faced task in a distributed setting since defining which invocation happened first is not as clear as in a linear setup. Consequently using a priority queue keyed on lexicographic timestamp order allows us to ensure some order locally to guarantee FIFO consistency to the user once linearized.

3.1.1 Confirmation Lists

The main structure of this algorithm it to utilize a structure we will define called a *Confirmation List* to track the responses for a given dequeue locally for each process. Upon

receiving a dequeue request message, a given process will either declare that dequeue ”safe” or ”unsafe” if that process is or isn't in the active process of dequeuing an element. If a process will declare that message safe, it sends a message confirming that to the process that invoked the dequeue, and if it will declare the message unsafe, it sends a message stating that to all processes in the system. Both safe and unsafe messages will be marked with the invoking process, the responding process, and the invoking dequeue vector clock.

Upon hearing about a dequeue request, which are sent globally, a process will instantiate a Confirmation List in it's local memory, indexed by the timestamp of the invoking processes dequeue. For the dequeue invoking process, the confirmation list will fill by recieving messages from the other processes in the system with a corresponding timestamp. A safe message will be marked as a 1 in the corresponding index of the process in the Confirmation List and an unsafe will be marked as a 2. Once the Confirmation List has no undefined indices, the number of 2's within the list will be counted, and the queue will be accessed at that index, and the element removed.

For processes that are not the invoking process, only the unsafe messages are received, and marked as 2 in the local Confirmation Lists. To fill the rest of the list, messages with timestamps strictly greater than the invoking timestamp will be treated as implicit ”safe” messages, as there cannot be an unsafe message that has not been received from that process. Thus, with the same number of unsafe messages, the non-invoking processes will naturally come to the same conclusion and remove the same element as the invoking processes. Confirmation lists, like enqueue requests are sorted by timestamp order, as when they are created, the invoking dequeue timestamp is included. The general structure of a of conf list is as follows [[Flags for responses from a given process][Invoking Process ID][Timestamp of invocation]]

3.2 Algorithm

3.3 Correctness

▶ **Lemma 2.** *Every invocation has a matching response.*

Proof. Based on teh specifications of the algorithm, if there is an *Enqueue*, the request ffor that is sent globally and responded to. In teh case of a *Dequeeu*, the request is sent globally, and the response for that *Dequeue* iss also sent globally. In all cases of requests being sent, there is a delay of $2d$, where d is the maximum message delay. ◀

▶ **Lemma 3.** *The local views of the queues are equivalent at the time of Dequeue.*

Proof. For this algorithm, we will consider the entire history of the execution, such that all elements that were at one point in the queue are considered a part of the prefix of the queue. Once an element is dequeued, that element remains a part of the prefix, but no longer can be accessed, and is marked unavailable for dequeues.

While processes may have different complete histories for the histories of their enqueues at some point in time, given they may be in different local states with respect to active enqueues that they may have not recieved, the prefix of all local views up to the timestamp of the local execution of a dequeue will be the same. This is due to the fact that there is an agreed upon order for sorting the enqueues, and as such all processes will agree on the order. Furthermore, at the time of a dequeue, we can confirm that a given process will have heard of all prior enqueues. This is due to the fact that in order to locally execute a dequeue, a given process will have to have heard all messages prior to that dequeue, via the Confirmation List structure. The nature of the communication between processes being FIFO results in

the knowledge that there cannot have been earlier messages that have gone unrelieved if a later message has been processed and therefore, if a process is actively completing a dequeue invocation, at the point of that invocation every message prior has been received. Thus, we can formally state that at such a point, the prefix of all messages and queue states are equal across all processes. ◀

▶ **Lemma 4.** *In order to dequeue an element, there must be a corresponding Enqueue.*

Proof. Proof by contradiction. Assume that some process invokes a dequeue, receives responses and locally executes, removing some element that was not enqueued. That element, by the algorithm definition, must be held within the local view of the queue. This is because dequeues are accessed via the index provided by the unsafes. Therefore, the element that was dequeued by the process must exist in the local queue. The only way that can occur if the element was enqueued. Additionally, when removing an element from the queue, it is important to not remove elements with a larger timestamp that the dequeue. In the case in which there is a dequeue request and a timestamp where $dqts < eqts$, and the dequeue executes locally, given we cannot linearize the enq before the deq, we must return $\perp$ and not the element. This behavior is defined in the algorithm (IM LYING ITS NOT WE NEED TO ADD IT :D), and allows us to establish a specified order. ◀

▶ **Lemma 5.** *Elements that are dequeued are unique. (No double dequeue.)*

Proof. Given that at the time of execution we have proved that the head of the list of enqueues, combined with the algorithm definition guaranteeing that the local conf list of all processes contains the same information, we can state that all processes at the time of local execution will act on the same information to select an index, and therefore will select the same index. Given that, we can state that they will remove the same element from the queue, regardless of the time of execution locally. ◀

▶ **Theorem 6.** *Algorithm* **??** *correctly implements a FIFO queue.*

Proof. Based on the information of the lemmas, we can conclude that the algorithm definition for the queue follows the specifications for the ADT. Enqeues have corresponding dequeues, no double dequeues, we can linearize by timestamp such that the first element in the queue is always removed first. ◀

3.4 Complexity

4 Asynchronous Out-of-Order Queues

4.1 Description

4.2 Algorithm

4.3 Correctness

In the case where $k > n$, since the number of messages required to have a successful dequeue in bounds is less than the number of permitted dequeues ooo, dequeues can be instant. This information is based off of PUT THE PAPER NAME HERE which utilizes prior labeling of elements within the queue structure with a process that is allowed to instantly dequeue it. Given each element is distinct and can only be dequeued once, the labeling must be distinct, in that each process agrees on the labelling that a given value in the queue receives. Whereas in PAPER NAME AGAIN this is done via a process of synchronization of clocks, we will be

Algorithm 1 Code for each process p_i to implement a Queue with out-of-order k-relaxed *Dequeue*, where $k \geq n$ and $l = [k/n]$

```
1: function ENQUEUE(val)
2:     EnqCount = 0
3:     updateTS(v_i)
4:     enq_timestamp = v_i
5:     send (EnqReq, val, i, enq_timestamp) to all processes
6: end function
7: function RECEIVE(EnqReq, val, j, enq_timestamp) from p_j
8:     updateTS(v_i, v_j)
9:     if enq_timestamp not in Pending_Enqueues then
10:        Pending_Enqueues.insertByTS(enq_timestamp, val)
11:    end if
12:    send (EnqAck, i) to p_j
13: end function
14: function RECEIVE(EnqAck from p_j)
15:    EnqCount+ = 1
16:    if EnqCount == n then
17:        if localQueue.size < k then
18:            send (EnqConfirm, enq_timestamp) to all processes
19:        end if
20:    end if
21:    return EnqResponse
22: end function
23: function RECEIVE(EnqConfirm, enq_timestamp from p_j)
24:    localQueue.insertByTS(Pending_Enqueues.getByTS(enq_timestamp))
25:    if clean == true and localQueue.size() ≤ k then ▷ localQueues agree by this point
26:        let procNum = (localQueue.size() − 1  mod n
27:        localQueue.label(p_procNum, localQueue.tail)      ▷ I may have mangled this line
28:    end if
29: end function
```

Algorithm 2 Continued, part 2

```
1: function DEQUEUE
2:     v_i += 1
3:     let Deq_ts = v_i
4:     if localQueue.peekByLabel(p_i) ≠ ⊥ then        ▷ Check that I didn't change this
5:         let ret = localQueue.deqByLabel(p_i)
6:         send (Deq_f, ret, Deq_ts) to all processes
7:     else
8:         send (Deq_s, null, Deq_ts) to all processes
9:     end if
10: end function
11: function RECEIVE(deq_f, val, Deq_ts) from p_j)
12:     if j ≠ i then localQueue.remove(val)
13:     end if
14: end function
15: function RECEIVE(deq_s, val, Deq_ts from p_j)
16:     UpdateTs(v_i, Deq_ts)
17:     if Deq_ts is not in PendingDequeues then
18:         PendingDequeues.insertByTs(createList(Deq_ts, p_invoker))        ▷ Check line
19:     end if
20:     let p_invoker = p_j ▷ This doesn't make sense to me? What are you doing on this line?
21:     if Deq_ts ≠ 0 and Deq_ts < v_i then
22:         send (Unsafe, Deq_ts, i, p_invoker) to all processes
23:     else
24:         send (Safe, Deq_ts, i, p_invoker) to all processes
25:     end if
26: end function
```

Algorithm 3 Continued, part 3

```
1: function RECEIVE(Safe/Unsafe, Deq_ts, j, p_invoker)
2:     if Deq_ts not in PendingDequeues then
3:         PendingDequeues.insertByTs(createList(Deq_ts, p_invoker))
4:     end if
5:     for confirmationList in PendingDequeues do
6:         if confirmationList.ts == Deq_ts then
7:             if Unsafe then
8:                 response = 2
9:             else
10:                response = 1
11:            end if
12:            confirmationList.list[j] = response
13:        end if
14:        propagateEarlierResponses(PendingDequeues)
15:    end for
16:    for (index, confirmationList) in PendingDequeues do
17:        if not confirmationList.contains(0) and not confirmationList.handled then
18:            pos = 0
19:            for response in confirmationList.list do
20:                if response == 2 then
21:                    pos+ = 1
22:                end if
23:            end for
24:            confirmationList.handled = True
25:            updateUnsafes(Lists, index)
26:            ret = localQueue.deqByIndex(pos)
27:            labelElements(p_invoker)
28:            if i == p_invoker then
29:                return ret
30:            end if
31:        end if                          ▷ Not sure I left the nesting right on these.
32:    end for
33: end function
```

adjusting the labelling to occur at points when we can guarantee that the view of distinct processes agree, at the point of dequeueing. As we have discussed, at the point in which a process dequeues, the information prior to that dequeue, the prefix to the execution ρ to that point will be the same for all processes. Given that all processes will agree on a select set of information at the same relative point in their executions, we can treat this as a sort of synchronization, and make decisions as to the labelling of elements at this point.

For this relaxation to still follow the established queue definition, it must only return elements once. Therefore, we must prove that a) Each process dequeues each element only once and b) that elements removed are within the first n elements within the queue. Given that the labeling is agreed upon by every process, and done by calculations with an invariant, we can state that each process will agree on the elements that are ”fast dequeued” with respect to which process is allowed to ”fast dequeue” them. Given that processes agree which elements are ”reserved” for a fast dequeue by another process, we can furthermore state that a process will not slow dequeue an element that has been marked.

Thus, elements can not be dequeued by processes that they have not been marked for, so we can guarantee that they will ONLY be dequeued by the assigned process. By thje definition of the algorithm on line GIVE ME THE LINE NUMBER HERE COLE we can also state that the second criterion is satisfied, as at a point where a given process will attempt to fast dequeue, the position of the element that is attempting to be fast dequeued is checked, and the process is only allowed to execute the operation in the case where the element is within the legal bounds for dequeueing.

4.3.1 Less Relaxation than Number of Processes

In the case in which the X of out of order is less than the number of processes,there isn't an order of magnitude benefit to relaxation. The benefit of relaxation is that rather than forcing a wait of 2 times the maximum message delay, the wait can be cut to 2 times the (n-x+1)th slowest message delay. In a close connection system, the benefit is minimal as the difference between the max message delay and the (n-x+1)th is minimal, but in an actualized system with nonstandard message delays, this could have some substantial benefits.

4.4 Complexity

5 Conclusion