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. 23:1–23:6
Leibniz International Proceedings in Informatics
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

3.4 Complexity

4 Asynchronous Out-of-Order Queues

4.1 Description

4.2 Algorithm

4.3 Correctness

4.4 Complexity

5 Conclusion

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

```
function ENQUEUE(val)
    EnqCount = 0
    updateTS(v_i)
    enq_timestamp = v_i
    send (EnqReq, val, i, enq_timestamp) to all processes
end function
function RECEIVE(EnqReq, val, j, enq_timestamp) from p_j
    updateTS(v_i, v_j)
    if enq_timestamp not in Pending_Enqueues then
        Pending_Enqueues.insertByTS(enq_timestamp, val)
    end if
    send (EnqAck, i) to p_j
end function
function RECEIVE(EnqAck from p_j)
    EnqCount+ = 1
    if EnqCount == n then
        if localQueue.size < k then
            send (EnqConfirm, enq_timestamp) to all processes
        end if
    end if
    return EnqResponse
end function
function RECEIVE(EnqConfirm, enq_timestamp from p_j)
    localQueue.insertByTS(Pending_Enqueues.getByTS(enq_timestamp))
    if clean == true and localQueue.size() ≤ k then ▷ localQueues agree by this point
        let procNum = (localQueue.size() − 1  mod n
        localQueue.label(p_procNum, localQueue.tail)        ▷ I may have mangled this line
    end if
end function
```

Algorithm 2 Continued, part 2

function DEQUEUE
  $v_i += 1$
  let $Deq_{ts} = v_i$
  if $localQueue.peekByLabel(p_i) \neq \perp$ **then** ▷ Check that I didn't change this
    let $ret = localQueue.deqByLabel(p_i)$
    send (Deq_f, ret, Deq_{ts}) to all processes
  else
    send $(Deq_s, null, Deq_{ts})$ to all processes
  end if
end function
function RECEIVE(deq_f, val, Deq_{ts}) from p_j)
  if $j \neq i$ **then** $localQueue.remove(val)$
  end if
end function
function RECEIVE$(deq_s, val, Deq_{ts}$ from p_j)
  $UpdateTs(v_i, Deq_{ts})$
  if Deq_{ts} is not in $PendingDequeues$ **then**
    $PendingDequeues.insertByTs(createList(Deq_{ts}, p_{invoker}))$ ▷ Check line
  end if
  let $p_{invoker} = p_j$ ▷ This doesn't make sense to me? What are you doing on this line?
  if $Deq_{ts} \neq 0$ and $Deq_{ts} < v_i$ **then**
    send $(Unsafe, Deq_{ts}, i, p_{invoker})$ to all processes
  else
    send $(Safe, Deq_{ts}, i, p_{invoker})$ to all processes
  end if
end function

Algorithm 3 Continued, part 3

```
function RECEIVE(Safe/Unsafe, Deq_ts, j, p_invoker)
    if Deq_ts not in PendingDequeues then
        PendingDequeues.insertByTs(createList(Deq_ts, p_invoker))
    end if
    for confirmationList in PendingDequeues do
        if confirmationList.ts == Deq_ts then
            if Unsafe then
                response = 2
            else
                response = 1
            end if
            confirmationList.list[j] = response
        end if
        propagateEarlierResponses(PendingDequeues)
    end for
    for (index, confirmationList) in PendingDequeues do
        if not confirmationList.contains(0) and not confirmationList.handled then
            pos = 0
            for response in confirmationList.list do
                if response == 2 then
                    pos+ = 1
                end if
            end for
            confirmationList.handled = True
            updateUnsafes(Lists, index)
            ret = localQueue.deqByIndex(pos)
            labelElements(p_invoker)
            if i == p_invoker then
                return ret
            end if
        end if                          ▷ Not sure I left the nesting right on these.
    end for
end function
```