# 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. While there exists existing solutions for queues in this model, none offer full replication, and therefore the ability to extend to fault tolerant systems. 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 $\Pi = [p_0, \dots, p_{n-1}]$ 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 is accomplished by establishing a buffer for incoming messages for each process, and that when messages are sent, they are explicitly marked in the order they are sent. 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. 23:1–23:7
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 $\rho$ 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 $\rho$ that does not already have a matching $Dequeue(-, val)$ in $\rho$. Furthermore, $\rho \cdot Dequeue(-, \perp)$ is legal iff every $Enqueue(val, -)$ in $\rho$ has a matching $Dequeue(-, val)$ in $\rho$.*

# 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. We will refer to this vector clock time stamp in terms of $v_i$ where i represents the local vector clock view of some external process i. The vector clock 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 invokes 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 at the point of invocation of 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 will be up to date with the view of other process' progress through 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-forward 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 is to utilize a structure we will define called *Confirmation List* to track the acknowledgements from other processes for a given dequeue locally for each process. For each process, upon receiving a dequeue request message, that process will either declare that dequeue ”safe” or ”unsafe” if that process isn't or is respectively in the active process of dequeuing an element. The process will then globally send a message containing it's response to the invocation, declaring if it views the dequeue invocation safe or unsafe. Both safe and unsafe messages will be marked with the invoking process, the responding process, and the invoking dequeue vector clock.

When a process first recieves information about an invocation for a dequeue, it creates a Confirmation List object, keyed to the timestamp of the dequeue invocation and stores that object. This information can be in the form of a dequeue request, the local process dequeueing, or a response to a dequeue invocation from a third party process. At it's inception, the Confirmation List is empty, and will await the responses to the dequeue invocation. 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 indices containing 0's, the number of unsafe responses, denoted by 2's within the list will be counted, and the queue will be accessed at that index, and the element removed.

At the point of the processing of a dequeue via the confirmation list structure, the confirmation list lexicographically after the executed instance must also be updated. In the case where an invoking process i has a dequeue processed (Need a better word), there is a need to update the responses within Confirmation Lists stored lexicographically after the processed Confirmation List. This is due to the fact that a process i will return an unsafe response to any dequeue request during the period between when it invokes and when it consumes? an element, and once the invocation is processed, the element that the process is effectively "reserving" with the unsafe responses will have been removed. Therefore, all Confirmation Lists lexicographically later stored in memory must be updated at the index i from representing an unsafe response to representing a safe response, to represent this updated view of the execution history. This updating must be done for all Confirmation Lists prior to the next invocation of process i, as at that point the system will have an updated view of it's own execution.

## 3.2 Algorithm

## 3.3 Correctness

In order to prove that this algorithm alligns with the Queue specification set forth, we will consider an arbitrary admissable run of invocations denoted R. We construct $\pi$, a permutation of all operation instances in R, by ordering operations in ascending order based on their lexicographic vector timestamp values.

▶ **Lemma 2.** *$\pi$ respects the real time order of non-overlapping instances*

**Proof.** Proof by contradiction. Let there be two non-overlapping operation instances $op1$ and $op2$, where $op1$ is invoked prior to $op2$ in real time order. We will then linearize $op2$ prior to $op1$ in $\pi$. Given that we have stated that $op1$ occurs before $op2$ in real time, and that the two operation instances are non-overlapping, by the definition of the algorithm, $op1$ must have therefore completed its execution prior to the invocation of $op2$. The completion of $op1$ requires acknowledgement from the invoking process of $op2$ regardless of the type of invocation, and therefore it consequently follows that $op2$ must have a larger timestamp than $op1$. Due to the fact that $op2$ must have recieved and acknowledged $op1$, it follows that it must have a larger timestamp, and thereofre must be linearized after $op1$. THerefore $op1$ must linearize before $op2$, which results in a contradiction to our asusmption that $op2$ preceeds $op1$ ion $\pi$ ◀

▶ **Lemma 3.** *Every invocation has a matching response.*

**Proof.** Based on the specifications of the algorithm, if there is an *Enqueue*, the request for that is sent globally and responded to only to the invoking process. In the case of a *Dequeue*, the request is sent globally, and the response for that *Dequeue* iss also sent globally. ◀

▶ **Lemma 4.** *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.

Though processes 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 with a lexicographically smaller timestamp to that dequeue, via the Confirmation List structure. This is due to the fact that in order to complete a dequeue execution, the Confirmation List strucuture must be filled, meaning that it must have recieved a message responding to the invocation from every other process in the system. Given that a message can only be acted upon in the case that it is the oldest unactioned message in the communication history between the two processes, we can be certain that there are no prior inter-process messages from a process which has a filled index in a confirmation list. The nature of the communication between processes being FIFO results in the knowledge that there cannot have been earlier messages that have gone unrecieved if a later message has been recieved and therefore, if a process is actively completing a dequeue invocation, at the point of that local execution, every message with a smaller timestamp 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 5.** *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, and allows us to establish a specified order. ◀

▶ **Lemma 6.** *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 7.** *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

The asynchronous out of order queue is a relaxation of the existing queue structure that relaxes the specification that the queue can only return the first *Enqueue* instance in $\rho$ that does not already have a matching $Dequeue(-, val)$ in $\rho$. Instead, it allows for the return of any of the first $K Enqueue$ instances in $\rho$ that do not already have a mathing $Dequeu(-, val)$ in $\rho$.

The K-OOO queue also adds the behavior of prior claiming of elements for a system we will be referring to as "fast dequeueing". The existing behavior for dequeuing will be refered to as a "slow dequeue", which still utilizes the same behavior of Confirmation Lists to replicate information across the network. At points where processes share information however, there is additional behavior developed that allows processes to mark values in the queue that only they will be able to access, that will require no communication to be dequeued. At the point of execution of a "slow dequeue" if a process has a value that is marked to be dequeued by by that process, and the value that is marked is within the bounds of K, that it is one of the first K values in the queue structure, that value can be removed immediately without establishing a Confirmation List and requiring rounds of inter-process communication. Processes that are not the process that has "claimed" the value will additionally view that at the point of the slow dequeue, there is a labeled element for a fast dequeue from the same process, and remove both values from their local view of the queue, in order to ensure the same veiw of the structure.

The process of labeling is done by guaranteeing that processes have the same view of the queue at certain points, and utilizing this similar view and invariant calculations to collectively agree on which processes claim which values. Given the processes agree on the ordering of values within the queue, the labeling of the elements and the ordering of slow dequeues, we can prove that fast dequeues do not disrupt the existing behavior of the Queue ADT.

**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($val$)
    $EnqCount = 0$
    $incrementTS(v_i)$
    $Enq_{ts} = v_i$
    send $(EnqReq, val, i, Enq_{ts})$ to all processes
end function
function RECEIVE($EnqReq, val, j, Enq_{ts}$) from $p_j$
    $updateTS(v_i, Enq_{ts})$
    $lQueue.insertByTS(val, j, Enq_{ts})$
    for $confirmationList$ in $PendingDequeues$ do
        if $confirmationList.ts \prec Enq_{ts}$ then
            $confirmationList.responses[j] = Safe$
        end if
    end for
    send $(EnqAck, i)$ to $p_j$
end function
function RECEIVE($EnqAck, j$ from $p_j$)
    $EnqCount+ = 1$
    if $EnqCount == n$ then return $EnqResponse$
    end if
end function
```

**Algorithm 2** Continued, part 2

```
function DEQUEUE
    $incrementTS(v_i)$
    let $Deq_{ts} = v_i$
    if $localQueue.peekByLabel(p_i) \neq \bot$ then
        let $ret = localQueue.deqByLabel(p_i)$
        send $(Deq_f, ret, i, Deq_{ts})$ to all processes
    else
        send $(Deq_s, null, i, Deq_{ts})$ to all processes
    end if
end function
function RECEIVE($(op, val, j, Deq_{ts})$ from $p_j$)
    $updateTs(v_i, Deq_{ts})$
    let $p_{invoker} = p_j$
    if $Deq_{ts}$ is not in $PendingDequeues$ then
        $PendingDequeues.insertByTs(createList(Deq_{ts}, p_{invoker}))$
    end if
    if $Deq_{ts} \neq 0$ and $Deq_{ts} \prec v_i$ then
        let $safetyFlag = Unsafe$
    else
        let $safetyFlag = Safe$
    end if
    send $(op, val, safetyFlag, Deq_{ts}, i, p_{invoker})$ to all processes
end function
```

**Algorithm 3** Continued, part 3

```
function RECEIVE($(op, val, safetyFlag, Deq_{ts}, i, p_{invoker})$ from $p_j$)
    if $Deq_{ts}$ not in $PendingDequeues$ then
        $PendingDequeues.insertByTs(createList(Deq_{ts}, p_{invoker}))$
    end if
    $propagateEarlierResponses(PendingDequeues)$
    for $(index, confirmationList)$ in $PendingDequeues$ do
        if $confirmationList.ts == Deq_{ts}$ then
            $confirmationList.responses[j] = safetyFlag$
        end if
        if $confirmationList.full()$ and not $confirmationList.handled$ then
            $confirmationList.handled = True$
            if $op == Deq_f$ and $p_i \neq p_{invoker}$ then
                $lQueue.remove(val)$
            end if
            $updateUnsafes(PendingDequeues, index)$
            if $op == Deq_f$ then
                return
            end if
            if $localQueue.peekByLabel(p_{invoker}) \neq \perp$ then
                let $ret = lQueue.deqByLabel(p_{invoker})$
            else
                let $pos$ = Number of $Unsafe$ flags in $confirmationList.responses$
                let $ret = lQueue.deqByIndex(pos)$
            end if
            $labelElements(p_{invoker})$
            $updateUnsafes(PendingDequeues, index)$
            if $p_i == p_{invoker}$ then
                return $ret$
            end if
        end if
    end for
end function
```

## 4.2 Algorithm

## 4.3 Correctness

For the K-OOO queue the same strucutre of a run R and a permutation $\pi$ is established. Lemmas 5.1, 5.2, and 5.4 remain unchanged, as the out of order behavior does not inherently adjust those behaviors of the queue structure. The Out of order behavior does not allow for the violation of real time order linearization, as it only allows for the structure to be accessed in an expanded manor. It also doesn't adjust how the there must be a matching response for any invocation or that there must be a corresponding enqueue for each dequeue invocations.

However, given that the relaxation adjusts the structures of dequeues, we must further prove that with this adjusted behavior the local view of the queues are equivalent at the time of dequeue, and that elements, when dequeued are unique.

▶ **Lemma 8.** *The local view of the queue across processes are equivalent at the time of dequeue*

**Proof.** blee ◀

▶ **Lemma 9.** *The claiming of elementy by individual processes will be agreed upon by all processes in the system*

**Proof.** blee ◀

▶ **Lemma 10.** *Elements that are dequeued are unique*

**Proof.** bibliography ◀

### 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

## References

1 Edward Talmage and Jennifer L. Welch. Improving average performance by relaxing distributed data structures. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 421–438. Springer, 2014. doi:10.1007/978-3-662-45174-8_29.