LINEARIZABLE RELAXATIONS OF STACKS AND THEIR GENERALIZATIONS TO ORDERED DATA STRUCTURES

Approved by:

Dr. Byron Boots, Advisor
School of Interactive Computing
Georgia Institute of Technology

Dr. Umakishore Ramachandran
School of Computer Science
Georgia Institute of Technology

Dr. Richard Peng
School of Computer Science
Georgia Institute of Technology

Date Approved: April 26, 2017
ACKNOWLEDGEMENTS

This work was supported in part by NSF grant 1526725. I am thankful for the guidance and mentorship provided by Edward Talmage and Prof. Jennifer Welch over the course of this research.
# TABLE OF CONTENTS

Acknowledgments .............................................. v

List of Tables ...................................................... ix

List of Figures ...................................................... x

Summary ........................................................... xi

Chapter 1: Introduction and Background ......................... 1

Chapter 2: Definitions and Model .............................. 4
  2.1 Relaxed Stacks .............................................. 4
  2.2 System Model ................................................ 9
  2.3 Correctness Condition ..................................... 11
  2.4 Performance Metrics ....................................... 12

Chapter 3: Algorithm and Upper Bounds ......................... 13
  3.0.1 Parameters ............................................... 14
  3.0.2 Local Variables ......................................... 14
  3.1 Out-of-Order Relaxed Stacks ............................... 17
    3.1.1 Minimal Implementation ............................. 17
    3.1.2 Improvements ......................................... 24
3.1.3 Correctness ................................................. 28
3.1.4 Performance ............................................. 44

Chapter 4: Generalizations and Extensions ............................. 49
4.1 Relaxed Priority Queues ....................................... 49
   4.1.1 Reductions .............................................. 50
   4.1.2 Relaxed Queues as a Special Case ..................... 54

Chapter 5: Conclusion ................................................ 62

Appendix A: Additional Algorithms and Upper Bounds ............ 64
   A.1 Lateness Relaxed Stacks ................................. 64
       A.1.1 Correctness .......................................... 67
       A.1.2 Performance ........................................ 70
   A.2 Windowed Relaxed Stacks ............................... 71
       A.2.1 Correctness .......................................... 75
       A.2.2 Performance ........................................ 78
   A.3 Restricted Relaxed Stacks ............................... 79
       A.3.1 Correctness .......................................... 82
       A.3.2 Performance ........................................ 87
   A.4 Stuttering Relaxed Queues .............................. 87
       A.4.1 Correctness .......................................... 89
       A.4.2 Performance ........................................ 91

Appendix B: Sample of Derived Algorithms ............................ 93
LIST OF TABLES

4.1 Bounds on *Push/Insert* Time Complexity∗ . . . . . . . . . . . . . . . . . 53
4.2 Bounds on *Pop/ExtractMax* Time Complexity . . . . . . . . . . . . . . . . . 53
4.3 Bounds on *Deque* Time Complexity in Heavily-Loaded Runs Only** . . 59
4.4 Bounds on *Deque* Time Complexity** . . . . . . . . . . . . . . . . . . . . . 60
4.5 Upper Bounds without Theorem 20’s Optimizations in the General Case . . 61
LIST OF FIGURES

3.1 Partitioning elements among the processes using labels, with \( k = 6, n = 3 \). . 18

3.2 Invocation \((I)\), response \((R)\), and local execution \((E)\) of \( Pop \) instances. . . 19

3.3 The safe and contention regions. . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.4 (a) Non-short-circuited and (b) short-circuited \( Pops \), with \( l - \tau = 2 \). In each case, \( \uparrow \) indicates the earliest time another \( Pop \) instance can be invoked. 25

3.5 Plots of the amortized time complexities of \( Push \) (blue) and \( Pop \) (red) parameterized by \( \tau \), where \( l = 10 \) and \( d \) is an arbitrary constant. . . . . . . 48
SUMMARY

Relaxation of semantics is a technique for improving the amortized distributed time complexity of data structures. We exhibit the first known algorithms implementing linearizable relaxed stacks in a partially synchronous, message-passing system, and proceed to show that relaxed priority queues reduce to relaxed stacks, meaning that their implementations are equally as fast in terms of amortized performance. Furthermore, restricting these new algorithms to relaxed queues improves on the previously best known upper bounds.
CHAPTER 1
INTRODUCTION AND BACKGROUND

Distributed computing infrastructures have played an increasingly large role in business, government, and scientific research in recent years, and reliance on these infrastructures, which include grids and clouds, will only continue to grow. For instance, the mission of the Worldwide LHC Computing Grid according to its website (wlcg-public.web.cern.ch) is “to provide computing resources to store, distribute, and analyze the data generated by the Large Hadron Collider (LHC), making the data equally available to all partners, regardless of their physical location.” Perhaps a more immediate example of a sizable collection of distributed systems is the Internet itself, whose users now include over half of the world’s population and nearly ninety percent in North America. From the perspective of researchers, the growing complexity of these infrastructures may be addressed by breakthroughs in theoretical knowledge regarding their structure and verification.

One of the goals of distributed computing is to ensure that the participating processes always have the most up-to-date information regarding the system of interest. In a message-passing system, each process is unaware of any other’s actions unless the other communicates to it directly, but the illusion of a shared state must still be maintained in order to derive any benefit over a non-distributed system. For example, operating systems such as Windows and Linux support remote procedure call protocols which allow a procedure to execute in a separate address space while appearing as a normal procedure call to the user, helping to expedite the development of client-server applications. It is crucial in these situations that each party is guaranteed to have non-contradicting knowledge of shared information while minimizing the amount of time spent on interprocess communication.

To organize informational interactions, we often use ordered data types which include queues, stacks, and priority queues. These describe sets of data in which elements can be
added, retrieved, or removed one at a time by operations called by separate processes, but
differ in the order in which elements are retrieved or removed. In queues, elements added
first have the highest priority, while stacks follow the opposite order in which these are the
elements added last; priority queues are a generalization of the first two in which elements
are intrinsically given their own priorities according to any specific function.

In order to evaluate orderings on operation instances in concurrent executions based
on the specification of the data structure, many different types of consistency conditions
have been proposed. The strongest of these consistency conditions is linearizability, in
which instances of operations called by processes can be ordered in the same manner in all
processes, ensuring that the processes always finish in the same shared state [1].

Previous research on lower and upper bounds of strict queues and stacks has shown that
mixed accessor/mutator operations such as Dequeue and Pop must be slow in the worst
case to satisfy linearizability, because the calling process is required to wait for all other
processes to communicate their preceding operation instances [2]. More generally, lin-
earizable data structures are costly to implement in both shared-memory [3] and message-
passing systems [4], [5].

One workaround has been to replace linearizability with weaker consistency conditions,
such as quasi-linearizability [6], sequential consistency [7], eventual consistency [8], qui-
escent consistency [9], and others, to yield improved performance [5]. The negation of the
presence of the shared state has been the foremost drawback of these approaches, however.

More recently, attention has shifted to relaxing instead the semantics of operations [10],
[6]. Examples of relaxations, in terms of a parameter $k$, include

- the Out-of-Order relaxation which allows Dequeue to remove any of the $k$ highest-
priority elements (i.e., best candidates),

- the Lateness relaxation which allows up to $k - 1$ Dequeue instances to remove any
element without restriction before the next Dequeue instance must remove the top-
most element, or single element of highest priority, and
• the Restricted relaxation, defined by the requirement of satisfying the first two.

[11] was the first to investigate lower and upper bounds of relaxed ordered data structures, and discovered that relaxation provided beneficial results for amortized time complexity. In particular, lower bounds were given for a comprehensive variety of relaxed queues and stacks, and upper bounds were provided for the Out-of-Order $k$-Relaxed Queue and Restricted $k$-Relaxed Queue.

At the outset, we conceive of a number of different types of relaxed stacks (Section 2). Then we introduce the extensive theory leading to an algorithm implementing an Out-of-Order $k$-Relaxed Stack (Section 3), forming the basis for algorithms implementing several other types of “stronger” relaxed stacks (Appendix A).

Section 4 focuses on further extensions of these results. The latter phase of our paper consists of prescribing an algorithmic reduction from relaxed priority queue implementations to those of relaxed stacks. Finally, we come around full circle by restricting relaxed priority queues to relaxed queues which, when combined with optimizations tailored for the new techniques, improve on the previously discovered upper bounds.
CHAPTER 2
DEFINITIONS AND MODEL

We begin by defining abstract data types in the sequential setting, in the same manner as described in [11].

A data type $T$ of an object provides a set of operations $OPS(T)$ defined on this type. Each operation $OP$ in $OPS(T)$ has an invocation $OP_{inv}$, which may include an argument, and a response $OP_{resp}$, which may include a return value. Any invocation $OP_{inv}(arg)$ and its immediately following response $OP_{resp}(ret)$ are indivisible in the sequential case, so the pair can be written as $OP(arg, ret)$, which is called an instance or operation instance of $OP$.

The semantics of a data type $T$ are defined by a set of operation instances $L(T)$, called the legal sequences, which must satisfy the following properties:

- **Prefix Closure**: If $\rho$ is in $L(T)$, then every prefix of $\rho$, including the empty sequence, is in $L(T)$.

- **Completeness**: If $\rho$ is in $L(T)$, then for every argument $arg$ and every operation invocation $OP_{inv}$, there exists a response $OP_{resp}(ret)$ for $OP_{inv}(arg)$ such that $\rho \cdot OP(arg, ret)$ is in $L(T)$.

2.1 Relaxed Stacks

We initially define the strict or unrelaxed Stack data structure as a baseline.

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

- $Push(val, -), val \in V$,
- $Pop(-, val), val \in V \cup \{\bot\}$. 
Given a sequence $\rho$ of operation instances on a Stack, if we define the following terms:

- A $\text{Push}(val, -)$ is matched if there exists a $\text{Pop}(-, val)$ in $\rho$. If this is the case, $\text{Push}(val, -)$ is said to match $\text{Pop}(-, val)$ in $\rho$.
- A $\text{Pop}(-, val)$ in $\rho$ is headmost if there exists a $\text{Push}(val, -)$ in $\rho$ such that every $\text{Push}(val', -)$ succeeding $\text{Push}(val, -)$ in $\rho$ is matched by a $\text{Pop}(-, val')$ appearing in $\rho$ before $\text{Pop}(-, val)$.

then $\rho$ is legal iff it satisfies the following conditions:

(C1) The empty sequence is legal.

(C2) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Push}(val, -)$ is legal iff there is no $\text{Push}(val, -)$ already in $\rho$.\footnote{Unique values can easily be achieved by timestamping elements as they are added.}

(C3) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Pop}(-, val)$, $\text{val} \neq \bot$ is legal iff $\text{Pop}(-, val)$ is headmost.

\[ \rho \cdot \text{Pop}(-, \bot) \text{ is legal iff every instance of } \text{Push} \text{ in } \rho \text{ is matched.} \]

In particular, $\text{Pop}(-, \bot)$ is always a $\text{Pop}$ that is not headmost.

We now describe a number of different ways to relax the Stack data structure, each of which is defined in terms of a relaxation parameter $k$ specifying the degree of relaxation. To clarify, the $\text{Pop}$ operation is what will be relaxed in particular.

Intuitively, the following definition describes the relaxation in which a $\text{Pop}$ may remove any of the top $k$ elements.

**Definition 2.** An Out-of-Order $k$-Relaxed Stack over a set of values $V$ provides the same operations as in Definition 1.

A sequence $\rho$ of operation instances on an Out-of-Order $k$-Relaxed Stack is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as
(C3) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Pop}(-, \text{val}), \text{val} \neq \bot$ is legal iff there is an unmatched $\text{Push}(\text{val}, -)$ in $\rho$ that is succeeded by fewer than $k$ unmatched $\text{Push}$ instances in $\rho$.

Furthermore, $\rho \cdot \text{Pop}(-, \bot)$ is legal iff there are fewer than $k$ unmatched $\text{Push}$ instances in $\rho$.

In the definition that immediately follows, successive $\text{Pop}$ instances may remove elements from anywhere in the stack (or even $\bot$) up to $k - 1$ times before being required to remove the topmost element.

**Definition 3.** A Lateness $k$-Relaxed Stack over a set of values $V$ provides the same operations as in Definition 1.

Given a sequence $\rho$ of operation instances on a Lateness $k$-Relaxed Stack, if we define the following additional terms:

- A $\text{Pop}(-, \text{val})$ in $\rho$ is resolved if $\text{Pop}(-, \text{val})$ is either headmost or succeeded by a headmost $\text{Pop}$ instance.

  A value $\text{val}$ is resolved in $\rho$ if there exists a $\text{Pop}(-, \text{val})$ in $\rho$ that is resolved.

  A matched $\text{Push}(\text{val}, -)$ in $\rho$ is resolved if some $\text{Pop}(-, \text{val})$ it matches is resolved.

- The lateness of $\rho$, $\lambda(\rho)$, is the number of unresolved $\text{Pop}$ instances in $\rho$.

then $\rho$ is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Pop}(-, \text{val})$ is legal iff there is an unmatched $\text{Push}(\text{val}, -)$ in $\rho$ and $\lambda(\rho \cdot \text{Pop}(-, \text{val})) < k$.

Furthermore, $\rho \cdot \text{Pop}(-, \bot)$ is legal iff every instance of $\text{Push}$ in $\rho$ is matched or $\lambda(\rho \cdot \text{Pop}(-, \bot)) < k$.

The next definition is identical except that lateness is replaced by a new concept called current-top lateness. This time the intent is to keep track of a separate lateness counter for each element, which comes into effect when it is the topmost element.
Definition 4. A Per-Top Lateness $k$-Relaxed Stack over a set of values $V$ provides the same operations as in Definition 1.

Given a sequence $\rho$ of operation instances on a Per-Top Lateness $k$-Relaxed Stack, if we define the following additional terms:

- A $\text{Pop}(\neg,\text{val})$ in $\rho$ is own-top resolved if $\text{Pop}(\neg,\text{val})$ is either headmost or succeeded by a $\text{Pop}$ instance whose matching $\text{Push}$ instance is the last $\text{Push}$ instance preceding $\text{Pop}(\neg,\text{val})$.

A value $\text{val}$ is own-top resolved in $\rho$ if there exists a $\text{Pop}(\neg,\text{val})$ in $\rho$ that is own-top resolved.

A matched $\text{Push}(\text{val},\neg)$ in $\rho$ is own-top resolved if some $\text{Pop}(\neg,\text{val})$ it matches is own-top resolved.

- The current-top lateness of $\rho$, $\gamma(\rho)$, is the number of own-top unresolved $\text{Pop}$ instances succeeding the last unmatched $\text{Push}$ in $\rho$.

then $\rho$ is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Pop}(\neg,\text{val}), \text{val} \neq \bot$ is legal iff there is an unmatched $\text{Push}(\text{val},\neg)$ in $\rho$ and $\gamma(\rho \cdot \text{Pop}(\neg,\text{val})) < k$.

Furthermore, $\rho \cdot \text{Pop}(\neg,\bot)$ is legal iff every instance of $\text{Push}$ in $\rho$ is matched or $\gamma(\rho \cdot \text{Pop}(\neg,\bot)) < k$.

We also present two types of relaxed stacks which combine the out-of-order and lateness restrictions.

Definition 5. A Restricted $k$-Relaxed Stack over a set of values $V$ provides the same operations as in Definition 3.

A sequence $\rho$ of operation instances on a Restricted $k$-Relaxed Stack is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as
(C3) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Pop}(-,\text{val})$, $\text{val} \neq \bot$ is legal iff there is an unmatched $\text{Push}(\text{val},-) \in \rho$ that is succeeded by fewer than $k - \lambda(\rho \cdot \text{Pop}(-,\text{val}))$ unmatched Pushes in $\rho$.

Furthermore, $\rho \cdot \text{Pop}(-,\bot)$ is legal iff there are fewer than $k - \lambda(\rho \cdot \text{Pop}(-,\bot))$ unmatched Pushes succeeding the last unmatched Push.

**Definition 6.** A Windowed $k$-Relaxed Stack over a set of values $V$ provides the same operations as in Definition 3.

A sequence $\rho$ of operation instances on a Windowed $k$-Relaxed Stack is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If $\rho$ is a legal sequence of operation instances, then $\rho \cdot \text{Pop}(-,\text{val})$, $\text{val} \neq \bot$ is legal iff there is an unmatched $\text{Push}(\text{val},-) \in \rho$ that is succeeded by fewer than $k$ Pushes that are either unmatched or unresolved in $\rho$.

Furthermore, $\rho \cdot \text{Pop}(-,\bot)$ is legal iff there are fewer than $k$ Push instances that are either unmatched or unresolved.

Of the last two definitions, the first states that the top “$k$ minus lateness” elements are the set of elements legal for a $\text{Pop}$ to return. The second states that the legal set of elements that can be $\text{Popped}$ includes all present elements in a window of (constant) size $k$ from the top, where windows include all present and unresolved elements; an informal description is that $\text{Pushing}$ an element “shifts the window up” by 1, while $\text{Popping}$ the topmost element resolves all the unresolved elements so that they disappear from the window, and then “shifts the window down” until it is aligned with the topmost element present in the stack.

We can also state the Per-Top versions of Definitions 5 and 6 using current-top lateness and own-top unresolved elements just as Definition 4 does for Definition 3. What is interesting to consider in the case of a Per-Top Windowed $k$-Relaxed Stack is that the own-top unresolved elements are invisible, or not contained in the window, when the element whose
removal would resolve them is not currently the top of the stack, but they become visible again if that element returns to the top.

Our final definition allows Pop instances to leave the stack unchanged (in other words, to act as instances that perform a peek operation) up to $k - 1$ times for each element.

**Definition 7.** A Stuttering $k$-Relaxed Stack over a set of values $V$ provides the same operations as in Definition 1.

Given a sequence $\rho$ of operation instances on a Stuttering $k$-Relaxed Stack, if we define the following additional terms:

- **The stutter count** of a $\text{Pop}(\_, val), val \neq \bot$ is the number of times $\text{Pop}(\_, val)$ appears in $\rho$.

  The stutter count of a value $val$, $s(val)$, is the stutter count of $\text{Pop}(\_, val)$ in $\rho$.

- **The stutter count** of $\rho$, $s(\rho)$, is the stutter count of the last Pop instance in $\rho$.

- A $\text{Pop}(\_, val), val \neq \bot$ in $\rho$ is removed if $s(val) = k$ or $\text{Pop}(\_, val)$ is succeeded by a $\text{Pop}(\_, val')$ where $val' \neq val$.

  A value $val$ is removed in $\rho$ if there exists a $\text{Pop}(\_, val)$ in $\rho$ that is removed.

then $\rho$ is legal iff it satisfies conditions (C1) and (C2) from Definition 1, as well as

(C3) If $\rho$ is a legal sequence, then $\rho \cdot \text{Pop}(\_, val), val \neq \bot$ is legal iff

  - there is a $\text{Push}(val, \_)$ in $\rho$,
  - every Push succeeding $\text{Push}(val, \_)$ is matched, and
  - $val$ is not removed in $\rho$.

$\rho \cdot \text{Pop}(\_, \bot)$ is legal iff every Push in $\rho$ is matched.

### 2.2 System Model

The distributed system model we adopt is defined in [11], which we re-state in this section. We consider a set $\Pi = \{p_0, \cdots, p_{n-1}\}$ of $n$ processes, each modeled as a state machine
whose transitions are triggered by occurrences of three kinds of events: the invocation of an operation instance, the receipt of a message, and the expiration of a timer.

A *step* of a process is a 6-tuple \((s, T, C, M, R, s')\), where \(s\) and \(s'\) are respectively the old and new states, \(T\) is a trigger event, \(C\) is a local clock value taken from the reals, \(M\) is the set of messages sent, and \(R\) is either empty or the response of an operation instance, such that \(M, R,\) and \(s'\) are the result of the transition function operation on \(s, T,\) and \(C\).

A *view* of a process is a sequence of steps such that

- the old state of the first step is an initial state;
- the old state of each subsequent step is the same as the new state of the step following that one;
- the value of each timer in the old state of each step is at most the clock time of the step;
- if the trigger for a step is a timer going off, then there exists a timer in the old state of the step whose value equals the clock time of the step;
- the clock times in steps are increasing, and if the sequence is infinite then they are unbounded;
- at most one operation instance is pending at a time.

A *timed view* is a view in which a real number, or the *real time*, is associated with each step. The real time must proceed at the same rate as the clock time; i.e., there must exist a real number \(c\), called the *clock offset*, such that for every step, the difference between the clock time and the real time is exactly \(c\).

A *run* is a set of \(n\) timed views, one for each process, such that every message receipt has exactly one matching message send, and every message send has at most one message receipt. A run is *complete* if

- every message sent is received; and
• each timed view is either infinite or ends in a state in which no timers are set.

A run is admissible with respect to parameters $\epsilon$, $d$, and $u$ if

• the local clocks are synchronized to within $\epsilon$ time; i.e., if $c_i$ is the local clock offset for process $p_i$, then $|c_i - c_j| \leq \epsilon$ for all processes $p_i$ and $p_j$, and

• the delays for received messages lie within the range $[d - u, d]$, and if time $t$ has the send of a message with no matching receive, then the last real time for any step by the intended recipient process must be less than $t + d$.

Implicit in these requirements is that $\epsilon < u < d$, and in fact, we know that $\epsilon < (1 - 1/n)u$ [12]. We further assume that any message from a process to itself is simulated as taking the minimum message delay $d - u$.

We will consider only algorithms which are eventually quiescent – that every complete, admissible run with a finite number of operation instances is finite, implying every view is finite.

2.3 Correctness Condition

Our objective is to provide linearizable implementations of data types described in Section 2.1 that have their basis in the message-passing model described in Section 2.2. These algorithms must satisfy the following conditions:

• Liveness: In every complete, admissible run, the matchings between operation invocations and responses exhibit a one-to-one correspondence. This allows matching invocations and responses to be paired up as operation instances.

• Linearizability: For every complete, admissible run $R$, there is a permutation $\pi$ of the set of operation instances in $R$ such that (i) $\pi$ is legal and (ii) with respect to real time in $R$, if operation instance $op_1$ responds before operation instance $op_2$ is invoked, then $op_1$ precedes $op_2$ in $\pi$. $\pi$ is called a linearization of $R$. 
2.4 Performance Metrics

Lastly, we give two upper bound metrics on the performance of data types in this model.

The **worst-case time complexity** of operation $OP$ is the maximum over every instance of $OP$ that may occur in any complete, admissible run of the real time that elapses between the invocation of the instance and its response.

The **amortized time complexity** of operation $OP$ is the least upper bound over every complete, admissible run $R$ and every real time $t$ of $\text{avg\_time}(R, OP, t)$, where $\text{avg\_time}(R, OP, t)$ is the sum of the elapsed time taken by all instances of $OP$ in $R$ which complete by time $t$ divided by the number of such instances of $OP$. 
CHAPTER 3
ALGORITHM AND UPPER BOUNDS

In this paper, we contribute implementations of several kinds of relaxed stacks (Out-of-Order in this section, with the remaining deferred to the Appendix) according to the specified model. Each process instantiates a local copy of the relaxed data structure that is referred to as its local stack, and synchronization of these local stacks is a cornerstone of the reasoning involved. The algorithms are interrupt-driven, with three types of events that may trigger actions at a process: the invocation of an operation instance, the receipt of a message from another process, and the expiration of a earlier-set timer. The timers ensure that event occurrences are coordinated despite the delays in communication between processes, and each is associated with an operation instance and an action which are used by the algorithm to determine the return value of the operation instance.

In the algorithms, the PUSH or POP event handler takes place in a process $p_i$ when Push or Pop instances respectively are invoked in $p_i$, and a return statement refers to the response accompanied by a return value (or ACK, if there is no return value) of the previous operation instance invoked in the same process. Each operation instance causes the EXECUTELOCALLY function to be executed at all processes a given amount time after the operation instance is invoked, and in each process this is referred to as the instance’s execution or local execution.

We use timestamps to determine the order in which operation instances should be executed that satisfies the linearizability condition. Each operation instance upon invocation is given a timestamp, an ordered pair consisting of the local clock time and the id of the process in which the instance was invoked. We maintain at each process a priority queue of operation instances waiting for the correct time to execute locally, the priority function of which is the lowest-first lexicographic ordering of timestamps. If the timer for execut-
ing locally expires earlier in some operation instance with a later timestamp compared to another, then the operation with the earlier timestamp is allowed to execute locally first, canceling its own timer.

We assume that all built-in functions used in the pseudocode, such as \( \min \), are deterministic, i.e., there is some method of tie-breaking that guarantees a unique output for every input. In this way, different processes that execute the same pseudocode will get the same result.

### 3.0.1 Parameters

The parameters we use which are universal to the rest of this paper include:

- \( n \geq 1 \), the number of processes;
- \( k \geq 0 \), the relaxation parameter;
- \( l := \lfloor k/n \rfloor \);
- \( l_s := \lceil k/n \rceil - 1 \); and
- \( \tau \in [1, l] \), the trade-off parameter (we briefly introduce it here by giving the guideline that a higher value of \( \tau \) favors the performance of \( Pushes \) over \( Pops \));

all of which are integer-valued.

### 3.0.2 Local Variables

The algorithms use a number of local variables, which are specified completely here (it is recommended that this subsection be referred back to for reference):

- \( localTime \): Current local time
- \( lStack \): Set representation of local stack, initially empty. Elements have the following associated fields:
- `label`, which takes on process ids, `dummy`, and `null` (signifying that the element is unlabeled) as values and is initially `null`

- `popped`, a Boolean field which is initially `false`

- `t_volatile`, a time which is initially $-\infty$

- `higher`, which takes on elements and `null` as values and is initially `null`

- `lower`, which takes on elements and `null` as values and is initially `null`

- `canPopByLabel`, a Boolean field which is initially `true`

- `unresolved`, a Boolean field which is initially `false`

- `t_popped`, a time which is initially $\infty$; used in Algorithm 6 only

Operations on `lStack` include:

- `push(val)`: inserts a new element `val` at the top

- `pop()`: removes and returns the top non-popped element

- `peek()`: returns the top non-popped element

- `popByLabel(p_j)`: removes and returns the topmost non-popped, `canPopByLabel` element labeled $p_j$, or returns $\bot$ if none exists

- `peekByLabel(p_j)`: returns the topmost non-popped, `canPopByLabel` element labeled $p_j$, or $\bot$ if none exists

- `peekByLabel(p_j, S)`: returns the topmost non-popped, `canPopByLabel` element labeled $p_j$ which is also in the set $S$, or $\bot$ if none exists

- `topAvailableByLabel(p_j)`: returns the topmost non-popped element labeled $p_j$, or $\bot$ if none exists

- `peekBySet(S)`: returns the topmost element which is also in the set $S$, or $\bot$ if none exists

- `getFromTop(m)`: returns, without removing, the $m$th-topmost element, or $\bot$ if `size() < m` or $m \leq 0$

- `setLabel(p_j)`: returns the set of all non-popped elements labeled $p_j`
- `topSet(m)`: returns the set of the \(\min\{m, lStack.size()\}\) topmost elements in `lStack`
- `topSetBySet(m, S)`: returns the set of the \(\min\{m, lStack.size(), |S|\}\) topmost elements in `lStack` which are also in the set `S`
- `contains(val)`: returns `true` if `val` is an element in `lStack`, `false` otherwise
- `size()`: returns current number of elements
- `labeledSize()`: returns current number of labeled elements
- `remove(val)`: removes the element sharing the same value as `val`
- `label(pj, val)`: sets the label of the element `val` to `pj` and returns the process id number of its previous label if `val \neq \bot`

- **Pending**: Priority queue containing operation instances keyed by timestamp; initially empty; supports standard operations `insert(val, ts), min(), extractMin()`
- **localFastPushesActive**: Integer-valued variable, initially 0
- **popsInPending**: Integer-valued variable, initially 0
- **reservedQueues[]**: Array of size `n` of queues of data elements, each initially empty; support standard operations `enqueue(val), dequeue() and peek()` (the former of which does nothing if `val = \bot`, and the latter two of which return `\bot` if it is empty)
- **localPopsInvoked**: Integer-valued variable, initially 0
- **localPopsExecuted**: Integer-valued variable, initially 0
- **kMinusLateness**: Integer-valued variable, initially `k`

Here we define modifications of \(\tau\) which depend on the size of the local stack.

**Definition 8.** At each process `pi`, the packing-\(\tau\), `\tau_p`, and the covering-\(\tau\), `\tau_c`, are defined respectively as

\[
\tau_p := \min \left\{ \tau, \max \left\{ \left\lfloor \frac{lStack.size() - k}{n} \right\rfloor + \tau, 0 \right\} \right\}
\]
\[ \tau_c := \max \left\{ \tau, \left\lceil \frac{k - lStack.size()}{n} \right\rceil \right\}. \]

Definition 8 says that \( \tau_p \leq \tau \), with equality if and only if the local stack is of size \( k \) or greater; otherwise, it gives approximately the difference between the size of the local stack and \( k - \tau n \), divided by the number of processes \( n \).

Likewise, \( \tau_c \geq \tau \), with equality if and only if the local stack is of size \( k - \tau n \) or greater; otherwise, it gives approximately the difference between \( k \) and the size of the local stack, divided by \( n \).

In the algorithms given throughout the rest of this section, it is assumed that \( k \geq n \). All three of them give improved amortized performance over algorithms for unrelaxed Stacks, increasing as \( k \) increases by multiples of \( n \).

### 3.1 Out-of-Order Relaxed Stacks

Algorithm 1 gives a minimal implementation of the Out-of-Order \( k \)-Relaxed Stack (Definition 2) for the purpose of understanding, whereas Algorithm 2 gives an improved implementation which will be the one studied exhaustively for correctness and performance.

#### 3.1.1 Minimal Implementation

The initial motivating idea is that in each process’ local stack, all elements legal to return by locally executing \textit{Pops} are \textit{labeled} with process ids, signifying that the element may only be returned by that process. This idea allows for elements labeled with the id of a process at the invocation of a \textit{Pop} in that process to be \textit{claimed} by that \textit{Pop}, meaning that the \textit{Pop} returns that element’s value in the earliest time possible, \( \epsilon \) after invocation. Such a \textit{Pop} is called a \textit{fast Pop}, an operation abbreviated as \textit{pop.f} in the algorithm (This naturally leads to any other \textit{Pop} being called a \textit{slow Pop}, abbreviated \textit{pop.s}).
If all elements labeled with the id in the local stack are already fast \textit{Popped} at the time another \textit{Pop} is invoked by a process, then that \textit{Pop} is a slow \textit{Pop} and waits until another element is labeled with the required process id, which we guarantee occurs before the \textit{Pop} executes locally in the same process.

When the size of each local stack is large enough, a \textit{uniformity invariant} ensures that a certain number of elements are labeled with each process id and can be claimed by fast \textit{Pops}, which leads to an amortized performance improvement for this relaxed stack, contrasting with unrelaxed stacks.

A sample run involving processes \( p_i \), \( p_j \), and \( p_h \) is depicted in the figure on the following page, where \( \rightarrow \) is the forward direction in time. As we can see, the invocation of new operation instances is permitted in a process as long as all previous ones have responded. (Omitted from the figure for the sake of understanding are the clock offsets between processes, as well as the fact that after each message is received, the operation instance sits in \textit{Pending} for a specified amount of time before executing locally.)

The top \( k \) elements are always labeled in each local stack. It is important to note that not all labeled elements are necessarily legal to return by fast \textit{Pops}. We discuss the restrictions on elements that can be returned by fast \textit{Pops} in the context of active operation instances.
Figure 3.2: Invocation ($I$), response ($R$), and local execution ($E$) of $Pop$ instances.

**Definition 9.** An operation instance is active in a process $p_i$ if it has been invoked (in terms of real time) but not yet executed locally in $p_i$.

**Definition 10.** Two operation instances mutually active if, in the process in which the one with the later timestamp is invoked, the one with the earlier timestamp has not yet completed local execution by the other’s invocation time.

One example scenario is that the $k$th-topmost element in the local stack is illegal to return by a $Pop$ if there is a mutually active $Push$ and no mutually active $Pops$, because since the $Push$ has an earlier timestamp, the number of unmatched $Pushes$ preceding the $Pop$ in the linearized order of operation instances is here not less than $k$.

Furthermore, there may be more than a single mutually active $Push$, as the maximum number among all processes is $n$. This means that at the invocation of a $Pop$, which we call $pop$, in process $p_i$ when the local stack has at least $k - n$ elements, there might be up to $n$ $Pushes$ mutually active with $pop$ that precede $pop$ in the linearization. We conclude that only $k - n$ elements in $p_i$’s local stack are guaranteed legal to return by $pop$ (if they have not already been returned by other fast $Pops$).

A silver lining is that other $Pops$ mutually active with $pop$ that precede $pop$ in the linearization increase the number of elements legal to return by $pop$, because they decrease the number of unmatched $Pushes$ in the sequence. When a process receives the message sent
at most $d$ time after the real time of invocation of a $Pop$, it increments its $popsInPending$ counter. As expected, each $Pop$ decreases the $popsInPending$ counter after it executes locally. Now we have that at least the topmost $k - n + popsInPending$ elements in $p_i$’s local stack at $pop$’s invocation are legal for $pop$ to return, if they have not already been returned by other fast $Pops$.

Here we introduce the notion of the safe region, which is a set of $\min\{k - n, lStack.size()\}$ elements in the local stack legal for fast $Pops$ to return. The set of labeled elements outside the safe region is referred to as the contention region. The uniformity invariant introduced earlier applies to the safe region, and intuitively describes the even distribution of labels among elements in the safe region if it is of size $k - n$. Preserving the uniformity invariant at all processes is one reason why a fast $Pop$ marks an element popped instead of directly removing it from the local stack during invocation (the other motive being that the deterministically-chosen new label at line 47 is the same in the local execution at every process). Recall that $peekByLabel()$ and $popByLabel()$ return only elements that are not popped.

![Figure 3.3: The safe and contention regions.](image)

Maintaining the uniformity invariant requires that whenever the local execution of a $Pop$ in $p_i$ removes an element from the safe region and an element previously not in the safe
region enters it, it must relabel some element in the safe region. By default, the relabeled element is the newly entering element, which is also the bottommost in the updated safe region. However, one issue this causes is that the element could have already been returned by another Pop invoked in \( p_i \), which we call pop, so it is dangerous to relabel it \( p_j \), as the element is not popped in the view of \( p_j \) and could be returned again by a fast Pop invoked in \( p_j \), causing an illegal linearization of operation instances since pop already matches the Push that added this element.

To decide which element should be relabeled, we give the following procedure. The safe region is initially the topmost \( k - n \) elements in the local stack. Any new element added by the local execution of a Push in any process \( p_h \) is temporarily marked volatile, meaning it can be relabeled, for as long as required by a Pop invoked (in any process) before the local execution of the Push (in that process) to itself locally execute in \( p_h \), removing its returned element from \( p_h \)’s local stack. pop from the previous paragraph is such a Pop, because the only way for an element in the initial safe region to leave the topmost \( k - n \) elements is through locally executing Pushes. Thus, if another locally executing Pop intervenes by removing some element in the safe region from the local stack, it relabels the topmost volatile element instead of the element returned by pop.

It is illegal to return volatile elements that are about to be relabeled. We know whether they will be by Pops that were invoked at least \( d \) time ago, because they contribute to the \( \text{popsInPending} \) counter; therefore, the topmost \( \text{popsInPending} \) volatile elements in the local stack are excluded from the safe region. We also guarantee that any volatile elements outside the topmost \( \text{popsInPending} \) are legal to return by an invoking Pop, because this Pop sends a message to all processes to mark its claimed element non-volatile within \( d \) time. Any Pops not contributing to \( \text{popsInPending} \) will not execute locally in less than \( d \) time, because in this algorithm operation instances execute locally in at least \( 2d \) time.

We add that whenever a locally executing Push adds a new element to the local stack, which is then marked volatile, it is labeled with the same label as the element that was
moved out of the topmost $k - n$ in the local stack. These two elements form a construct called a \((\text{higher}, \text{lower})\)-pair, which offers a number of guarantees. First of all, the topmost \textit{popsInPending} elements that are the \textit{lower} in some \((\text{higher}, \text{lower})\)-pair are included in the safe region, which allows the size of the safe region to stay at size $k - n$. We show that in this way, the local executions of \textit{Pushes} also preserve the uniformity invariant.

At this point, we are able to state the \textit{labeling invariant} that the set of labeled elements is not only the elements legal for a locally executing \textit{Pop} to return, but also elements which will become legal to return when all \textit{Pops} contributing to \textit{popsInPending} are executed locally. We observe that if any elements are labeled, say a number $m$, then they must be the top $m$ elements in the local stack.

The algorithm we have described so far is a complete implementation the Out-of-Order $k$-Relaxed Stack, and we give this implementation in Algorithm 1.

\begin{algorithm}
\textbf{Algorithm 1} A minimal version of the pseudocode for each process $p_i$ implementing an Out-of-Order $k$-Relaxed Stack, where $k \geq n$.
\begin{center}
\begin{tabular}{l}
1: \textbf{HandleEvent} \textit{PUSH}(val) \\
2: \hspace{1em} send \((\text{push}_s, \text{val}, (\text{localTime}, i))\) to all \\
3: \textbf{HandleEvent} \textit{POP} \\
4: \hspace{1em} \textbf{if} \ lStack.peekByLabel(p_i, \text{safeRegion}()) \neq \perp \textbf{or} \ lStack.size() < k - n \textbf{then} \\
5: \hspace{2em} \textbf{let} \ ret := lStack.peekByLabel(p_i, \text{safeRegion}()) \\
6: \hspace{2em} \text{ret.popped} := \text{true} \\
7: \hspace{2em} \text{send} \((\text{pop}_f, \text{ret}, (\text{localTime}, i))\) to all \\
8: \hspace{2em} \text{setTimer}(\epsilon, (\text{pop}_f, \text{ret}, \text{null}), \text{respond}) \\
9: \hspace{2em} \textbf{else} \text{send} \((\text{pop}_s, \text{null}, (\text{localTime}, i))\) to all \\
10: \textbf{HandleEvent} \textit{RECEIVE} \((\text{op}, \text{val}, \text{ts})\) \text{FROM} \ p_j \\
11: \hspace{1em} \text{Pending.insert}((\text{op}, \text{val}, \text{ts})) \\
12: \hspace{1em} \text{setTimer}(d + u, (\text{op}, \text{val}, \text{ts}), \text{execute}) \\
13: \hspace{1em} \textbf{if} \ \text{op} == \text{pop}_f \text{or} \ \text{op} == \text{pop}_s \textbf{then} \\
14: \hspace{2em} \text{popsInPending}++ \\
15: \hspace{2em} \textbf{if} \ \text{op} == \text{pop}_f \textbf{then} \\
16: \hspace{2em} \text{resetVolatility}(\text{val}) \\
17: \hspace{2em} \textbf{HandleEvent} \textit{EXPIRE TIMER}((\text{op}, \text{val}, (\ast, j)), \text{respond}) \\
18: \hspace{2em} \textbf{if} \ \text{op} == \text{pop}_f \textbf{then return} \text{val} \\
19: \hspace{2em} \text{else return ACK}
\end{tabular}
\end{center}
\end{algorithm}
20: \textbf{HandleEvent} \texttt{EXPIRE\_TIMER((op, val, ts), execute)}
21: \hspace{1em} \textbf{while} \( ts \geq \text{Pending.min()} \) \textbf{do}
22: \hspace{2em} \langle op', val', ts' \rangle := \text{Pending.extractMin()}
23: \hspace{2em} execute\_Locally(op', val', ts')
24: \hspace{2em} cancel\_Timer((op', val', ts'), execute)
25: \textbf{function} EXECUTE\_LOCALLY(op, val, (time, j))
26: \hspace{1em} \textbf{if} op == \textit{push\_f} or op == \textit{push\_s} \textbf{then}
27: \hspace{2em} lStack.push(val)
28: \hspace{2em} \textbf{if} lStack.size() \leq k - n \textbf{then}
29: \hspace{3em} lStack.label(\text{argmin}_{p_h} |lStack.setByLabel(p_h) \cap \text{safeRegion()}|, val)
30: \hspace{2em} \textbf{else}
31: \hspace{3em} lStack.label(lStack.getFromTop(k - n + 1).label, val)
32: \hspace{3em} set\_Volatility(val, lStack.getFromTop(k - n + 1))
33: \hspace{3em} lStack.label(null, lStack.getFromTop(k + \min\{\text{lowerRegion()}, \text{popsInPending}\} + 1))
34: \hspace{2em} \textbf{else}
35: \hspace{3em} \textbf{if} op == \textit{pop\_f} \textbf{then} let toRemove := val
36: \hspace{3em} \textbf{else}
37: \hspace{4em} let toRemove := lStack.peekByLabel(p_i)
38: \hspace{3em} \textbf{return} toRemove
39: \hspace{3em} \textbf{if} toRemove.higher \neq \text{null} \textbf{then} set\_Volatility(toRemove.higher)
40: \hspace{3em} \text{popsInPending} --
41: \hspace{3em} \textbf{if} toRemove \in \text{safeRegion()} \textbf{then}
42: \hspace{4em} \textbf{if} volatile\_Region() \cap lStack.topSet(k - n) \neq \emptyset \textbf{then}
43: \hspace{5em} let toRelabel := lStack.peekBySet(volatile\_Region())
44: \hspace{5em} reset\_Volatility(toRelabel)
45: \hspace{4em} \textbf{else} let toRelabel := lStack.getFromTop(k - n + 1)
46: \hspace{5em} lStack.label(j, toRelabel)
47: \hspace{4em} lStack.remove(toRemove)
48: \hspace{4em} \textbf{if} lStack.labeledSize() < k \textbf{then} lStack.label(dummy, lStack.peekByLabel(null))
49: \textbf{function} SAFE\_REGION
50: \hspace{1em} \textbf{return} [topSet(k - n) \cup topSetBySet(popsInPending, lowerRegion())] \setminus
51: \hspace{1em} topSetBySet(popsInPending, volatile\_Region())
52: \textbf{function} VOLATILE\_REGION
53: \hspace{1em} \textbf{return} \{elem \in lStack.topSet(k - n) : elem.t_{volatile} > local\_Time - (2d + 2u + \epsilon)\}
54: \textbf{function} LOWER\_REGION
55: \hspace{1em} \textbf{return} \{elem.lower : elem \in volatile\_Region()\}
56: \textbf{function} SET\_VOLATILITY(elem, other)
3.1.2 Improvements

Instead of allowing only a single active Push at a time per process, we may permit up to $\tau$, so that the maximum number of mutually active Pushes among all the processes is $\tau n$. In the same spirit as before, Pushes deciding at invocation to respond in $\epsilon$ time are referred to as fast Pushes (abbreviated push $f$), and other Pushes are referred to as slow Pushes (abbreviated push $s$).

The trade-off inherent in the $\tau$ parameter is that the size of the safe region is now $k - \tau n$, and all occurrences of $k - n$ in Algorithm 1 are now replaced by $k - \tau n$ in Algorithm 2. Hence, another interpretation of $\tau$ is the size of the contention region divided by $n$ when $k$ elements are labeled.

We generalize our usage of $\tau$ to $\tau_c$ and $\tau_p$ as follows. For a local stack of any size, not just $k - \tau n$ or greater, the maximum number of mutually active Pushes is actually permitted to be $\tau_c n$. $\tau_p$ gives the size of the contention region divided by the number of processes when fewer than $k$ elements are labeled.

Lastly, we describe an optimization on operation response time. Given that the safe region is of size $k - \tau n$, the idea is that if a series of $l - \tau$ fast Pops is invoked and respond in $p_i$ as quickly as possible in a process before a slow Pop is invoked, then the slow Pop only needs to wait until the first fast Pop has executed locally, labeling another element in the local stack $p_i$, before the slow Pop may return the element newly labeled $p_i$. We use an array of reserved queues, one dedicated to Pops invoked by each process, whose purpose
is to guarantee that the element returned by a Pop is the same as the element removed from the local stack by its local execution in each process. If the size of the safe region is less than \( k - \tau n \), it is possible that fewer than \( l - \tau \) elements in the safe region are labeled \( p_i \), but it is in this case legal by the uniformity invariant for fast Pops to return \( \perp \).

A similar optimization is made for slow Pushes, this time accomplished by limiting the number of active fast Pushes in a process to \( \tau_c - 1 \). We dub this overall optimization technique “short-circuiting,” with the slow Pushes and Pops referred to as “short-circuited”.

\[
\begin{array}{c}
I(\text{pop}_f^1)R(\text{pop}_f^1) I(\text{pop}_s) E(\text{pop}_f^1) E(\text{pop}_s) \\
I(\text{pop}_f^2)R(\text{pop}_f^2) E(\text{pop}_f^2) \uparrow
\end{array}
\]

Figure 3.4: (a) Non-short-circuited and (b) short-circuited Pops, with \( l - \tau = 2 \). In each case, \( \uparrow \) indicates the earliest time another Pop instance can be invoked.

This final implementation of the Out-of-Order \( k \)-Relaxed Stack is given in Algorithm 2, and is also the version that we will analyze throughout the remainder of this section.

**Algorithm 2** Pseudocode for each process \( p_i \) implementing an Out-of-Order \( k \)-Relaxed Stack, where \( k \geq n \).

1: **HandleEvent** Push(val)
2:  if localFastPushesActive < \( \tau_c - 1 \) then
3:    send \( (\text{push}_f, \text{val}, \langle \text{localTime}, i \rangle) \) to all
4:    setTimer(\( \epsilon \), \( \langle \text{push}_f, \text{null}, \langle \text{localTime}, i \rangle \rangle \), respond)
5:  localFastPushesActive++
6:  else
7:    send \( (\text{push}_s, \text{val}, \langle \text{localTime}, i \rangle) \) to all
8:    setTimer(\( \epsilon \), \( \langle \text{push}_s, 1, \langle \text{localTime}, i \rangle \rangle \), respond)
9:  **HandleEvent** Pop
if \( lStack.peekByLabel(p_i, safeRegion()) \neq \perp \) or \( lStack.size() < k - \tau_n \) then

let ret := lStack.peekByLabel(p_i, safeRegion())
ret.popped := true
send (pop_f, ret, ⟨localTime, i⟩) to all
setTimer(\( \epsilon \), ⟨pop_f, ret, null⟩, respond)

else
send (pop_s, null, ⟨localTime, i⟩) to all
setTimer(\( \epsilon \), ⟨pop_s, 1, ⟨localTime, i⟩⟩, respond)

HandleEvent RECEIVE (op, val, ts) FROM p_j
Pending.insert((op, val, ts))
setTimer(\( d + u \), (op, val, ts), execute)

if op == pop_f or op == pop_s then
popsInPending++

if op == pop_f then
resetVolatility(val)

HandleEvent EXPIRE TIMER ((op, val, ⟨∗, j⟩), respond)
if op == pop_f then return val
else if op == pop_s then
flushReservedQueueFront(j)

if reservedQueues[j].peek \neq \perp then
reservedQueues[j].peek().popped := true
return reservedQueues[j].peek()

else if val \cdot \epsilon - [2d - (1 - \tau)\epsilon] < \epsilon then
setTimer(min{\( \epsilon \), 2d - (1 - \tau)\epsilon - val \cdot \epsilon}, (pop_s, val + 1, ⟨∗, j⟩), respond)
else return \perp

else if op == push_s and localFastPushesActive == \( \tau_c - 1 \) then
setTimer(min{\( \epsilon \), 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon}, (push_s, val + 1, ⟨∗, j⟩), respond)
else return ACK

HandleEvent EXPIRE TIMER ((op, val, ts), execute)
while ts \geq Pending.min() do
⟨op', val', ts'⟩ := Pending.extractMin()
executeLocally(op', val', ts')
cancelTimer((op', val', ts'), execute)

function ExecuteLocally(op, val, ⟨time, j⟩)
if op == push_f or op == push_s then
lStack.push(val)
if lStack.size() \leq k - \tau_n then
lStack.label(argmin_{p_k} lStack.setByLabel(p_k) \cap safeRegion(), val)
else

lStack.label(lStack.getFromTop(k - τn + 1).label, val)

setVolatility(val, lStack.getFromTop(k - τn + 1))

lStack.label(null, lStack.getFromTop(k +

   min{|lowerRegion()|, popsInPending} + 1))

if j == i and op == push then localFastPushesActive--

else

if j ≠ i then flushReservedQueueFront(j)

if op == pop then let toRemove := val

else let toRemove := reservedQueues[j].dequeue()

if toRemove.higher ≠ null then resetVolatility(toRemove.higher)

popsInPending--

if toRemove ∈ safeRegion() then

   if volatileRegion() ∩ lStack.topSet(k - τn) ≠ Ø then

      let toRelabel := lStack.peekBySet(volatileRegion())

      resetVolatility(toRelabel)

   else let toRelabel := lStack.getFromTop(k - τn + 1)

   lStack.label(j, toRelabel)

   reservedQueues[j].enqueue(toRelabel)

   lStack.remove(toRemove)

   if lStack.labeledSize() < k then lStack.label(dummy, lStack.peekByLabel(null))

function SAFEREGION

return [topSet(k - τn) ∪ topSetBySet(popsInPending, lowerRegion())] \topSetBySet(popsInPending, volatileRegion())

function VOLATILEREGION

return {elem ∈ lStack.topSet(k - τn) : elem.tvolatile > localTime - (2d + 2u + ϵ)}

function LOWERREGION

return {elem.lower : elem ∈ volatileRegion()}

function SETVOLATILITY(elem, other)

elem.tvolatile := localTime

elem.lower := other

elem.lower.higher := elem

function RESETVOLATILITY(elem)

elem.tvolatile := −∞

elem.lower.higher := null

elem.lower := null

function FLUSHRESERVEDQUEUEFRONT(j)
83:   let front := reservedQueues[j].peek()
84:   while ¬lStack.contains(front) or front.popped or front.label ≠ p_j do
85:       reservedQueues[j].dequeue()
86:   let front := reservedQueues[j].peek()

3.1.3 Correctness

Let ts(op) denote the timestamp associated with an operation instance op given as the first argument in line 3, 7, 13, or 16.

Construction 1. Define the permutation π of operation instances in a complete, admissible run of Algorithm 2 as the order given by sorting by ts(op) for each operation instance op.

We define the pending delay, which is the amount of time that passes between the receipt of a message sent by the invocation of an operation instance and the local execution of that operation instance. In Algorithm 2, the pending delay is d + u (line 19).

The following lemmas show that Construction 1 respects the real-time partial order of operation instances, following a similar method as in [11].

Lemma 1. If the pending delay of an algorithm is at least u + ϵ, then each process running the algorithm locally executes all Pushes and Pops in timestamp order.

Proof. Let p be the pending delay, which is at least u + ϵ.

Each operation instance op_1 upon invocation sends a message to all processes, each of which receives this message within d − u to d time. Each receiving process, including the sender, then inserts the operation instance into its local Pending priority queue and sets a timer to the pending delay p. The process executes op_1 if the timer expires, so each operation instance is locally executed at each process no later than d + p = d + u + ϵ real time after it is invoked.

op_1 may be executed earlier than the expiration time of its timer, however, if the execute timer for another instance, op_2, with ts(op_2) > ts(op_1) expires sooner. But the earliest time
the timer may expire is \((d - u) + p = d + \epsilon\) real time after \(op_2\) is invoked, and since the maximum message delay is \(d\) and \(op_1\) could have been invoked no later than \(\epsilon\) real time after \(op_2\), the sum of the delays is \(d + \epsilon\), the latest in real time after \(op_1\)'s invocation that \(op_1\) could have been inserted into \(Pending\). Since \(ts(op_1) < ts(op_2)\) by assumption, \(op_1\) is in \(Pending\) before the timer for \(op_2\) expires, so it is executed before \(op_2\).

**Lemma 2.** Each process running Algorithm 2 locally executes all Pushes and Pops in timestamp order.

*Proof.* Since \(d + u > u + \epsilon\), Lemma 1 holds for Algorithm 2.

**Lemma 3.** \(\pi\) respects the real-time order of non-overlapping operation instances.

*Proof.* If we have two instances \(op_1\) and \(op_2\) with \(ts(op_1) < ts(op_2)\), then \(op_1\) responds before \(op_2\) does, because the time that passes between invocation and response is always at least \(\epsilon\) for all operation instances in Algorithm 2, while the difference between the local clocks of the invoking processes is at most \(\epsilon\).

We proceed to introduce a number of new definitions and establish essential properties of Algorithm 2 which will appear repeatedly and remain useful throughout this paper.

**Lemma 4.** There are at most \(\tau_c\) mutually active Pushes occurring in a single process and at most \(\tau_c n\) mutually active Pushes occurring over all processes at any given time.

*Proof.* In a process \(p_i\), since each fast Push increments \(localFastPushesActive\) at invocation (line 5) and decrements it at local execution (line 5), \(localFastPushesActive\) counts the number of fast Pushes currently taking place in process \(p_i\). Because the number of mutually active fast Pushes at \(p_i\) is limited to \(\tau_c - 1\) (line 2), a \(\tau_c\)th mutually active Push must be slow, and we show in the proof of Theorem 15 in Section 3.1.4 that between the invocation and response of a slow Push, at least one fast Push has executed locally, so the invocation of the next Push increases the number of mutually active Pushes to at most \(\tau_c\). Since at most \(\tau_c\) Pushes in total may be mutually active in \(p_i\), we conclude that among all processes, there can be up to \(\tau_c n\) mutually active Pushes.


**Definition 11.** Elements are safe if they are legal to return by Pops that are at any point between invocation and local execution, given that they are not already returned.

**Lemma 5.** The safeRegion() function given in Algorithm 2 returns a set of elements that are guaranteed to be safe. We also refer to this set as the safe region.

**Proof.** We can deduce from Lemma 4 that $\tau n$ is an upper bound for the number of active Pushes in each process, while $popsInPending$ is a lower bound for the number of active Pops since for each Pop, it is incremented once after invocation (line 22) and decremented once at local execution (line 58). Then $k - \tau n + popsInPending$ is a lower bound for the number of topmost elements that are legal to return. We can see that safeRegion() is contained in this set of elements. \qed

**Definition 12.** The uniformity invariant states that

$$|\text{lStack.setByLabel}(p_i) \cap \text{safeRegion}()| \leq \left\lfloor \frac{k}{n} \right\rfloor - \tau \quad \forall i \in [0, n - 1]$$

with the number of $i$ such that the bound is reached being at most $n_a := k - (\lceil k/n \rceil - 1)n$, and when $\text{lStack.size}() \geq k - \tau n$,

$$|\text{lStack.setByLabel}(p_i) \cap \text{safeRegion}()| - |\text{lStack.setByLabel}(p_j) \cap \text{safeRegion}()| \leq 1$$

$\forall i, j \in [0, n - 1]$.

**Definition 13.** Any element in the local stack is volatile if it is contained in volatileRegion(); in other words, its $t_{\text{volatile}}$ field was set less than $2d + 2u + \epsilon$ time before the current local time (line 91).

**Definition 14.** If the size of the local stack is at least $k$ and a Push executes locally, then as long as the new top element higher remains volatile, higher and the new $(k - \tau n + 1)$-th-topmost element lower form a (higher, lower)-pair.
The volatile region is the set of volatile elements, or alternately the set of all elements that are the higher element in some \((\text{higher, lower})\)-pair, in the topmost \(k - \tau n\) elements. The lower region is the set of all elements that are the lower element in some \((\text{higher, lower})\)-pair where higher is in the volatile region.

Lemma 6. If an element \(\text{elem}\) is the lower element in some \((\text{higher, lower})\)-pair, then the value of \(\text{elem}.\text{higher}\) is the other element; if not, \(\text{elem}.\text{higher}\) is null. Likewise, if \(\text{elem}\) is the higher element in some \((\text{higher, lower})\)-pair, then the value of \(\text{elem}.\text{lower}\) is the other element, and if not, \(\text{elem}.\text{lower}\) is null.

Proof. \((\text{higher, lower})\)-pairs are created only by locally executing \text{Pushes} that increase the size of the local stack to greater than \(k - \tau n\) (line 50 is the only place where elements are made volatile). The lower element in the new \((\text{higher, lower})\)-pair is some element that already exists in the local stack. Then lines 76 and 77 establish that \(\text{higher}.\text{lower} = \text{lower}\) and \(\text{lower}.\text{higher} = \text{higher}\).

Events that cause a \((\text{higher, lower})\)-pair to cease to exist by definition include the removal of either element or the loss of \(\text{higher}\)’s volatility. By definition, the \text{remove()}\ operation on the local stack sets \(\text{higher}.\text{lower}\) and \(\text{lower}.\text{higher}\) to null. Furthermore, we can see from line 78 that a generic function call \(\text{resetVolatility}(\text{higher})\) sets \(\text{higher}.\text{lower}\) and \(\text{lower}.\text{higher}\) to null and makes \(\text{higher}\) non-volatile at once.

Lastly, there will not be a volatile element \(\text{elem}\) with \(\text{elem}.\text{lower} = \text{null}\), because the only time the lower element of a \((\text{higher, lower})\)-pair can be removed is at the local execution of a \text{Pop}, at which \(\text{higher}\) is made non-volatile (line 57).

Definition 15. A set of elements in the local stack is connected or are contiguous if as long as it contains no element in the topmost \(m\) elements but not the topmost \(m - 1\) elements for any \(m > 1\), then it contains no element in the topmost \(m + 1\) elements but not the topmost \(m\) elements.

Lemma 7. The volatility invariant holds, and states that in the local stack of each process,
(i) any two elements that form a \((\text{higher}, \text{lower})\)-pair share the same label;

(ii) if an element \(\text{elem}\) is the topmost in the volatile region, then \(\text{elem}.\text{lower}\) is currently the \((k - \tau n + 1)\)st-topmost element, so that any element is the \(m\)th-topmost volatile element if and only if it is the higher of a \((\text{higher}, \text{lower})\)-pair in which lower is the \(m\)th-topmost element in \(\text{lowerRegion}\);

(iii) the lower region is mutually exclusive with the topmost \(k - \tau n\) elements and connected; and

(iv) member elements of a \((\text{higher}, \text{lower})\)-pair cannot mutually exist in the safe region.

\textbf{Proof.} The statements rely on the fact that \((\text{higher}, \text{lower})\)-pairs are created only by locally executing \textit{Pushes}.

(i) Whenever a \((\text{higher}, \text{lower})\)-pair is created, \textit{higher} is given the same label as \textit{lower} (line 49). Whenever any element is relabeled, if it is the \textit{higher} element of a \((\text{higher}, \text{lower})\)-pair, then it is made non-volatile (line 62), and if it the \textit{lower} element of a \((\text{higher}, \text{lower})\)-pair, then its \textit{higher} element is made non-volatile (line 57); the \((\text{higher}, \text{lower})\)-pair necessarily no longer exists in either case.

(ii) In a series of \textit{Pushes}, each newly added volatile element must be the new top element and hence the topmost volatile element in the local stack, and each new \((k - \tau n + 1)\)-st topmost element which is added to the lower region is also higher up than the previous \((k - \tau n + 1)\)-st element in the local stack. Any \textit{Pop} that makes a volatile element non-volatile must do so for the topmost volatile element, and also moves the previously \((k - \tau n + 1)\)st-topmost element to the \((k - \tau n)\)th place, so that the new \((k - \tau n + 1)\)st-topmost element is now the topmost member of the lower region if \((\text{higher}, \text{lower})\)-pairs still exist, thereby preserving the stated property.

(iii) An element is made the \textit{lower} element of a \((\text{higher}, \text{lower})\)-pair if and only if it becomes the \((k - \tau n + 1)\)st-topmost element due to a \textit{Push} (line 76), and subsequent \textit{Pushes} only move existing lower region elements farther from the topmost \(k - \tau n\)
while maintaining their adjacency since they are all moved down simultaneously in the local stack.

Only a $Pop$ may bring a lower region element $elem$, which must be the topmost, back into the topmost $k - \tau n$ elements, but such a $Pop$ would also make the topmost volatile element non-volatile (line 62), and by (ii) it is $elem.higher$, which implies that $elem$ is no longer the lower element in a $(higher, lower)$-pair. While $elem$ is no longer part of the lower region, the remaining elements in the lower region are still outside the topmost $k - \tau n$ and contiguous.

(iv) This follows from (ii) and the definition of $safeRegion()$ (line 69).

Corollary 8. The size of the safe region is always $\min\{k - \tau n, lStack.size()\}$.

Proof. From Lemma 6, we know that every volatile element $elem$ has $elem.lower \neq null$, so $volatileRegion()$ and $lowerRegion()$ are of the same size. In the definition of $safeRegion()$ (line 69), since the top $k - \tau n$ elements are mutually exclusive with the lower region by (iii) from Lemma 7, the size of their union is the sum of their separate sizes. Taking the union’s set difference with a set contained in the union of the same size as the lower region results in a set of size $k - \tau n$.

Corollary 9. Incrementing or decrementing $popsInPending$ preserves the uniformity invariant.

Proof. First, we consider incrementing. If there are no volatile elements in the safe region, then the safe region by its definition is left unchanged when $popsInPending$ is incremented. Otherwise, if there are volatile elements in the safe region, meaning they are not the topmost $popsInPending$ volatile elements, then incrementing $popsInPending$ moves an element of the volatile region out of the safe region and moves another element of the lower region into the safe region. From Lemma 7, (iv) requires that these elements form a
(higher, lower)-pair, and thus they share the same label by (i). The number of elements with each label is preserved.

Next, we consider decrementing. If all the volatile elements are in the safe region, then the safe region by its definition is left unchanged when \( \text{popsInPending} \) is decremented. Otherwise, if there are volatile elements outside the safe region, meaning they are among the topmost \( \text{popsInPending} \) volatile elements, then decrementing \( \text{popsInPending} \) moves an element of the volatile region into the safe region and moves another element of the volatile region out of the safe region. Then by the same reasoning as before, the uniformity invariant holds.

\[ \square \]

**Remark 1.** \( \text{popsInPending} \) is incremented by 1 (line 22) every time the process receives a message sent from any \( \text{Pop} \) and is decremented by 1 (line 58) every time any \( \text{Pop} \) executes locally, which means this variable counts the number of active \( \text{Pops} \) from which this process received a message.

**Definition 16.** The labeling invariant states that the labeled elements in the local stack are the topmost \( k + \min\{|\text{lowerRegion()}|, \text{popsInPending}\} \) elements.

The arguments required in the next lemma form the most subtle part of this proof of correctness and inspire the more complicated procedures of the algorithm in large part, making use of message passing.

**Lemma 10.** Any element returned by a \( \text{Pop} \) will not be relabeled with another process id.

**Proof.** We consider the process \( p_i \). Only \( \text{Pops} \) relabel elements with new process ids, and a \( \text{Pop} \) can only relabel an element \( \text{elem} \) if it is either volatile in the topmost \( k - \tau n \) elements (line 61) or, if there is no such element, the new \( (k - \tau n) \)th-topmost element in \( p_i \)’s local stack (line 63).

In the former case, the volatile element \( \text{elem} \) can only be returned by a \( \text{Pop} \), which we call \( \text{pop} \), invoked by \( p_i \) at time \( t_0 \) if \( \text{pop} \) is fast and the element is in the safe region, which from its definition means that it is not in the topmost \( \text{popsInPending} \) elements. Since
each Pop contributing to popsInPending relabels only the top volatile element (line 61) at the time of its local execution, it will not relabel elem. Also, receiving a message takes at most $d$ time, so any active Pop from which $p_i$ has not received a message by time $t_0$, and hence does not contribute to popsInPending, will not locally execute until after $t_0 + (2d - d) = t_0 + d$. Lastly, pop sends a message at time $t_0$ to all other processes to make elem non-volatile (line 24), which they receive by time $t_0 + d$. Thus, none of the active Pops have the risk of relabeling elem.

In the latter case where pop’s invocation claims an element elem to return which becomes the $(k - \tau n)$th-topmost element in $p_i$’s local stack resulting from another locally executing Pop we call pop$_2$, then pop$_2$ cannot have relabeled elem unless there were no volatile elements in the topmost $k - \tau n$ (line 60). pop must have claimed elem as an element in the safe region. By the assumption of pop$_2$’s existence, at least one of the following must be true: (i) elem was outside the topmost $k - \tau n$ elements at the time of pop’s invocation, which means that the topmost $k - \tau n$ elements had volatile elements and popsInPending $> 0$ by the definition of safeRegion(), or (ii) some locally executing Push moved elem outside the topmost $k - \tau n$ elements in $p_i$’s local stack between pop’s invocation and pop$_2$’s local execution in $p_i$.

Given that (i) is true, all the Pops contributing to popsInPending must execute locally within $d + u$ time from pop’s invocation since they were invoked at least $d$ real time ago, and the local executions remove a number equal to popsInPending of elements which moves elem into the topmost $k - \tau n$ elements, unless another Push is locally executed, the case in which is handled in (ii).

Consider when (ii) is true. We know that at each process, pop may locally execute up to $2d + u$ real time after invocation, because it can take up to $d$ time for pop’s message to arrive at a process, and the execute timer is set to $d + u$. The local execution of the Push, which succeeds pop’s invocation in $p_i$ but which may happen $u + \epsilon$ time earlier in real time in other processes, makes the newly added element volatile for $2d + 2u + \epsilon$ time in each
process, so the element is still volatile in each process by the time pop executes locally. Then this element is relabeled instead of elem.

The reservedQueues[] local variable helps guarantee that the element returned by \( p_i \) for the slow Pop will be the same element that is removed in every process’ local stack upon the slow Pop’s local execution, so that every process maintains the same local state. We state this in the following lemma.

**Lemma 11.** If slow Pops invoked by \( p_i \) respond before local execution, then any element returned by a slow Pop will be the same element that is removed from every process’ local stack during local execution.

**Proof.** Let \( \text{pop} \) denote a slow Pop. If we show that calling \( \text{flushReservedQueueFront}(i) \) in \( p_i \) by \( \text{pop} \)'s response (line 28) and in all other processes by \( \text{pop} \)'s local execution (line 54) removes the same elements from \( \text{reservedQueues}[i] \) both in \( p_i \) and in every other process, then we guarantee that \( \text{reservedQueues}[i].peek() \) at the response (line 31) and \( \text{reservedQueues}[i].dequeue() \) at the local execution (line 56) give the same element. This is done casewise by considering each predicate for removing elements from \( \text{reservedQueues}[i] \) in the definition of \( \text{FlushReservedQueueFront} \) (line 84):

- \( \neg lStack.contains(front) \): If an element is removed from the front of \( \text{reservedQueues}[i] \) in \( p_i \) by \( \text{pop} \)'s response due to being already removed from \( p_i \)'s local stack, then it is also removed from \( \text{reservedQueues}[i] \) in every process by \( \text{pop} \)'s local execution, because it is also already removed from every process’ local stack, as elements are only removed during \( \text{EXECUTELOCALLY} \) in all processes.

We show the converse that if an element is removed from \( \text{reservedQueues}[i] \) in every process by \( \text{pop} \)'s local execution due to being already removed from every process’ local stack, then it has also been removed from \( \text{reservedQueues}[i] \) in \( p_i \) by \( \text{pop} \)'s response. This is done by induction, starting with the base case that if \( \text{pop} \) is mutually active with no other slow Pops, then an element already removed from
every process’ local stack before pop’s local execution must have been removed by an earlier-invoked fast Pop. The fast Pop must have returned the same element and marked it popped before pop’s response, fulfilling the condition to remove it from reservedQueues[i] in pi by pop’s response, since the response time of fast Pops is $\epsilon$, which is at most that of slow Pops.

The inductive step is that if we assume the converse statement holds for a certain number of mutually active slow Pops, then it holds if pop is an additional mutually active slow Pop invoked later than the rest. Slow Pops invoked by other processes do not use the ith index of reservedQueue[], so if there are no slow Pops invoked by pi then the reasoning is the same as the base case. Assume that there are mutually active slow Pops invoked by pi, and let pop_prev denote the previous such slow Pop. If an element elem is removed from reservedQueues[i] in all processes due to being already removed in their local stacks sometime between the local executions of pop_prev and pop, then elem was not removed from reservedQueues[i] in pi at the time of pop_prev’s response by the first proposition of this bullet point. elem must, however, be removed from reservedQueues[i] in pi by pop’s response, either because of an earlier-invoked fast Pop (the reasoning proceeds like the base case) or because it was returned and marked popped (line 30) by pop_prev, since these are the only types of operation instances that can remove elem from every process’ local stack before pop executes locally. Assuming the latter case, since pop_prev must respond before pop is invoked and hence before pop responds, elem is removed from reservedQueues[i] in pi by pop’s response.

• front.popped: If an element elem labeled pi is popped, then it must have returned by a Pop, and elem.popped = true only in process pi (line 12). This means that if an element is removed from reservedQueues[i] in pi by pop’s response due to being popped in pi, then it must have been returned by a Pop invoked in pi before pop was invoked, and hence will execute locally before the slow Pop does because all
operation instances invoked by \( p_i \) execute locally in \( p_i \) in the same amount of time, \( 2d \). Hence, the element will be removed from the local stack in all processes on line 54.

Considering the converse, \( \text{reservedQueues}[i] \) never contains an \textit{popped} element in any process other than \( p_i \), because if it is labeled \( p_i \), it is marked \textit{popped} only in \( p_i \).

- \( \text{front.label} \neq p_j \): By Lemma 10, an element already returned by a \textit{Pop} is never relabeled. Then between \textit{pop}'s response in \( p_i \) and local execution in each process, whether an element in \( \text{reservedQueues}[i] \) is still labeled \( p_i \) remains constant.

We have shown that if an element is removed from \( \text{reservedQueues}[i] \) in \( p_i \) at \textit{pop}'s response, then it is removed from \( \text{reservedQueues}[i] \) in all other processes at \textit{pop}'s local execution, and vice versa. Therefore, the element \( p_i \) returns on line 31 is the same element that all processes share at the front of their reserved queues on line 56, which they will then remove from their local stacks.

By inductively showing that the local execution of each operation type maintains a legal linearization and satisfies the invariants defined thus far, we demonstrate that any execution of Algorithm 2 when linearized produces a legal sequence of operation instances according to Definition 2.

**Theorem 12.** For any execution of Algorithm 2, the permutation \( \pi \) given by Definition 1 is legal by the specification of an Out-of-Order \( k \)-Relaxed Stack. Thus, Algorithm 2 is a correct implementation of an Out-of-Order \( k \)-Relaxed Stack. \( \square \)

**Proof.** We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that \( \pi \) is legal, and also maintains (b) the labeling invariant and (c) the uniformity invariant. (b) and (c) are demonstrated by induction on the local stack, with the base case being that the invariants are vacuously satisfied when the stack is empty.

\( \text{Push:} \)
(a) As long as every value can be uniquely identified (such as in the method suggested in the footnote under Definition 1), (C2) in Definition 1 is satisfied and hence any \textit{Push} is legal.

(b) We assume inductively that the elements of the current local stack for any process exhibit a labeling satisfying the labeling invariant before the \textit{Push} executes locally.

If the size of the local stack is at most \( k \) following the \textit{Push}, then the \textit{Pushed} element is labeled with a new label (line 47 – chosen to be the process id such that the fewest elements are labeled with it) in order to preserve the labeling invariant, which states that all the elements in the local stack when it is of this size must be labeled because they are legal for a \textit{Pop} to return.

Otherwise, the new topmost element is labeled with the label belonging to the element that just left the safe region (line 49), and it is justified to unlabel the element given on line 51 by the following argument. By (iii) in the volatility invariant, the bottommost element \( \text{bottom} \) in the safe region is the \((k - \tau n + \min\{|\text{lowerRegion}()|, \text{popsInPending}\})\)th-topmost, and any action that decreases either operand in the \( \min \) function must be a \textit{Pop} that decreases it by 1 so that \( \text{bottom} \) moves closer to the top by 1, so by Lemma 4 no mutually active \textit{Push} will move a safe region element returned by a \textit{Pop} out of the topmost \( k \) elements in the local stack. Therefore, the top \( k \) and only the top \( k \) elements are labeled, and the the labeling invariant is maintained.

(c) Assume that the uniformity invariant holds before the \textit{Push} locally executes.

If the local stack still has at most \( k - \tau n \) elements after the element is added by the locally executed \textit{Push} (line 46), then we first state that every element is in the safe region – volatile elements exist only if the size of the local stack is greater than \( k - \tau n \), since only in this case do locally executing \textit{Pushes} make them volatile; also, locally executing \textit{Pops} that reduce the size of the stack make volatile elements
non-volatile. Without volatile elements, the safe region is simply the set of $k - \tau n$ topmost elements.

Then, by the pigeonhole principle, fewer than $\left\lceil \frac{(k - \tau n)}{n} \right\rceil = \left\lceil \frac{k}{n} \right\rceil - \tau$ elements have the label shared by the smallest number of elements (chosen on line 47) before the $Push$. This increases to at most or fewer than $\left\lceil \frac{k}{n} \right\rceil - \tau$ following the $Push$, depending on whether the number of processes such that $\left\lceil \frac{k}{n} \right\rceil - \tau$ elements are labeled with its id is fewer than $n_a$ before the $Push$.

Also, if after the $Push$ the local stack has exactly $k - \tau n$ elements, then they are all labeled. If $n \mid k$ then every process must have $\left\lceil \frac{k}{n} \right\rceil - \tau = k/n - \tau$ elements labeled with its id, or the pigeonhole principle would require that some process has more than $\left\lceil \frac{k}{n} \right\rceil - \tau$ elements, violating the uniformity invariant. We can see that in this case, the second part of the uniformity invariant is maintained, that the number of elements labeled with each label differ by at most one.

If $n \nmid k$ and $n'_a$ denotes the number whose label is shared by $\left\lceil \frac{k}{n} \right\rceil - \tau$ elements, then the uniformity invariant implies that the number of elements with labels shared by fewer than $\left\lceil \frac{k}{n} \right\rceil - \tau$ elements is at least

$$(k - \tau n) - (\left\lceil \frac{k}{n} \right\rceil - \tau)n_a = \left(\frac{k - \left\lceil \frac{k}{n} \right\rceil n_a}{n - n_a} - \tau\right)(n - n_a)$$

and we know that the number of processes whose id is given by one of these labels
is $n - n'_a$. In fact, the above bound is tight and $n'_a = n_a$ because otherwise, by the pigeonhole principle, one of these processes would have at least $\lceil k/n \rceil$ elements, contradicting its definition. Using the fact that $\lceil k/n \rceil - \lfloor k/n \rfloor = 1$ shows that the second part of the uniformity invariant is also maintained in this case.

In the remaining case that the local stack has greater than $k - \tau n$ elements, the element new to the safe region is the higher element in a (higher, lower)-pair, so by the volatility invariant, the lower element shares the same label and is the only element that leaves the safe region. The number of elements in the safe region with each label is preserved, so the uniformity invariant holds.

**Pop:**

(a) We now show that each *Pop* instance satisfies (C3) in Definition 2.

First, a *Pop* invoked by a process $p_i$ only returns an element labeled with the process’ id, and subsequently makes the element popped so that $p_i$ will not again return the same element in a later *Pop*. Because *Pushes* add unique elements and Lemma 11 states that a *Pop* that returns an element in the local stack removes the same element from each local stack, *Pops* return unique values.

Since every process executes operation instances in linearized order, the second part of (C3) is equivalent to stating that the element removed by the *Pop* is among the topmost $k$ in every process’ local stack at the time of the *Pop*’s local execution.

During invocation in a process $p_i$, a *Pop* instance, which we denote *pop*, finds (line 10) whether the safe region of $p_i$’s local stack contains a non-popped element *val* labeled $p_i$. If so, *val* is legal to return and *pop* is a fast *Pop*, because by (iii) in Lemma 7, the bottommost element in the safe region of the local stack when a fast *Pop* occurs is the $(k - \tau n + \text{popsIncoming})$th-topmost in the local stack; therefore, it will not be moved outside the topmost $k$ elements by any of the at most $\tau, n$ mutually active *Pushes* that precede *pop* in $\pi$ (if they exist) without being moved back in by
any of the at least \( \text{popsIncoming} \) mutually active \( \text{Pops} \) that precede \( \text{pop} \) in \( \pi \) (if they exist). If not, but the safe region constitutes the entire local stack (line 10), then \( \bot \) will be returned, which is legal by the definition of an Out-of-Order \( k \)-Relaxed Stack because the mutually active \( \text{Pushes} \), being limited to \( \tau_c n \) in number, will not cause the size of the local stack to become \( k \) or greater.

In the remaining case, where \( \text{val} \) does not exist and there are elements outside the safe region, \( \text{pop} \) is a slow \( \text{Pop} \). A slow \( \text{Pop} \) is only invoked in \( p_i \) if the size of the local stack is at least \( k - \tau n \) and all the elements in the safe region labeled \( p_i \) are popped (line 10), which means at least that from the uniformity invariant, at least \( l - \tau \) fast \( \text{Pops} \) (line 12) have responded but not yet executed locally.

Any of the fast \( \text{Pops} \) at local execution relabels either a volatile element within the topmost \( k - \tau n \) elements in the local stack if one exists (line 61) or the new \( (k - \tau n) \)th element (line 63) \( p_i \), adding it to the safe region by its definition. It also enqueues the element in the reserved queue for process \( p_i \) (line 65), which \( \text{pop} \) may then return (line 56).

\( \text{pop} \)'s response occurs within \( \epsilon \) time of the element being added to the reserved queue since the timer set at invocation is reset at \( \epsilon \)-length time intervals before the maximum waiting time has passed (line 33). No more than \( \tau n \) \( \text{Pushes} \) could have executed locally by Lemma 4 and the observation that each process cannot locally execute more than one operation instance in an \( \epsilon \)-length time interval. Therefore, the returned element is still within the topmost \( k \) in the local stack at the time of \( \text{pop} \)'s response and is hence legal to return.

If \( \bot \) is returned after the maximum time \( 2d - (l - \tau)\epsilon \) has passed between \( \text{pop} \)'s invocation and response, which is the latest time that the first fast \( \text{Pop} \) may return since it takes \( (l - \tau)\epsilon \) time for \( l - \tau \) fast \( \text{Pops} \) to invoke and return in the same process and each executes locally in \( 2d \) time, then we may deduce that the fast \( \text{Pops} \)
each found no volatile element to relabel and the \((k - \tau n)\)th element was \(\perp\); thus, the local stack was of size less than \(k - \tau n\), and \(\perp\) is legal to return.

(b) By induction, the labeling done by \(Pops\) satisfies the labeling invariant, because if after the local execution of a \(Pop\) removes an element from the local stack it subsequently has fewer than \(k\) labeled elements, then it is necessary and sufficient to label the topmost unlabeled element if one exists (lines 67). If the local stack has at least \(k\) labeled elements on the other hand, then by the definition of the labeling invariant there must previously have been volatile elements. Since the \(Pop\) decrements \(|lowerRegion()|\) by 1 by removing a volatile element and also decrements \(popsInPending\) by 1, the labeling invariant is automatically maintained in this case.

(c) If the local stack had at most \(k - \tau n\) elements before the \(Pop\) executed locally, then the number of elements with some label is decreased following the \(Pop\) and none are increased, so the first part of the uniformity invariant still holds, and the second part does not apply. We are also done if the element removed by the \(Pop\) was in the contention region at the time of \(Pop\)'s local execution, because in this case decrementing \(popsInPending\) (line 58) is the only change made to the safe region, and it preserves the uniformity invariant by Corollary 9.

Assuming that the none of the above conditions hold, the local execution of the \(Pop\) moves an element from the contention region to the safe region; since the number of elements labeled \(p_j\) has decreased by one, labeling some element \(p_j\) which becomes part of the safe region allows the uniformity invariant to again hold. Depending on whether there are any volatile elements, this element is chosen to be either the newly-entered element in the safe region from the bottom (line 63) or some element in the volatile region sharing the same label (line 61) which is then made non-volatile. In the latter case, if the element was outside the safe region, then this action together with decrementing \(popsInPending\) (line 58) brings this element into the safe region.
Lastly, we show that message receipts from \textit{Pops} preserve the uniformity invariant. The execution that increments \textit{popsInPending} (line 22) upon receiving such a message satisfies the invariant by Corollary 9, and the execution upon receiving a message from a fast \textit{Pop} (line 24) only makes an element that was already in the safe region (line 24) non-volatile, so it does not modify the set returned by \textit{safeRegion}().

\[
\square
\]

### 3.1.4 Performance

Our discussion of performance clarifies the purpose of defining the effective quantities given at the beginning of this section.

We first consider the \textit{Push} operation. The first result immediately follows from observing the response time for fast \textit{Pushes} (line 4) and the two different possible response times for slow \textit{Pushes} (lines 8 and 36, with the second being the sum) in Algorithm 2.

**Theorem 13.** The worst-case time complexity of \textit{Push} among all complete, admissible executions of Algorithm 2 is no more than \(\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}\).

\[
\square
\]

**Remark 2.** At least \(\tau_c - 1\) fast \textit{Pushes}, which together take at least \((\tau_c - 1)\epsilon\) time (if each is executed one right after another with no gaps in between), must be invoked by a single process and not yet executed locally before the next \textit{Push} is slow. We note that this is impossible if \(2d < (\tau_c - 1)\epsilon\) implying \(d < (\tau_c - 1)\epsilon/2\); in this case, the first operand in the upper bound given in Theorem 13 would become negative, which would have been an invalid result. We reach the interesting conclusion that all \textit{Pushes} are fast when and only when \(d < (\tau_c - 1)\epsilon/2\).

To lead up to our next theorem, we consider the view of a single process \(p_i\) in which occurs a slow \textit{Push}, i.e., the \(\tau_c\)th in a subsequence \(\sigma\) of \textit{Pushes} contained in the sequence of operation instances invoked by \(p_i\) in which less than \(2d\) time has passed since the invocation of the first instance. We also require \(\sigma\) to consist of at least \(2\tau_c\) \textit{Pushes}.
Let $t_0$ denote the time of invocation of the first instance in $\sigma$, and let $gap_x$ denote the amount of time that passes between the time the $x$th instance returns and the time the $(x + 1)$st instance is invoked for $x \in [1, \tau_c - 1]$. Then the time of invocation of the slow $Push$ is $t_0 + (\tau_c - 1)\epsilon + \sum_{m=1}^{\tau_c-1} gap_m$. From Theorem 13, the slow $Push$ returns in $\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}$ time, so the time between its invocation and return is

$$
t_0 + (\tau_c - 1)\epsilon + \sum_{m=1}^{\tau_c-1} gap_m + \max\{2d + (1 - \tau_c)\epsilon, \epsilon\}
$$

(3.1)

In the following argument, we look ahead to the $\tau_c - 1$ $Push$ instances that follow after the $\tau_c$th in $\sigma$.

**Lemma 14.** For $2 \leq x \leq \tau_c$, the latest time the $x$th $Push$ executes locally is $\epsilon$ time after the $(\tau_c + x - 1)$st $Push$ is invoked.

**Proof.** First, consider the invocation of the $(\tau_c + 1)$st $Push$ instance in $\sigma$. Since the earliest time it can be invoked is given by (3.1), and the second instance in $\sigma$ is invoked at time $t_0 + \epsilon + gap_1$ and executed locally at time

$$
t_0 + \epsilon + gap_1 + d + \epsilon,
$$

the difference in times between the local execution of the second instance and the invocation of the $(\tau_c + 1)$st instance is $\epsilon - \sum_{m=2}^{\tau_c-1} gap_m \leq \epsilon$.

Continuing the argument by induction for $2 \leq x \leq \tau_c - 1$, if the latest time the $x$th instance executes locally is $\epsilon$ time after the $(\tau_c + x - 1)$st instance is invoked, then, since the $(x + 1)$st instance executes locally at time $gap_x + \epsilon$ after the $x$th instance does so while the earliest time the $(\tau_c + x)$th instance in $\sigma$ can be invoked is $\epsilon$ after the $(\tau_c + x - 1)$st is invoked, we have that the difference in times between the local execution of the $(x + 1)$st instance and the invocation of the $(\tau_c + x)$th instance is less than $\epsilon - \sum_{m=x+1}^{\tau_c-1} gap_m \leq \epsilon$. 

45
Theorem 15. The amortized time complexity of Push among all complete, admissible executions of Algorithm 2 is no more than \( \max\{2d/\tau_c, \epsilon\} \).

Proof. From Remark 2, if \( d < (\tau_c - 1)\epsilon/2 \), then every Push takes \( \epsilon \) time, giving an amortized time complexity of \( \epsilon \) (which satisfies the proposition since \( 2d/\tau_c < \epsilon \)), so we assume that \( d \geq (\tau_c - 1)\epsilon/2 \) so that it is possible for slow Pushes to occur.

With Lemma 14 in hand, we now know that the \( \tau_c - 1 \) instances following the \( \tau_c \)th instance also execute locally in \( \epsilon \) time, the fastest possible. This is because for \( 2 \leq x \leq \tau_c \), if the \((x + 1)\)st instance in \( \sigma \) is executed locally before the \((\tau_c + x)\)th is invoked, then the \((x + 1)\)st instance decrements localFastPushesActive (line 52), guaranteeing that the \((\tau_c + x)\)th instance is a fast Push. If the \((x + 1)\)st instance is locally executed after the latter is invoked, then it must be within \( \epsilon \) time, and thus in all processes the former decrements localFastPushesActive soon enough that the latter is able to respond \( \epsilon \) time after invocation, as equally as fast as a fast Push.

The first Push instance after the \( \tau_c \)th that may require a response time of more than \( \epsilon \) is the \( 2\tau_c \)th, and so by Theorem 13 it returns in at most \( \max\{2d + (1 - \tau_c)\epsilon, \epsilon\} \) time. Thus, we have a repeating pattern of \( \tau_c \) instances, and the amortized cost of each repetition is

\[
\frac{(\tau_c - 1)\epsilon + \max\{2d + (1 - \tau_c)\epsilon, \epsilon\}}{(\tau_c - 1) + 1} = \frac{\max\{2d, \tau_c\epsilon\}}{\tau_c} = \max\left\{ \frac{2d}{\tau_c}, \epsilon \right\}.
\]

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow Push which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of Pushes at any process \( p_i \), it is also an upper bound on the average cost of Pushes at all processes; hence, it is the amortized cost. \( \square \)
The Pop operation mirrors the Push operation in that a slow Pop in any sequence of Pop instances follows a series of fast Pops, but the minimum number of fast operation instances is now \( l - \tau \) instead of \( \tau_c - 1 \). The first result comes directly from simplifying the response time for fast Pops (line 14) and the two different possible response times for slow Pops (lines 17 and 33) in Algorithm 2.

**Theorem 16.** The worst-case time complexity of Pop among all complete, admissible executions of Algorithm 2 is no more than \( \max\{2d + (\tau - l)\epsilon, \epsilon\} \).

**Remark 3.** At least \( l - \tau \) fast Pops, which together take at least \( (l - \tau)\epsilon \) time (if each is executed one right after another with no gaps in between), must be invoked by a single process and not yet executed locally before the next Pop is slow. We note that this is impossible if \( 2d < (l - \tau)\epsilon \) implying \( d < (l - \tau)\epsilon/2 \); in this case, the first operand in the upper bound given in Theorem 16 would become negative, which would have been an invalid result. We reach the interesting conclusion that all Pops are fast when and only when \( d < (l - \tau)\epsilon/2 \).

**Lemma 17.** For \( 2 \leq x \leq l - \tau + 1 \), the latest time the \( x \)th Pop executes locally is \( \epsilon \) time after the \( (l - \tau) \)th Pop is invoked.

**Theorem 18.** The amortized time complexity of Pop among all complete, admissible executions of Algorithm 2 is no more than \( \max\{2d/(l - \tau + 1), \epsilon\} \).

**Proof.** The reasoning is the same as for the amortized time complexity of Push (Theorem 15), and the associated calculation is as follows:

\[
\frac{(l - \tau)\epsilon + \max\{2d + (\tau - l)\epsilon, \epsilon\}}{(l - \tau) + 1} = \frac{\max\{2d, (l - \tau + 1)\epsilon\}}{l - \tau + 1} = \max\left\{\frac{2d}{l - \tau + 1}, \epsilon\right\}.
\]

We can now remark that during usage of the Out-of-Order \( k \)-Relaxed Stack it is always worth tuning the \( \tau \) parameter to strike a fine balance between the performance of Push
and *Pop*, since their amortized time complexities are both concave functions. This fact is further illustrated in the plot below.

![Figure 3.5: Plots of the amortized time complexities of *Push* (blue) and *Pop* (red) parameterized by $\tau$, where $l = 10$ and $d$ is an arbitrary constant.](image)

This completes our analysis of Algorithm 2.
CHAPTER 4
GENERALIZATIONS AND EXTENSIONS

4.1 Relaxed Priority Queues

Relaxed stacks and queues can be thought of as special cases of relaxed priority queues, which have operations Insert and ExtractMax (the one that is relaxed) and whose elements are ordered from the top in decreasing priority. We give the definition of an unrelaxed Priority Queue based on that of the unrelaxed Stack, Definition 1.

Definition 17. A Priority Queue over a set of values \( V \) is a data type with two operations:

- \( \text{Insert}(\text{val}, -) \), \( \text{val} \in V \),
- \( \text{ExtractMax}(-, \text{val}) \), \( \text{val} \in V \cup \{\bot\} \).

A Priority Queue provides a priority function \( P : V \rightarrow S \), where \( S \) is defined as a well-ordered set such that \( P \) is bijective. Elements are timestamped as they are added, so a Priority Queue also provides a function \( t : V \rightarrow \mathbb{R}_{>0} \) that can be used to determine the timestamp when each element was Inserted.

Given a sequence \( \rho \) of instances of operations on a Stack, define the following terms:

- An \( \text{Insert}(\text{val}, -) \) is matched if there exists a \( \text{ExtractMax}(-, \text{val}) \) in \( \rho \). If this is the case, \( \text{Insert}(\text{val}, -) \) is said to match \( \text{ExtractMax}(-, \text{val}) \) in \( \rho \).
- An \( \text{ExtractMax}(-, \text{val}) \) in \( \rho \) is headmost if there exists an \( \text{Insert}(\text{val}, -) \) in \( \rho \) such that for every unmatched \( \text{Insert}(\text{val}', -) \) in \( \rho \), \( P(\text{val}) > P(\text{val}') \).
- An \( \text{ExtractMax}(-, \text{val}) \) in \( \rho \) is resolved if it is either headmost or succeeded by a headmost \( \text{ExtractMax} \) instance.

A matched \( \text{Insert}(\text{val}, -) \) in \( \rho \) is resolved if some \( \text{ExtractMax}(-, \text{val}) \) it matches is resolved.
• The lateness of \( \rho, \lambda(\rho) \), is the number of unresolved ExtractMax instances in \( \rho \).

• An ExtractMax\((-, val)\) in \( \rho \) is own-top resolved if it is either headmost or succeeded by a headmost ExtractMax\((-, val')\) such that

\[
- t(val) > t(val'), \text{ and}
\]

\[
- \text{there is no Insert}(val'', -) such that } t(val) > t(val'') > t(val').
\]

• The current-top lateness of \( \rho, \gamma(\rho) \), is the number of own-top unresolved ExtractMax instances succeeding the unmatched Insert in \( \rho \) whose value has the latest timestamp.

A sequence of operation instances is legal iff it satisfies the following conditions:

(C1) The empty sequence is legal.

(C2) If \( \rho \) is a legal sequence of operation instances, then \( \rho \cdot \text{Insert}(val, -) \) is legal iff there is no Insert\((val, -)\) already in \( \rho \).

(C3) If \( \rho \) is a legal sequence of operation instances, then \( \rho \cdot \text{ExtractMax}\((-, val)\), val \neq \perp \) is legal iff ExtractMax\((-, val)\) is headmost.

\[ \rho \cdot \text{ExtractMax}\((-, \perp) \) is legal iff every instance of Insert in \( \rho \) is matched. \]

By methodically replacing corresponding operations and terms, analogous definitions (e.g. Out-of-Order \( k \)-Relaxed Priority Queue, Lateness \( k \)-Relaxed Priority Queue, etc.) can be stated for every type of relaxed stack presented in Section 2. The ensuing discussion assumes that these definitions have been implicitly stated.

4.1.1 Reductions

Relaxed stacks are in fact relaxed priority queues in which the priority of elements is determined by how recently they were Pushed, where we consider the respective analogous operations to be Push – Insert and Pop – ExtractMax. This means that relaxed priority queues behave just like relaxed stacks with the exception that Pushed elements can be
added at the head, the tail, or between any existing elements in the data structure depending on their priority.

Any algorithm implementing any type of relaxed stack given in Section 3 or Appendix A can be reduced to an algorithm implementing the analogous type of relaxed priority queue with identical performance, where reducibility is given in the sense that the distributed time complexity in terms of our model is unchanged. The procedure is detailed in the following theorem.

**Theorem 19.** If the following changes are applied to any existing algorithm that implements a relaxed stack:

- the local variables are defined identically as in the beginning of Section 3, although names may be changed as appropriate;
- the topmost and bottommost elements now respectively refer to the elements of highest priority and the elements of lowest priority in all instances, except that wherever we previously considered the topmost volatile elements, we now instead use volatile elements having the latest timestamp, defining the new operations latestBySet() and latestSetBySet() on the local priority queue for this purpose;
- the definitions of the terms and the labeling, uniformity, and headmost invariants are the same as before, in the algorithms where they are applicable;
- when an Insert takes place,
  - if the Inserted element is placed either in the safe region or at a higher priority than some element in the safe region, then the procedure followed in the Insert’s local execution remains the same (the Push operation in the relaxed stack algorithms falls within this category);
– if it is placed at a lower priority than all the elements in the safe region but still within the set of elements legal to return, then we follow the more straightforward procedure of assigning it a dummy label and unlabeling the element moved out of the legal set in order to maintain the labeling invariant, while the safe region and hence the legality of elements and all the other invariants are preserved;
– otherwise, if it is outside the set of legal elements, then all the legal elements are preserved, so no labels need to be changed;

- for the relaxed ExtractMax operation, the executions in the messages it passes and its local execution remain the same;

then the new algorithm correctly implements the corresponding relaxed priority queue, and furthermore, the relaxed priority queue shares the same worst-case and amortized time complexities and lower bounds as its analogous relaxed stack.

Proof. If we combine the reasoning associated with the augmented procedures for Insert in addition to the fact that the relaxed ExtractMax operation behaves identically to the Pop operation, then the reasoning for the proof of correctness represented by the statements and proofs of each lemma and theorem will be identical. The performance proofs also remain the same because the changes are all accounted for during local execution of operation instances.

As an example, the reduction from Theorem 19 applied to the Out-of-Order $k$-Relaxed Stack algorithm (Algorithm 2) results in the Out-of-Order $k$-Relaxed Priority Queue algorithm (described in Appendix B as Algorithm 7).

The currently known optimal bounds are summarized comprehensively in Tables 4.1 and 4.2.
Table 4.1: Bounds on \textit{Push/Insert} Time Complexity

<table>
<thead>
<tr>
<th></th>
<th>Worst-Case Cost</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower Bound</td>
<td>Upper Bound</td>
</tr>
<tr>
<td>Unrelaxed</td>
<td>((1 - \frac{1}{n})u) [2]</td>
<td>(\epsilon) [2]</td>
</tr>
<tr>
<td>Out-of-Order, Restricted, Windowed</td>
<td>?</td>
<td>(\max{2d + (1 - \tau_c)\epsilon, \epsilon})</td>
</tr>
<tr>
<td>Lateness</td>
<td>?</td>
<td>(\epsilon)</td>
</tr>
</tbody>
</table>

Table 4.2: Bounds on \textit{Pop/ExtractMax} Time Complexity

<table>
<thead>
<tr>
<th></th>
<th>Worst-Case Cost</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower Bound</td>
<td>Upper Bound</td>
</tr>
<tr>
<td>Unrelaxed</td>
<td>(d + \min{\epsilon, u, \frac{d}{3}}) [2]</td>
<td>(d + \epsilon) [2]</td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>(\frac{d}{\tau}, k &lt; n^2) (\uparrow)</td>
<td>(\max{d + \epsilon[2], 2d + (\tau - l)\epsilon, \epsilon})</td>
</tr>
<tr>
<td>Lateness</td>
<td>(d) [11]</td>
<td>(d + \epsilon) [2]</td>
</tr>
<tr>
<td>Restricted, Windowed</td>
<td>(d) [11]</td>
<td>(\frac{d}{\lfloor k/n \rfloor} \uparrow\uparrow)</td>
</tr>
<tr>
<td>Per-Top</td>
<td>(d) [11]</td>
<td>(d + \epsilon) [2]</td>
</tr>
<tr>
<td>Stuttering</td>
<td>(d) [11]</td>
<td>(d + \epsilon) [2]</td>
</tr>
</tbody>
</table>

\(^\ast\)Assumes the bounds on \textit{Pop/ExtractMax} given in Table 4.2.

\(^\dagger\)Upper bound on worst-case cost is an upper bound on amortized cost.

\(^\ddagger\)Lower bound on amortized cost is a lower bound on worst-case cost.

\(\uparrow\uparrow\)The lower bound proof is the same as for the Restricted \(k\)-Relaxed Stack given in [11]. (The same proof contains a minor error in that \(\lfloor k/n \rfloor\) should be used in place of \(l\)).
4.1.2 Relaxed Queues as a Special Case

Because relaxed priority queues also generalize relaxed queues, where we consider the respective analogous operations to be $\text{Enqueue} - \text{Insert}$ and $\text{Dequeue} - \text{ExtractMax}$, we can deduce that the same upper and lower bounds apply for relaxed queues by extension, although it is obvious that they may not necessarily be the optimal upper and lower bounds [11] due to the specific properties of relaxed queues. In particular, we give a nontrivially-bounded algorithm implementing the Stuttering $k$-Relaxed Queue in Appendix A.4.

Because one property of relaxed queues is that the topmost element is changed only when it is removed, in the definitions where it applies, all unresolved elements become resolved whenever the topmost element is changed. Hence, the Lateness and Per-Top Lateness relaxations both equivalently correspond to the Lateness $k$-Relaxed Queue [10], and all four of the Restricted, Windowed, Per-Top Restricted, and Per-Top Windowed relaxations equivalently correspond to the Restricted Out-of-Order $k$-Relaxed Queue [10], which we refer to interchangeably as the Restricted $k$-Relaxed Queue or the Windowed $k$-Relaxed Queue.

In the same way that the Restricted $k$-Relaxed Priority Queue algorithm (reduced from Algorithm 5) provides an upper bound only for heavily loaded runs whereas the Windowed $k$-Relaxed Priority Queue algorithm (reduced from Algorithm 4) does so for any runs in general, a new Restricted $k$-Relaxed Queue algorithm derived from the Windowed $k$-Relaxed Priority Queue algorithm exhibits a similar additional performance improvement. The Restricted $k$-Relaxed Queue algorithm will hereafter refer to this derivation.

We give three theorems which illustrate further conditional properties of relaxed priority queue algorithms that apply unconditionally to relaxed queues, leading to their improved performance. It is worthwhile to mention that the Lateness Out-of-Order $k$-Relaxed Priority Queue algorithm (reduced from Algorithm 3) satisfies these theorems as well, which is why it exhibits the same performance improvements as the relaxed queue algorithms.
Definition 18. An Insert is safe if it is guaranteed not to move any element out of the set of safe elements in each local priority queue at local execution.

Theorem 20. Safe Inserts are always permitted to be fast, taking $\epsilon$ time to respond, without changing the response time of ExtractMaxes, unless they return $\bot$ before local execution in the implementations of relaxed priority queues where returning $\bot$ depends on the size of the local stack.

Proof. ExtractMaxes which are invoked after a safe Insert’s invocation but return an element in the local stack before its local execution will still behave legally as when no such safe Insert is invoked – because the ExtractMax necessarily returned before its local execution, the returned element is safe by definition, and will not be made unsafe by the safe Insert. Thus, allowing the Insert to respond at the earliest time possible does not require an increase in the response time of such ExtractMaxes.

In particular, if the Inserted element is placed below the safe region but is still among the set of elements legal to return, then the necessary changes in labeling to maintain the labeling invariant will not be made too soon or too late since every operation instance reassigns labels only at its local execution, which follows the same linearized ordering as invocation.

On the other hand, if a ExtractMax may return $\bot$ before local execution, then since the number of fast Inserts that may be invoked before the ExtractMax is invoked but locally executed after it returns may be much larger than $k$ (depending on the value of $d$ relative to $\epsilon$), the sequence obtained from linearizing these operation instances by invocation time may actually be illegal, if the definition of the relaxed priority queue requires that the number of unmatched Inserts is smaller than some number at most $k$. This means that in implementations of such relaxed priority queues, all ExtractMaxes returning $\bot$ may only do so at local execution, so no ExtractMaxes returning $\bot$ are fast. We will see that this impacts the performance of the Out-of-Order $k$-Relaxed Queue and the Restricted...
Another caveat is that an Insert can only be fast if it is possible to decide that it is safe within $\epsilon$ time of invocation, during which the state of the invoking process’ local priority queue may differ from that during the time of the Insert’s local execution. This is not an issue for relaxed queues, as Enqueue operations add elements to the opposite end of the data structure from where they will be removed and hence will not affect the legality of existing elements according to any relaxed queue definition, so Theorem 20 states that every Enqueue is predetermined to be safe.

**Theorem 21.** The $d + u$ minimum restriction on the pending delay that originates from Algorithms 2, 4, and 5 does not apply if Inserts are safe. □

**Proof.** If the condition holds, then the $t_{volatile}$ field and $remsIncomingSeen$ ($popsInPending$ in relaxed stacks) counter become extraneous because their use is contingent on the possibility that elements may be relabeled, which in turn is not necessary if elements never leave and re-enter the safe region, so the related actions that take place in messages passed by ExtractMaxes can be safely ignored. Thus, the restriction of $2d$ required by the message passing on the time between invocation and local execution of operation instances can be removed. □

The new pending delay for Algorithm 2 becomes $u + \epsilon$ so that operation instances still execute locally in timestamp order by Lemma 1. The new pending delay for Algorithms 4 and 5 becomes $2u$, as is required by the headmost invariant from the proof of correctness of Algorithm 3.

**Theorem 22.** If Inserts are safe, then the safe elements include all elements that are legal to return by locally executing Inserts. □

**Proof.** Since elements are never relabeled, if an element is currently legal for a locally executing ExtractMax to return, then it is also legal to return by any ExtractMax that will locally execute in the future, and hence we conclude that the element must be safe.
From the proof of Theorem 21, there are no volatile or lower regions, so the definitions of the labeling invariants state in this setting that labeled elements include only elements that are legal for a locally executing $ExtractMax$ to return.

Hence, in relaxed queue algorithms the uniformity invariant applies to the entire set of labeled elements, which is larger than the safe region as defined in the relaxed stack algorithms, and this leads to a further performance improvement.

Because there is no limit to the number of mutually active fast $Dequeue$s and there is no contention region, the algorithms and their time complexities for relaxed queues are independent of the $\tau$ parameter.

In summary, we may reduce from relaxed stack algorithms to relaxed priority queue algorithms and then apply the optimizations given in these theorems to the algorithms if they are restricted to specifically implement relaxed queues. The reasoning for the corresponding performance proofs for the Lateness $k$-Relaxed Stack algorithm in Section 3 is identical for the optimized Lateness $k$-Relaxed Queue algorithm, allowing us to obtain new upper bounds for this type of relaxed queue.

However, we recall that if Theorem 20’s optimization is applied to the Out-of-Order $k$-Relaxed Queue and Restricted $k$-Relaxed Queue algorithms, then any $Dequeue$ returning $\perp$ requires $d + \epsilon$ time, and this new restriction changes the performance proofs for these data structures.

We begin by introducing a definition describing runs in which the size of the local stack for each process is at least $k$ before every $ExtractMax$, so that $\tau_p = \tau_c = \tau$.

**Definition 19.** A run $R$ is heavily loaded if, for some linearization $\pi$ of $R$, every prefix of $\pi$ which is immediately followed by an instance of $ExtractMax$ has at least $k$ unmatched $Insert$.

It turns out that for heavily loaded runs, the amortized complexities for these data structures still follow the same proofs as before, so we obtain results for this specific case. The
amortized time complexities for non-heavily loaded runs require new proofs, however. For this purpose, we introduce a modified form of the \( l \) parameter which is the same as before in heavily loaded runs, but gives the number of elements labeled with each process id in each process’ local stack at any time.

**Definition 20.** At any process \( p_i \), the effective-\( l \), \( l_e \), and the effective-\( l \)se, \( l_{se} \), are defined as

\[
l_e := \begin{cases} 
  l, & \text{if } lPQ.size() \geq k \\
  \lfloor I/n \rfloor, & \text{if } lPQ.size() < k
\end{cases}
\]

and \( l_{se} := \begin{cases} 
  l_{se}, & \text{if } lPQ.size() \geq k \\
  \lfloor I/n \rfloor, & \text{if } lPQ.size() < k
\end{cases} \)

where \( I \) is the number of Inserts that executed locally in \( p_i \) after the local execution of the previous ExtractMax.

**Theorem 23.** For a complete, admissible execution of the derived Out-of-Order \( k \)-Relaxed Queue algorithm, the amortized time complexity of Dequeues at any process \( p_i \) during a time interval in which its effective-\( l \) is given by \( l_e \) is at most \( d/(l_e + 1) + \epsilon \). \( \square \)

**Proof.** The number of elements labeled \( p_i \) during the given time interval must be at least \( l_e \), because if the size of \( p_i \)’s local stack is less than \( k \) and \( I \) consecutive Enqueues execute locally, then the Enqueues must have labeled at least \( l_e \) elements with each process’ id.

Thus, if a slow Dequeue is invoked in \( p_i \), it must have followed \( l_e \) fast Dequeues, which returned the elements labeled \( p_i \) and marked them dequeued. Since each fast Dequeue takes \( \epsilon \) time and the slow Dequeue takes \( d + \epsilon \) time, the average cost of each repetition of a pattern of \( l_e \) fast Dequeues and one slow Dequeue is

\[
\frac{l_e \epsilon + (d + \epsilon)}{l_e + 1} = \frac{d + (l_e + 1) \epsilon}{l_e + 1} = \frac{d}{l_e + 1} + \epsilon.
\]

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow Dequeue which ends a pattern have the highest average cost.
Because this provides an upper bound on the cost of Dequeue at any process $p_i$, it is also an upper bound on the average cost of Dequeue at all processes; hence, it is the amortized cost.

**Theorem 24.** For a complete, admissible execution of the derived Restricted $k$-Relaxed Queue algorithm, the amortized time complexity of Dequeue at any process $p_i$ during a time interval in which its effective-$l$ is given by $l_{se}$ is at most $(d + u - \epsilon)/l_{se} + \epsilon$ if $l_{se} > 0$, or $d + u$ otherwise.

**Proof.** The proof is similar as in the previous theorem, but the pending delay is now $2u$ causing the response time of slow Dequeue to become $(d - u) + 2u = d + u$; additionally, the number of elements labeled $p_i$ that can be returned by a fast Dequeue is $l_{se} - 1$; the topmost is excluded, since by the headmost invariant it cannot be returned by a fast Dequeue.

Table 4.3: Bounds on Dequeue Time Complexity in Heavily-Loaded Runs Only**

<table>
<thead>
<tr>
<th></th>
<th>Worst-Case Cost</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower Bound</td>
<td>Upper Bound</td>
</tr>
<tr>
<td>Unrelaxed</td>
<td>$d + \min{\epsilon, u, \frac{d}{T}}$ [2]</td>
<td>$d + \epsilon$ [2]</td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>$\frac{d}{T}, k &lt; n^2$</td>
<td>$\max{d + (1 - l)\epsilon, \epsilon}$</td>
</tr>
<tr>
<td>Lateness</td>
<td>$d$ [11]</td>
<td>$d + \epsilon$ [2]</td>
</tr>
<tr>
<td>Restricted</td>
<td>$d$ [11]</td>
<td>$d + \epsilon$ [2]</td>
</tr>
<tr>
<td>Stuttering</td>
<td>$d$ [11]</td>
<td>$d + \epsilon$ [2]</td>
</tr>
</tbody>
</table>

\*\†\†The lower bound proof is the same as for the Restricted $k$-Relaxed Stack given in [11]. (The same proof contains a minor error in that $\lfloor k/n \rfloor$ should be used in place of $l$.)

**Assumes an upper bound of $\epsilon$ on Enqueue, which changes the Out-of-Order and Restricted results.
Table 4.4: Bounds on Dequeue Time Complexity

<table>
<thead>
<tr>
<th></th>
<th>Worst-Case Cost</th>
<th>Amortized Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lower Bound</td>
<td>Upper Bound</td>
</tr>
<tr>
<td>Unrelaxed</td>
<td>(d + \min{\epsilon, u, \frac{d}{T}}[2])</td>
<td>(d + \epsilon[2])</td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>(\frac{d}{T}, k &lt; n^2)</td>
<td>(d + \epsilon[2])</td>
</tr>
<tr>
<td>Lateness</td>
<td>(d[11])</td>
<td>(d + \epsilon[2])</td>
</tr>
<tr>
<td>Restricted</td>
<td>(d[11])</td>
<td>(d + \epsilon[2])</td>
</tr>
<tr>
<td>Stuttering</td>
<td>(d[11])</td>
<td>(d + \epsilon[2])</td>
</tr>
</tbody>
</table>

The upper bounds on time complexities for relaxed queues are aggregated in Tables 4.3 and 4.4 (no table is given for Enqueue since it is assumed that the upper bounds are all \(\epsilon\)), which illustrate the following new results:

- The upper bounds are improved on Dequeue operations in heavily loaded runs of Out-of-Order \(k\)-Relaxed Queues, originating from slow Pops being able to respond in less than \(d + \epsilon\) time.
- A correct algorithm implementing the Lateness \(k\)-Relaxed Queue is (implicitly) derived.
- The derived Restricted \(k\)-Relaxed Queue algorithm improves the previous upper bound by nearly a factor of 2, and also guarantees an upper bound on non-heavily loaded runs.

Finally, if we decide to apply the same derivation as before for the Out-of-Order \(k\)-Relaxed Queue and Restricted \(k\)-Relaxed Queue algorithms, but without Theorem 20’s optimizations this time, then repeating the original performance proofs gives a few new results that apply generally (not just for heavily loaded runs). These are given in Table 4.5.
Table 4.5: Upper Bounds without Theorem 20’s Optimizations in the General Case

<table>
<thead>
<tr>
<th></th>
<th>Worst-Case Cost</th>
<th></th>
<th>Amortized Cost</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Enqueue</td>
<td>Dequeue</td>
<td>Enqueue</td>
<td>Dequeue</td>
</tr>
<tr>
<td>Out-of-Order</td>
<td>$\max{d + (2 - \tau_c)\epsilon, \epsilon}$</td>
<td>$\max{d + (\tau - l + 1)\epsilon, \epsilon}$</td>
<td>$\max{\frac{d+\epsilon}{\tau_c}, \epsilon}$</td>
<td>$\frac{d+\epsilon}{l-\tau+1}$</td>
</tr>
<tr>
<td>Restricted</td>
<td>$\max{d + (2 - \tau_c)\epsilon, \epsilon}$</td>
<td>$d + \epsilon$</td>
<td>$\max{\frac{d+\epsilon}{\tau_c}, \epsilon}$</td>
<td>$\frac{d+n-\epsilon}{\min{l, l-\tau}} + \epsilon$</td>
</tr>
</tbody>
</table>

For the purpose of completeness, the completely optimized (including Theorem 20) algorithms derived from the original relaxed stack algorithms are provided as Algorithms 8, 9, and 10 in Appendix B, simplified by having the functionally extraneous pseudocode removed.

In the Out-of-Order $k$-Relaxed Queue algorithm, the reason for `reservedQueues[]` not being needed is that new elements are added and labeled in the same ordering as the queue data structure itself.

In both of the Out-of-Order $k$-Relaxed Queue and Restricted $k$-Relaxed Queue algorithms, line 5 has been modified to prevent a fast `Dequeue` from returning $\bot$. `localFastPopsInvoked` and `localFastPopsExecuted` become extraneous in the latter because its only purpose was to limit the number of consective fast `Pops` that returned $\bot$. For the former in particular, lines 26-28 have been added to ensure that a slow `Dequeue` that would have previously returned $\bot$ instead continues for $d + \epsilon$ time before response, while for the latter, slow `Dequeues` always respond in $d + \epsilon$ time.
CHAPTER 5
CONCLUSION

From the starting point of relaxed stacks, we have fully characterized efficient algorithms for linearizable relaxed priority queues under a particular distributed system model. One way of conceptually surveying the relationships between the relaxed ordered data types in hindsight is that relaxed stacks are the “worst case” while relaxed queues are the “best case” of relaxed priority queues.

We intend to implement these data types in software with the help of existing libraries such as Open MPI, and compare them to existing benchmarks as well as release them under an open-source license. Relaxed ordered data types have previously been implemented in shared memory [10]; relaxed priority queues in particular have also seen implementations under different models and been analyzed empirically [13], [14]. From a broader perspective, we see an ongoing trend toward the loosening of the semantics of data types in a reasonable manner in order to improve their performance and scalability in the most general types of distributed systems. The work done so far in this paper serves to propel the theoretical aspects underlying this research direction.

Relaxed priority queues have a rich set of applications that we would like to explore, looking forward. For example, discrete event simulators, which are used to model the systems underlying scenarios as diverse as chemical processing, automotive manufacturing, and network traffic monitoring, can be implemented efficiently using priority queues. It is possible that if the data is sufficiently well-behaved, then relaxed priority queues can be used in place of priority queues in many more performance-critical settings where computation is distributed across multiple clusters.
Appendices
APPENDIX A
ADDITIONAL ALGORITHMS AND UPPER BOUNDS

A.1 Lateness Relaxed Stacks

Algorithm 3 implements the Lateness $k$-Relaxed Stack (Definition 3). The algorithm actually enforces the slightly stronger lateness condition that

$$\lambda(\rho) \leq l_s n = \left(\left\lceil\frac{k}{n}\right\rceil - 1\right) n = \begin{cases} k - n, & n \mid k \\ \lfloor k/n \rfloor \cdot n, & n \nmid k \end{cases}$$

(A.1)

rather than $\lambda(\rho) < k$ as in Definition 3, given that $\rho$ denotes the linearization of operation instances.

`localPopsInvoked` and `localPopsExecuted` are newly utilized local variables intended to ensure the above lateness condition. We show that `localPopsInvoked` and `localPopsExecuted` are respectively the number of unresolved `Pops` invoked and the number of unresolved `Pops` executed locally by the process. We can see from the algorithm that `localPopsExecuted` never exceeds `localPopsInvoked`, which is capped at $l_s$ in each process.

The labeling invariant in this algorithm states that every element in the local stack is labeled, because by the specification given by Definition 3, every element is legal to return as long as $\lambda(\rho) < k$. Labels are never changed after they are assigned, so there is no contention region, and all `Pushes` are permitted to fast, taking $\epsilon$ time; thus, the algorithm is independent of $\tau$ and is designed to optimize the performance of `Pops`.

The headmost invariant states that in each process $p_i$, the topmost non-popped element labeled $p_i$, and only that element, has `canPopByLabel = false`. `canPopByLabel` is also a newly utilized local variable which must be `true` in order for the element to be returned.
by `popByLabel()` or `peekByLabel()`.

When `localPopsInvoked = l_s` in some process, the next `Pop` to be invoked will be slow. The headmost invariant ensures that the very first slow `Pop, pop_1`, to execute locally always returns and removes the top element in the local stack (since it is the topmost element assigned some label), resolving all the previously locally executed `Pops`. Any slow `Pops` mutually active with `pop_1` will then have `localPopsExecuted = 0`, so they are free to return and remove a non-top element.

The first locally executing slow `Pop` not mutually active with `pop_1, pop_2`, will return and remove the topmost non-`popped` element, `elem`. We will see in the proof of correctness that because it is not `popped`, `elem` cannot have already been returned by a fast `Pop`, so legality is maintained. Additionally, let `p_i` be the process that invoked `pop_2`. If `elem` is not the top element, then the top element, being `popped`, must have been returned by a fast `Pop` whose timestamp will be earlier than that of the next `Pop` invoked by `p_i`. That fast `Pop` resolves the previous `Pops`, and thus `p_i` is permitted to invoke `l_s` more fast `Pops`.

The reasoning above for `pop_2` also holds for any later-timestamp slow `Pop` not mutually active with `pop_2`, which we call `pop_3`, and so on by induction.

**Algorithm 3** Pseudocode for each process `p_i` implementing a Lateness `k`-Relaxed Stack, where `k \geq n`.

1: **HandleEvent** `PUSH(val)`
2:   `send (push, val, \{localTime, i\})` to all
3:   `setTimer(\epsilon, (push, null, \{localTime, i\}), respond)`

4: **HandleEvent** `POP`
5:   **if** `localPopsInvoked < l_s` **then**
6:     `localPopsInvoked++`
7:     `let ret := lStack.peekByLabel(p_i)`
8:     `ret.popped := true`
9:     `send (pop_f, ret, \{localTime, i\})` to all
10:    `setTimer(\epsilon, (pop_f, ret, null), respond)`
11:  **else** `send (pop_s, null, \{localTime, i\})` to all
12: **HandleEvent** `RECEIVE (op, val, ts)` **FROM** `p_j`
13:   `Pending.insert((op, val, ts))`
setTimer(2u, ⟨op, val, ts⟩, execute)

if op == pop.f then val.popped := true

HandleEvent EXPIRE_TIMER((op, val, ts), respond)

if op == pop.f then return val
else return ACK

HandleEvent EXPIRE_TIMER((op, val, ts), execute)

while ts ≥ Pending.min() do

⟨op′, val′, ts′⟩ := Pending.extractMin()

executeLocally(op′, val′, ts′)

cancelTimer((op′, val′, ts′), execute)

function EXECUTE_LOCALLY(op, val, ⟨∗, j⟩)

if op == push then

lStack.topAvailableByLabel(pi).canPopByLabel := true
lStack.push(val)
lStack.label(argmin ph |lStack.setByLabel(ph)|, val)
lStack.topAvailableByLabel(pi).canPopByLabel := false

else

if op == pop.f then

if j == i then localPopsExecuted++

if val == lStack.getFromTop(1) then

localPopsInvoked ← localPopsExecuted
localPopsExecuted := 0

lStack.remove(val)

else

if localPopsExecuted == ls then

localPopsInvoked ← localPopsExecuted
localPopsExecuted := 0

let ret := lStack.pop()

if ret.label == pi then

lStack.peekByLabel(pi).canPopByLabel := false

else

if j == i then

localPopsInvoked++

localPopsExecuted++

let ret := lStack.popByLabel(pj)

if j == i then return ret
A.1.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance $op$ given as the first argument in line 2, 9, or 11.

**Construction 2.** Define the permutation $\pi$ of operation instances in a complete, admissible run of Algorithm 3 as the order given by sorting by $ts(op)$ for each operation instance $op$.

In Algorithm 3, the pending delay is $2u$ (line 14); we show that Lemma 1 holds.

**Lemma 25.** Each process running Algorithm 3 locally executes all Pushes and Pops in timestamp order.

_Proof._ Since $2u > u + \epsilon$, Lemma 1 holds for Algorithm 2.

The proof of the following fact is the same as in Algorithm 2.

**Lemma 26.** $\pi$ respects real-time order of non-overlapping operation instances.

In the remainder of the proof of correctness, we give a useful lemma and then show inductively that the local execution of each operation type satisfies the invariants defined thus far and maintains a legal linearization, thereby implementing a Lateness $k$-Relaxed Stack according to Definition 2.

**Lemma 27.** In each process, $localPopsInvoked \geq localPopsExecuted$.

_Proof._ $localPopsInvoked$ and $localPopsExecuted$ are both initialized to 0. $localPopsExecuted$ is incremented only at the local execution of a Pop (lines 32 and 47). Whenever $localPopsExecuted$ is incremented, $localPopsInvoked$ has also been incremented at either the invocation (line 6) or local execution (line 46) of the same Pop.

Also, whenever $localPopsInvoked$ is decreased (lines 34 and 39), $localPopsExecuted$ is decreased by the same amount (lines 35 and 40).

**Lemma 28.** Every Pop returns a unique element.
Proof. Fast Pops as well as slow Pops when localPopsExecuted < l return only elements sharing the same label as the invoking process, and fast Pops set their returned elements’ popped fields to true (the aforementioned Pops rely on the lStack functions peekByLabel() and popByLabel()), while slow Pops remove returned elements from the same process’ local stack.

The only time processes return elements not necessarily labeled with their own ids is during slow Pops when localPopsExecuted = l (line 38). Such a slow Pop, pop₂, cannot be mutually active with a preceding slow Pop in which localPopsExecuted = l, because the latter would have set localPopsExecuted := 0 in all processes, and no fast Pops could have been invoked in between. If there is no preceding slow Pop not mutually active with pop₂, then assuming the headmost invariant holds, pop₂ returns an element with canPopByLabel = false, which cannot have been returned by any of the Pops mentioned above.

Otherwise, there exists a preceding slow Pop not mutually active with pop₂, the latest of which we call pop₁. The local execution of pop₁ removes some element with canPopByLabel = false, and sets canPopByLabel = false for the topmost remaining non-popped element elem sharing the same label, which we call pᵢ. Any popped elements above elem labeled pᵢ, if they exist, must have been claimed by fast Pops invoked earlier in real time than the local execution of pop₁ in pᵢ; let popᴺ denote the latest such fast Pop. Because local execution differs in real time between processes by at most u, pop₂ may have been invoked in its own process earlier than popᴺ by any amount less than u in real time. However, when the message sent by popᴺ is received in any process, which takes at most d real time and hence takes place less than d + u real time after pop₂ is invoked, it marks the returned element popped. pop₂ executes locally in each process at a later real time, \((d - u) + 2u = d + u\) after pop₂ is invoked at the earliest.

Because pop₂ removes the topmost non-popped element at local execution (line 41), it will not return the same element as popᴺ or any of the earlier fast Pops, if they exist. □
**Theorem 29.** For any execution of Algorithm 3, the permutation $\pi$ given by Definition 2 is legal by the specification of a Lateness $k$-Relaxed Stack. Thus, Algorithm 3 is a correct implementation of a Lateness $k$-Relaxed Stack. □

**Proof.** We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that $\pi$ is legal, and also maintains (b) the labeling invariant and (c) the headmost invariant. (b) and (c) are demonstrated by induction on the local stack, with the base case being that the invariants are vacuously satisfied when the stack is empty.

**Push:**

(a) As long as every value is unique, (C2) in Definition 1 is satisfied and hence any Push is legal.

(b) If every element is labeled, then since the newly Pushed element is labeled also (line 28), the labeling invariant holds.

(c) In each process $p_i$, the topmost non-popped element labeled $p_i$ before the Push has $canPopByLabel = false$, assuming the headmost invariant holds. Following the Push, the new topmost non-popped element labeled $p_i$ has $canPopByLabel$ set to $false$ (line 29). If the previous topmost non-popped element labeled $p_i$ was distinct, then it has $canPopByLabel$ set to $true$ (line 26).

**Pop:**

(a) We use the fact that every process executes operation instances in linearized order. First, by Lemma 28, Pops return unique elements, so they match unmatched Pushes. We now show that Pops return values maintaining the legality of the linearization. An invoked Pop is fast when $localPopsInvoked < l_s$ (line 5), and as a result it increments $localPopsInvoked$ (line 6) so that the sum of $localPopsInvoked$ among all processes is at most $l_s n < k$. Any fast Pop also increments $localPopsExecuted$ at local execution. On the other hand, an invoked Pop, pop, is slow when
localPopsInvoked = l_s. This case is covered as follows.

If localPopsExecuted = l_s at pop’s local execution, then pop is not mutually active with a preceding slow Pop in which localPopsExecuted = l_s, because such a slow Pop would set localPopsExecuted := 0 in all processes (line 40). pop returns the topmost non-popped element elem, and either elem is the top element or a fast Pop returned the top element; either case resolves all the previously locally executed Pops, so it is justified to decrease localPopsInvoked and localPopExecuted by the same amount such that localPopExecuted = 0.

Otherwise, if localPopsExecuted ≠ l_s at pop’s local execution, then localPopsExecuted = 0 because pop must be mutually active with a preceding slow Pop in which localPopsExecuted = l_s. By the previous paragraph, pop must have resolved all the unresolved Pops invoked by the same process as pop, so pop may return a non-top element (line 48) before incrementing both localPopsInvoked and localPopsExecuted.

Because the number of unresolved instances is always less than k, Pops maintain (C3) in Definition 3.

(b) Because elements are never unlabeled, the labeling invariant still holds.

(c) By the headmost invariant, only a locally executing slow Pop when localPopsExecuted = l_s removes an element with canPopByLabel = false in any process. In the same process, the Pop sets the new topmost non-popped element’s canPopByLabel field to false (line 43), maintaining the headmost invariant.

A.1.2 Performance

An advantage of the Lateness k-Relaxed Stack implementation is that all Pushes are fast, which we state below.
Theorem 30. The worst-case and amortized time complexities of Push among all complete, admissible executions of Algorithm 3 is no more than $\epsilon$. □

Theorem 31. The amortized time complexity of Pop among all complete, admissible executions of Algorithm 3 is no more than $(d + u - \epsilon)/l_s + \epsilon$. □

Proof. We consider the view of a single process $p_i$ in which slow Pops occur. The first slow Pop occurs when $localPopsInvoked$ has been incremented to $l_s$ by the same number of fast Pops. If the slow Pop is mutually active with another slow Pop in whose process $localPopsExecuted = l_s$, then before the slow Pop, $localPopsExecuted = 0$ in $p_i$ and afterward, $localPopsInvoked$ and $localPopsExecuted$ become 1 (lines 46-47). In any other case, $localPopsInvoked$ and $localPopsExecuted$ become 0 in $p_i$. This means at least $l_s - 1$ subsequent Pops, if they exist, will be fast, and this pattern repeats.

Since each fast Pop takes $\epsilon$ time and the slow Pop takes $d + u$ time, the average cost of each repetition of the pattern of $l_s - 1$ fast Pops and 1 slow Pop is

$$\frac{(l_s - 1)\epsilon + (d + u)}{l_s} = \frac{d + u - \epsilon + l_s \epsilon}{l_s} = \frac{d + u - \epsilon + \epsilon}{l_s} + \epsilon.$$

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow Push which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of Pushes at any process $p_i$, it is also an upper bound on the average cost of Pushes at all processes; hence, it is the amortized cost. □

A.2 Windowed Relaxed Stacks

Algorithm 4 implements the Windowed $k$-Relaxed Stack algorithm. Because it is nearly completely a combination of the Out-of-Order $k$-Relaxed Stack algorithm (Algorithm 2)
and the Lateness $k$-Relaxed Stack algorithm (Algorithm 3), it is recommended that Algorithm 4 be compared side-by-side with those algorithms.

Having $unresolved = true$ signifies that an element has already been removed but is not yet resolved, which defines a class of elements that is essential to Definition 6.

A slow $Pop$ always returns and removes the top element at its local execution. It then proceeds to remove all the unresolved elements one-by-one while relabeling using the same method found in Algorithm 2.

Similar to Algorithm 3, this algorithm uses the $localPopsInvoked$ and $localPopsExecuted$ local variables, which here are necessary to limit the number of $Pops$ returning $\perp$, because they qualify as unresolved $Pops$ according to Definition 6.

**Algorithm 4** Pseudocode for each process $p_i$ implementing a Windowed $k$-Relaxed Stack, where $k \geq n$.

1: **HandleEvent** $PUSH(val)$
2:   **if** $localFastPushesActive < \tau_c - 1$ **then**
3:     send $(push_f, val, \langle localTime, i \rangle)$ to all
4:     setTimer($\epsilon, \langle push_f, null, \langle localTime, i \rangle \rangle$, respond)
5:   **else**
6:     send $(push_s, val, \langle localTime, i \rangle)$ to all
7:     setTimer($\epsilon, \langle push_s, 1, \langle localTime, i \rangle \rangle$, respond)

9: **HandleEvent** $POP$
10:   **if** $(lStack.peekByLabel(p_i, safeRegion()) \neq \perp$ or $lStack.size() < k - \tau n)$
11:     and $localPopsInvoked < l_s$ **then**
12:     $localPopsInvoked++$
13:     let $ret := lStack.peekByLabel(p_i, safeRegion())$
14:     $ret.popped := true$
15:     send $(pop_f, ret, \langle localTime, i \rangle)$ to all
16:     setTimer($\epsilon, \langle pop_f, ret, null \rangle$, respond)
17:   **else** send $(pop_s, null, \langle localTime, i \rangle)$ to all
18: **HandleEvent** $RECEIVE (op, val, ts)$ FROM $p_j$
19:     Pending.insert$(\langle op, val, ts \rangle)$
20:     setTimer($d + u, \langle op, val, ts \rangle$, execute)
21:   **if** $op == pop_f$ or $op == pop_s$ **then**
22:     $popsInPending++$
if \( op == \text{pop}\_f \) then
\[
\text{val.popped} := \text{true}
\]
resetVolatility(val)

HandleEvent \text{EXPIRE}\text{TIMER}((op, val, (\ast, j)), respond)
if \( op == \text{pop}\_f \) then return val
else if \( op == \text{push}\_s \) and localFastPushesActive == \( \tau_c - 1 \) then
setTimer(min\{\( \epsilon, 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon \}), \langle \text{push}\_s, val + 1, (\ast, j) \rangle, \text{respond})
else return ACK

HandleEvent \text{EXPIRE}\text{TIMER}((op, val, ts), execute)
while \( ts \geq \text{Pending.min}() \) do
\( \langle \text{op}', \text{val}', ts' \rangle := \text{Pending.extractMin}() \)
executeLocally(\text{op}', \text{val}', ts')
cancelTimer(\langle \text{op}', \text{val}', ts' \rangle, \text{execute})

function \text{EXECUTE}\text{LOCALLY}(\text{op}, \text{val}, (\text{time}, j))
if \( \text{op} == \text{push}\_f \) or \( \text{op} == \text{push}\_s \) then
\( \text{lStack.topAvailableByLabel(p_i).canPopByLabel} := \text{true} \)
\( \text{lStack.push(val)} \)
if \( \text{lStack.size()} \leq k - \tau n \) then
\( \text{lStack.label}(\arg \min \{ \text{lStack.setByLabel(p_h) \cap safe\,Region()} \}, \text{val}) \)
else
\( \text{lStack.label(lStack.getFromTop}(k - \tau n + 1).\text{label}, \text{val}) \)
setVolatility(elem, k - \tau n + 1)
\( \text{lStack.label(null, lStack.getFromTop}(k + \min\{\text{lower\,Region()}, \text{pops\,In\,Pending}\} + 1) \)
\( \text{lStack.topAvailableByLabel(p_i).canPopByLabel} := \text{false} \)
if \( j == i \) and \( \text{op} == \text{push}\_f \) then
localFastPushesActive--
else
if \( \text{op} == \text{pop}\_f \) then
if \( j == i \) then localPopsExecuted++
if \( \text{val} == \text{lStack.getFromTop}(1) \) then
localPopsInvoked -- localPopsExecuted
localPopsExecuted := 0
let ret := val
else
if \( \text{localPopsExecuted} == l_s \) then
localPopsInvoked -- localPopsExecuted
localPopsExecuted := 0
let ret := lStack.peak()

else
  if j == i then
    localPopsInvoked++
    localPopsExecuted++
    let ret := lStack.peekByLabel(pj)

    if j == i then return ret

  ret.popped := true
  ret.unresolved := true
  if !ret.canPopByLabel and ret.label == pi then
    lStack.peekByLabel(pi).canPopByLabel := false
  if ret.higher != null then resetVolatility(ret.higher)
  popsInPending--
  if ret == lStack.getFromTop(1) then
    while lStack has unresolved elements do
      let toRemove := the topmost unresolved element
      if toRemove ∈ safeRegion() then
        if volatileRegion() ∩ lStack.topSet(k − τn) ≠ ∅ then
          let toRelabel := lStack.peekBySet(volatileRegion())
          resetVolatility(toRelabel)
        else let toRelabel := lStack.getFromTop(k − τn + 1)
          lStack.label(j, toRelabel)
      lStack.remove(toRemove)
      if lStack.labeledSize() < k then
        lStack.label(dummy, lStack.peekByLabel(null))
  return [topSet(k − τn) ∪ topSetBySet(popsInPending, lowerRegion())] \ topSetBySet(popsInPending, volatileRegion())

function safeRegion
  return {elem ∈ lStack.topSet(k − τn) : elem.tvolatile > localTime − (2d + 2u + ϵ)}

function lowerRegion
  return {elem.lower : elem ∈ volatileRegion()}

function setVolatility(elem, other)
  elem.tvolatile := localTime
  elem.lower := other
  elem.lower.higher := elem

function resetVolatility(elem)
A.2.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance $op$ given as the first argument in line 3, 7, 14, or 16.

**Construction 3.** Define the permutation $\pi$ of operation instances in a complete, admissible run of Algorithm 4 as the order given by sorting by $ts(op)$ for each operation instance $op$.

First, the following two lemmas have the same proofs as in Algorithm 3.

**Lemma 32.** Each process running Algorithm 4 locally executes all Pushes and Pops in timestamp order.

**Lemma 33.** $\pi$ respects the real-time order of non-overlapping operation instances.

The lemmas up to Lemma 40 are the same as for Algorithm 2, and their proofs are identical.

**Lemma 34.** There are at most $\tau_c$ mutually active Pushes occurring in a single process and at most $\tau_c n$ mutually active Pushes occurring over all processes at any given time.

**Lemma 35.** The `safeRegion()` function given in Algorithm 4 returns a set of elements that are guaranteed to be safe.

**Definition 21.** The uniformity invariant is given by Definition 12.

**Lemma 36.** If an element elem is the lower element in some (higher, lower)-pair, then the value of elem.higher is the other element; if not, elem.higher is null. Likewise, if elem is the higher element in some (higher, lower)-pair, then the value of elem.lower is the other element, and if not, elem.lower is null.
Lemma 37. The volatility invariant holds, and states that in the local stack of each process,

(i) any two elements that form a (higher, lower)-pair share the same label;

(ii) if an element elem is the topmost in the volatile region, then elem.lower is currently
the \((k - \tau n + 1)\)th-topmost element, so that any element is the \(m\)th-topmost volatile
element if and only if it is the higher of a (higher, lower)-pair in which lower is the
\(m\)th-topmost element in lowerRegion();

(iii) the lower region is mutually exclusive with the topmost \(k - \tau n\) elements and con-

ected; and

(iv) member elements of a (higher, lower)-pair cannot mutually exist in the safe region.

Corollary 38. The size of the safe region is always \(\min\{k - \tau n, lStack.size()\}\).

Corollary 39. Incrementing or decrementing popsInPending preserves the uniformity
invariant.

Remark 4. popsInPending is incremented by 1 (line 21) every time the process receives
a message sent from any Pop and is decremented by 1 (line 71) every time any Pop exe-
cutes locally, which means this variable counts the number of active Pops from which this
process received a message.

Definition 22. The labeling invariant is given by Definition 16.

Lemma 40. Any element elem returned by a Pop will not be relabeled with another pro-
cess id unless elem.poppedState = executed in all processes.

Proof. The proof is identical to Lemma 10 except with one difference in Algorithm 4 being
that not all Pops remove their returned element elem at local execution, but they do set
elem.poppedState := executed.

Additionally, the following lemmas are the same as for Algorithm 3, and their proofs
are identical.
Lemma 41. In each process, \( \text{localPopsInvoked} \geq \text{localPopsExecuted} \).

Lemma 42. Every Pop returns a unique element.

Now we demonstrate that the linearization given by the algorithm is a legal sequence of operation instances according to Definition 6.

Theorem 43. For any execution of Algorithm 4, the permutation \( \pi \) given by Definition 3 is legal by the specification of a Windowed \( k \)-Relaxed Stack. Thus, Algorithm 4 is a correct implementation of a Windowed \( k \)-Relaxed Stack. \( \square \)

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that \( \pi \) is legal, and also maintains (b) the labeling invariant, (c) the uniformity invariant, and (d) the headmost invariant.

Since every process locally executes operation instances in linearized order, elements whose unresolved field is true correspond to unresolved elements in Definition 6, because they have been returned by Pops which have since executed locally (line 67), and they are removed by locally executing Pops which remove the top element (lines 73-83).

Along with the fact that not every locally executing Pop removes elements but those that do perform the same procedure as in Algorithm 2 albeit multiple times, the above shows that the reasoning for (a), (b), and (c) is identical to the proof of Theorem 12, except for (a) under Pop in two cases:

- If a fast Pop, \( \text{pop} \), returns \( \perp \) then we still have \( \lambda(\rho \cdot \text{pop}) = \lambda(\rho) + 1 \). We re-introduce \( \text{localPopsInvoked} \) and \( \text{localPopsExecuted} \) from Algorithm 3 to ensure that \( \lambda(\rho \cdot \text{pop}) < k \), and the proof that it does so is the same as before.

- Slow Pops behave similarly to and follow the same reasoning for legality as in the proof of Theorem 29.

Additionally, the reasoning for (d) is the same as that given in the proof of Theorem 29. \( \square \)
A.2.2 Performance

The mechanism for \textit{Push} and hence its upper bound proofs are the same as they were in Algorithm 2.

\textbf{Theorem 44.} The worst-case time complexity of \textit{Push} among all complete, admissible executions of Algorithm 4 is no more than \(\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}\). \hfill \Box

\textbf{Theorem 45.} The amortized time complexity of \textit{Push} among all complete, admissible executions of Algorithm 4 is no more than \(\max\{2d/\tau_c, \epsilon\}\). \hfill \Box

This is the first algorithm we encounter in which the headmost invariant has a slight impact on the amortized result.

\textbf{Theorem 46.} The amortized time complexity of \textit{Pop} among all complete, admissible executions of Algorithm 4 is no more than \((2d - \epsilon)\min\{l_s, l - \tau\} + \epsilon\) if \(\tau < l\), or \(2d\) otherwise. \hfill \Box

\textit{Proof.} First we assume \(\tau < l\), and consider the view of a single process \(p_i\) in which a slow \textit{Pop} occurs, which is either when \textit{localPopsInvoked} has been incremented to \(l_s\) by the same number of fast \textit{Pops} or when all the elements labeled \(p_i\) that are not the top element have been returned in a fast \textit{Pop} and \textit{lStack.size()} \(\leq k - \tau n\). Considering the proof of Theorem 31, at least \(l_s - 1\) fast \textit{Pops} occur before a slow \textit{Pop} in the former. In the latter, at least \(l - \tau - 1\) fast \textit{Pops} occur. Since each fast \textit{Pop} takes \(\epsilon\) time and the slow \textit{Pop} takes \(2d\) time, the average cost of each repetition of a pattern of \(\min\{l_s - 1, l - \tau - 1\}\) fast \textit{Pops} and one slow \textit{Pop} is

\[
\frac{\min\{l_s - 1, l - \tau - 1\}\epsilon + 2d}{\min\{l_s, l - \tau\}} = \frac{\min\{l_s, l - \tau\}\epsilon + 2d - \epsilon}{\min\{l_s, l - \tau\}} = \frac{2d - \epsilon}{\min\{l_s, l - \tau\}} + \epsilon.
\]

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow \textit{Pop} which ends a pattern have the highest average cost.
Otherwise, if \( \tau = l \), then the size of the safe region is smaller than \( n \), so there may be no elements labeled \( p_i \). Then every \( Pop \) is slow, taking \( 2d \) time.

In either case, because this provides an upper bound on the cost of \( Pops \) at any process \( p_i \), it is also an upper bound on the average cost of \( Pops \) at all processes; hence, it is the amortized cost.

\[ \square \]

### A.3 Restricted Relaxed Stacks

Algorithm 5 implements the Restricted \( k \)-Relaxed Stack (Definition 5), which should be compared side-by-side with both Algorithm 2 and Algorithm 4. The Restricted \( k \)-Relaxed Stack has the same upper bounds as the Windowed \( k \)-Relaxed Stack.

Because operation instances execute in linearized order, the requirement of Definition 5 is that only the topmost \( k - \lambda(\rho) \) elements are legal to return, where \( \rho \) is the linearization of operation instances that have executed locally. To obtain the functionality of allowing only these elements to be returned, elements are no longer relabeled every time the local execution of a fast \( Pop \) removes an element, so each one reduces the number of labeled elements.

Furthermore, we employ the local variable \( kMinusLateness \) to keep track of the quantity \( k - \lambda(\rho) \). In almost all instances \( k \) was used in Algorithm 2, \( k \) is now replaced with \( kMinusLateness \).

Thus, Algorithm 5 has weaker uniformity invariant since the size of the safe region may be smaller. More specifically, the first part still always holds, but the second part holds in fewer situations.

A slow \( Pop \) always removes the top element, which according to the definition must reset the lateness. Thus, the local execution of slow \( Pops \) labels elements until the size of the safe region is \( \min\{k - \tau n, lStack.size()\} \) and the number of labeled elements is \( \min\{k, lStack.size()\} \).
Algorithm 5 Pseudocode for each process $p_i$ implementing a Restricted $k$-Relaxed Stack, where $k \geq n$.

1: **HandleEvent** PUSH($val$)
2:   if $localFastPushesActive < \tau_c - 1$ then
3:     send ($push_f, val, \langle localTime, i \rangle$) to all
4:     setTimer($\epsilon, \langle push_f, null, \langle localTime, i \rangle \rangle$, respond)
5:     $localFastPushesActive$++
6: else
7:     send ($push_s, val, \langle localTime, i \rangle$) to all
8:     setTimer($\epsilon, \langle push_s, 1, \langle localTime, i \rangle \rangle$, respond)
9: **HandleEvent** POP
10:   if ($lStack.peekByLabel(p_i, safeRegion()) \neq \bot$ or $lStack.size() < kMinusLateness - \tau n$) and $localPopsInvoked < l_s$ then
11:     $localPopsInvoked$++
12:     let $ret := lStack.peekByLabel(p_i, safeRegion())$
13:     $ret.popped := true$
14:     send ($pop_f, ret, \langle localTime, i \rangle$) to all
15:     setTimer($\epsilon, \langle pop_f, ret, null \rangle$, respond)
16: else send ($pop_s, null, \langle localTime, i \rangle$) to all
17: **HandleEvent** RECEIVE ($op$, $val$, $ts$) FROM $p_j$
18:     $Pending.insert((op, val, ts))$
19:     setTimer($d + u$, ($op, val, ts$), execute)
20:   if $op == pop_f$ or $op == pop_s$ then
21:     $popsInPending$++
22:   if $op == pop_f$ then
23:     $val.popped := true$
24:     resetVolatility($val$)
25: **HandleEvent** EXPIRETimer($\langle op, val, \langle *, j \rangle \rangle$, respond)
26:   if $op == pop_f$ then return $val$
27: else if $op == push_s$ and $localFastPushesActive == \tau_c - 1$ then
28:     setTimer($\min(\epsilon, 2d - (\tau_c - 1)\epsilon - val \cdot \epsilon), \langle push_s, val + 1, \langle *, j \rangle \rangle$, respond)
29: else return ACK
30: **HandleEvent** EXPIRETimer($\langle op, val, ts \rangle$, execute)
31:   while $ts \geq Pending.min()$ do
32:     $\langle op', val', ts' \rangle := Pending.extractMin()$
33:     executeLocally($op', val', ts'$)
34:     cancelTimer($\langle op', val', ts' \rangle$, execute)
35: function EXECUTELocally($op, val, \langle time, j \rangle$)
if \( op == \text{push}_f \) or \( op == \text{push}_s \) then

\[ \text{lStack.topAvailableByLabel}(p_i).\text{canPopByLabel} := \text{true} \]

\[ \text{lStack.push}(\text{val}) \]

if \( \text{lStack.size()} \leq k\text{MinusLateness} - \tau n \) then

\[ \text{lStack.label}((\text{argmin}_{p_h} [\text{lStack.setByLabel}(p_h) \cap \text{safeRegion()}] \text{[val]}) \]

else

\[ \text{lStack.label}(\text{lStack.getFromTop}(k\text{MinusLateness} - \tau n + 1).\text{label}, \text{val}) \]

setVolatility(\text{val}, \text{lStack.getFromTop}(k\text{MinusLateness} - \tau n + 1))

\[ \text{lStack.label}([\text{null}, \text{lStack.getFromTop}(k\text{MinusLateness} + \min\{|\text{lowerRegion()}|, \text{popsInPending}\} + 1)] \]

\[ \text{lStack.topAvailableByLabel}(p_i).\text{canPopByLabel} := \text{false} \]

if \( j == i \) and \( op == \text{push}_f \) then

localFastPushesActive--

else

if \( op == \text{pop}_f \) then

let hasTop := (\text{val} == \text{lStack.getFromTop}(1))

if \( j == i \) then localPopsExecuted++

if hasTop then

localPopsInvoked -- localPopsExecuted

localPopsExecuted := 0

\[ \text{lStack.remove}(\text{val}) \]

else

if \( \text{lStack.pop}() \) == \( p_i \) then

\[ \text{lStack.peekByLabel}(p_i).\text{canPopByLabel} := \text{false} \]

else

let hasTop := false

if \( j == i \) then

localPopsInvoked++

localPopsExecuted++

let ret := \text{lStack.popByLabel}(p_j)

if \( j == i \) then return ret

if \( \text{ret.higher} \neq \text{null} \) then resetVolatility(ret.higher)

k\text{MinusLateness}--
A.3.1 Correctness

Let $ts(op)$ denote the timestamp associated with an operation instance $op$ given as the first argument in line 3, 7, 14, or 16.

Construction 4. Define the permutation $\pi$ of operation instances in a complete, admissible run of Algorithm 5 as the order given by sorting by $ts(op)$ for each operation instance $op$. 
First, the following two lemmas have the same proofs as in Algorithm 3.

**Lemma 47.** Each process running Algorithm 5 locally executes all Pushes and Pops in timestamp order.

**Lemma 48.** \( \pi \) respects the real-time order of non-overlapping operation instances.

We proceed in the same manner as in proving the correctness of Algorithm 2, but a major difference is that \( k \) is now replaced with \( k_{\text{MinusLateness}} \). Otherwise, the proofs not explicitly stated are identical as before.

**Lemma 49.** There are at most \( \tau_c \) mutually active Pushes occurring in a single process and at most \( \tau_c n \) mutually active Pushes occurring over all processes at any given time.

**Lemma 50.** The \( \text{safeRegion()} \) function given in Algorithm 5 returns a set of elements that are guaranteed to be safe.

**Definition 23.** The uniformity invariant states that

\[
|lStack.setByLabel(p_i) \cap \text{safeRegion()}| \leq \left\lceil \frac{k}{n} \right\rceil - \tau \quad \forall i \in [0, n - 1]
\]

with the number of \( i \) such that the bound is reached being at most \( n_a := k - (\lceil k/n \rceil - 1)n \), and when \( lStack.size() \geq k - \tau n \) with \( k_{\text{MinusLateness}} = k \),

\[
|lStack.setByLabel(p_i) \cap \text{safeRegion()}| - |lStack.setByLabel(p_j) \cap \text{safeRegion()}| \leq 1
\]

\( \forall i, j \in [0, n - 1] \),

**Lemma 51.** If an element \( \text{elem} \) is the lower element in some \((\text{higher}, \text{lower})\)-pair, then the value of \( \text{elem.higher} \) is the other element; if not, \( \text{elem.higher} \) is null. Likewise, if \( \text{elem} \) is the higher element in some \((\text{higher}, \text{lower})\)-pair, then the value of \( \text{elem.lower} \) is the other element, and if not, \( \text{elem.lower} \) is null.
While the statement of the volatility invariant is still similar as before, its proof has differences with that of Lemma 7.

**Lemma 52.** The volatility invariant holds, and states that in the local stack of each process,

(i) any two elements that form a \((\text{higher, lower})\)-pair share the same label;

(ii) if an element \(\text{elem}\) is the topmost in the volatile region, then \(\text{elem.lower}\) is currently the \((k - \text{Lateness} - \tau n + 1)\)th-topmost element, so that any element is the \(m\)th-topmost volatile element if and only if it is the higher of a \((\text{higher, lower})\)-pair in which \(\text{lower}\) is the \(m\)th-topmost element in \(\text{lowerRegion}\);

(iii) the lower region is mutually exclusive with the topmost \(k - \text{Lateness} - \tau n\) elements and connected; and

(iv) member elements of a \((\text{higher, lower})\)-pair cannot mutually exist in the safe region.

**Proof.** We refer to the proof of Lemma 7. The reasoning still holds, because although \(k - \text{Lateness}\) is not constant, locally executing fast \(\text{Pops}\) only decrease this quantity and move all the existing \(\text{lower}\) elements of \((\text{higher, lower})\)-pairs closer to the top, so they maintain the constraints given the new \(k - \text{Lateness}\) value.

The local execution of a slow \(\text{Pop}\), which labels up to \(k\) elements, marks all the elements in the topmost \(k - \text{Lateness} - \tau n\) non-volatile (lines 78-79), making them no longer part of some \((\text{higher, lower})\)-pair and hence vacuously satisfying the contraints. Volatile elements outside the topmost \(k - \text{Lateness} - \tau n\) are unchanged.  

**Corollary 53.** The size of the safe region is always \(\min\{k - \text{Lateness} - \tau n, l\text{Stack.size()}\}\).

**Corollary 54.** Incrementing or decrementing \(\text{popsInPending}\) preserves the uniformity invariant.
Remark 5. $\text{popsInPending}$ is incremented by 1 (line 21) every time the process receives a message sent from any $\text{Pop}$ and is decremented by 1 (line 73) every time any $\text{Pop}$ executes locally, which means this variable counts the number of active $\text{Pops}$ from which this process received a message.

Definition 24. The labeling invariant states that the labeled elements in the local stack are the topmost $\text{kMinusLateness} + \min\{|\text{lowerRegion()}|, \text{popsInPending}\}$ elements.

Lemma 55. Any element $\text{elem}$ returned by a $\text{Pop}$ will not be relabeled with another process id.

Proof. Only slow $\text{Pops}$ relabel elements, and by the time a slow $\text{Pop}$ does so at local execution, all other invoked $\text{Pops}$ with earlier timestamps will have already executed locally, removing their returned elements.

As before, the following lemmas are the same as for Algorithm 3, and their proofs are identical.

Lemma 56. In each process, $\text{localPopsInvoked} \geq \text{localPopsExecuted}$.

Lemma 57. Every $\text{Pop}$ returns a unique element.

Now we demonstrate that the linearization given by the algorithm is a legal sequence of operation instances according to Definition 5.

Theorem 58. For any execution of Algorithm 5, the permutation $\pi$ given by Definition 4 is legal by the specification of a Restricted $k$-Relaxed Stack. Thus, Algorithm 5 is a correct implementation of a Restricted $k$-Relaxed Stack.

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm (a) generates return values such that $\pi$ is legal, and also maintains (b) the labeling invariant, (c) the uniformity invariant,
and (d) the headmost invariant. The invariants are demonstrated by induction on the local stack, with the base case being that they are vacuously satisfied when the stack is empty.

The reasoning for (a), (b), and (c) is for the most part identical to that found in the proof of Theorem 12 except with $k$ replaced by $k\text{MinusLateness}$, and the reasoning for (d) is the same as that given in the proof of Theorem 29. We list only the differences below, letting $\rho$ be the linearization of all the operation instances with earlier timestamps.

**Pop:**

(a) The reasoning for fast $\text{Pops}$ is the same as in (a) under $\text{Pop}$ in the proof of Theorem 12, except with $k$ replaced by $k\text{MinusLateness}$ as expected, as well as $\text{localPopsInvoked}$ and $\text{localPopsExecuted}$ again introduced from Algorithms 3 and 4 to limit the number of fast $\text{Pops}$ returning $\bot$, the proof that it does so being the same as before.

\[
k\text{MinusLateness} = k - \lambda(\rho)
\]

because locally executing $\text{Pops}$ that do not return the top element (so that they are not headmost according to Definition 5) decrement $k\text{MinusLateness}$, and those that do (so that they are headmost) reset $k\text{MinusLateness}$ to $k$.

Even though $k\text{MinusLateness}$ is not constant, the local execution of a mutually active $\text{Pop}$ decreases $k\text{MinusLateness}$ also removes an element from the $k\text{MinusLateness}$ topmost, so any other elements that were in the topmost $k\text{MinusLateness}$ are still legal to return.

The reasoning for slow $\text{Pops}$ behaving legally is the same as in Algorithm 29.

(b) Each locally executing $\text{Pop}$ that does not return the top element decreases $k\text{MinusLateness}$ simultaneously while removing the element, and each locally executing $\text{Pop}$ that returns the top element relabels $k - k\text{MinusLateness}$ elements (lines 81-82) and resets $k\text{MinusLateness} = k$, maintaining the labeling invariant.

(c) If the $\text{Pop}$ does not return the top element, the first part of the uniformity invariant
holds since no elements are relabeled, and the second part does not apply because \(kMinusLateness < k\).

If it returns the top element, then it chooses the same new labels for elements (line 76) as done by \(Pushes\), so the proof for (c) under \(Push\) in the proof of Theorem 12 applies here and implies that both the first and second uniformity invariants are satisfied.

\[\square\]

A.3.2 Performance

The mechanism for \(Push\) and hence its upper bound proofs are the same as they were in Algorithm 2.

**Theorem 59.** The worst-case time complexity of \(Push\) among all complete, admissible executions of Algorithm 5 is no more than \(\max\{2d + (1 - \tau_c)\epsilon, \epsilon\}\).

\[\square\]

**Theorem 60.** The amortized time complexity of \(Push\) among all complete, admissible executions of Algorithm 5 is no more than \(\max\{2d/\tau_c, \epsilon\}\).

\[\square\]

This is the first algorithm we encounter in which the headmost invariant has a slight impact on the amortized result.

**Theorem 61.** The amortized time complexity of \(Pop\) among all complete, admissible executions of Algorithm 5 is no more than \((2d - \epsilon)/(\min\{l_s, l - \tau\} + \epsilon)\) if \(\tau < l\), or \(2d\) otherwise.

\[\square\]

**Proof.** The proof is similar to that of Theorem 46, and gives the same time complexity.

\[\square\]

A.4 Stuttering Relaxed Queues

Algorithm 2 implements the Stuttering \(k\)-Relaxed Queue, which is defined in a corresponding manner to the Stuttering \(k\)-Relaxed Stack (Definition 7).
The algorithm bears similarity to the Lateness $k$-Relaxed Queue algorithm (Algorithm 3), as localDequeuesInvoked and localDequeuesExecuted limit the number of fast Dequeues in each process since the local execution of the previous slow Dequeue in the linearization. It is also in a manner like an opposite, as Dequeues are only allowed to return the top element instead of any element in the local queue.

Only slow Dequeues remove elements from the local queue.

To certify that Dequeues do not return elements that have already been removed according to Definition 7, we prevent fast Dequeues from returning certain elements by marking those elements popped if the fast Dequeues’ timestamps are later by a certain amount than the timestamp of a slow Dequeue returning the same element.

Interestingly, this is the only algorithm that allows for $0 \leq k < n$, and if $k = 0$, it implements an unrelaxed Queue.

Algorithm 6 Pseudocode for each process $p_i$ implementing a Stuttering $k$-Relaxed Queue, where $k \geq 0$.

1: **HandleEvent** ENQUEUE(val)  
2: send (enq, val, ⟨localTime, i⟩) to all  
3: setTimer($\epsilon$, ⟨enq, null, ⟨localTime, i⟩⟩), respond  

4: **HandleEvent** DEQUEUE  
5: if localDeqsInvoked < $l_s$ and lQueue.peek() $\neq \perp$ then  
6: localDeqsInvoked++  
7: while lQueue.peek().tpopped + ($d + \epsilon$) < localTime do  
8: lQueue.peek().popped := true  
9: let ret := lQueue.peek()  
10: setTimer($\epsilon$, ⟨deq_f, ret, null⟩), respond  
11: send (deq_f, null, ⟨localTime, i⟩) to all  
12: else send (deq_s, null, ⟨localTime, i⟩) to all  

13: **HandleEvent** RECEIVE (op, val, ⟨t, j⟩) FROM $p_j$  
14: Pending.insert(⟨op, val, ⟨t, j⟩⟩)  
15: setTimer($u + \epsilon$, ⟨op, val, ⟨t, j⟩⟩), execute  
16: if op == deq_s then $val.tpopped := t$  

17: **HandleEvent** EXPIRETIMER(⟨op, val, ts⟩), respond  
18: if op == deq_f then return $val$  
19: else return ACK
20: \textbf{HandleEvent} \textsc{ExpireTimer}((\textit{op}, \textit{val}, \textit{ts}), \textit{execute})
21: \hspace{1em} \textbf{while} \hspace{0.5em} \textit{ts} \geq \textit{Pending.min}() \textbf{do}
22: \hspace{2em} \langle \textit{op}', \textit{val}', \textit{ts}' \rangle := \textit{Pending.extractMin}()
23: \hspace{2em} \textit{executeLocally}(\textit{op}', \textit{val}', \textit{ts}')
24: \hspace{2em} \textit{cancelTimer}(\langle \textit{op}', \textit{val}', \textit{ts}' \rangle, \textit{execute})
25: \hspace{1em} \textbf{function} \textsc{executeLocally}((\textit{op}, \textit{val}, \langle *, j \rangle))
26: \hspace{2em} \textbf{if} \hspace{0.5em} \textit{op} == \textit{enq} \textbf{then} \hspace{0.5em} \textit{lQueue.enq}()\textit{val}
27: \hspace{2em} \textbf{else if} \hspace{0.5em} \textit{op} == \textit{deq} \textbf{then} \hspace{0.5em} \textbf{localDeqsExecuted}++
28: \hspace{2em} \textbf{else}
29: \hspace{3em} \textbf{localDeqsInvoked} -= \textbf{localDeqsExecuted}
30: \hspace{2em} \textbf{localDeqsExecuted} := 0
31: \hspace{2em} \textbf{let} \textit{ret} := \textit{lQueue.deq}()
32: \hspace{2em} \textbf{if} \hspace{0.5em} j == i \textbf{then} \hspace{0.5em} \textbf{return} \textit{ret}

\textbf{A.4.1 Correctness}

Let $ts(\textit{op})$ denote the timestamp associated with an operation instance $\textit{op}$ given as the first argument in line 2, 10, or 12.

\textbf{Construction 5.} Define the permutation $\pi$ of operation instances in a complete, admissible run of Algorithm 6 as the order given by sorting by $ts(\textit{op})$ for each operation instance $\textit{op}$.

In Algorithm 6, the pending delay is $u + \epsilon$ (line 15), and so Lemma 1 holds.

\textbf{Lemma 62.} Each process running Algorithm 6 locally executes all \textit{Enqueue}s and \textit{Dequeue}s in timestamp order.

The proof of the following lemma is the same as before.

\textbf{Lemma 63.} $\pi$ respects the real-time order of non-overlapping operation instances.

\textbf{Lemma 64.} In each process, $\textbf{localDeqsInvoked} \geq \textbf{localDeqsExecuted}$.

\textbf{Proof.} The proof is similar to that of Lemma 27. \hfill \square
Theorem 65. For any execution of Algorithm 6, the permutation $\pi$ given by Definition 5 is legal by the specification of Stuttering $k$-Relaxed Queue. Thus, Algorithm 6 is a correct implementation of an Stuttering $k$-Relaxed Queue. 

Proof. We consider local execution by operation type, and show that for every sequence of operation instances invoked by the processes, the algorithm generates return values such that $\pi$ is legal.

Enqueue: As long as every value is unique, (C2) in the relaxed queue definition corresponding to Definition 7 is satisfied and hence any Enqueue is legal.

Dequeue: Consider a Dequeue instance, which we call $deq$. The stutter count, or $s(\rho \cdot deq)$ where $\rho$ is the linearization of operation instances, is equivalent to the number of locally executing Dequeues returning without removing the top element (line 40) after the last locally executing Dequeue that returned a distinct element, since every process executes operation instances in linearized order.

The sum of $localDeqsExecuted$ among all processes gives $s(\rho \cdot deq)$ because each fast Dequeue increments $localDeqsExecuted$ in some process (line 27) while each slow Dequeue, which removes the top element, resets $localDeqsExecuted$ in all processes (line 30).

From line 5, the sum of $localDeqsInvoked$ among all processes is at most $l_sn = \lceil k/n \rceil - 1)n < k$. Hence, with the result of Lemma 64, $s(\rho \cdot deq) < k$ also.

Lastly, we must show that a Dequeue cannot return an element higher in the local queue than one that has already been returned, or it would violate the condition that the element is not removed in $\rho$ according to Definition 7.

A Dequeue may only return a different element from any previous Dequeues if it succeeds a slow Dequeue, $deqs$, which removes the top element from the local queue in each process. $deqs$ sends a message to all processes to mark the $t_popped$ field of its returned element with its own timestamp (line 16), which occurs within $d$ real time in all processes. The element returned by $deqs$ will not be removed from the local queue in any process.
before this occurs in all processes, because the earliest it may execute locally in any process is \((d - u) + u + \epsilon = d + \epsilon\).

Since local times differ by at most \(\epsilon\), marking elements \(\text{popped}\) (line 8) if they were returned by slow \(\text{Dequeue}\)s timestamped at least \(d + \epsilon\) earlier guarantees that all preceding \(\text{Dequeue}\)s return the same element, and all succeeding \(\text{Dequeue}\)s return the same element.

\[\square\]

A.4.2 Performance

**Theorem 66.** The worst-case and amortized time complexities of \(\text{Enqueue}\) among all complete, admissible executions of Algorithm 6 is no more than \(\epsilon\).

**Theorem 67.** The worst-case time complexity of \(\text{Dequeue}\) among all complete, admissible executions of Algorithm 6 is no more than \(d + \epsilon\).

**Theorem 68.** The amortized time complexity of \(\text{Dequeue}\) among all heavily loaded, complete, admissible executions of Algorithm 6 is no more than \(d/(l_s + 1) + \epsilon\).

**Proof.** We consider the view of a single process \(p_i\) in which a slow \(\text{Dequeue}\) occurs, which is when \(\text{count}\) has been incremented to \(l_s\) by the same number of fast \(\text{Dequeue}\)s. Since each fast \(\text{Dequeue}\) takes \(\epsilon\) time and the slow \(\text{Dequeue}\) takes \(d + \epsilon\) time, the average cost of each repetition of a pattern of \(l_s\) fast \(\text{Dequeue}\)s and one slow \(\text{Dequeue}\) is

\[
\frac{l_s\epsilon + (d + \epsilon)}{l_s + 1} = \frac{d + (l_s + 1)\epsilon}{l_s + 1} = \frac{d}{l_s + 1} + \epsilon.
\]

The cost of any prefix of the infinite repetition of this pattern is bounded above by the maximum average cost of a single copy of this pattern, since prefixes ending with a slow \(\text{Enqueue}\) which ends a pattern have the highest average cost.

Because this provides an upper bound on the cost of \(\text{Enqueue}\)s at any process \(p_i\), it is also an upper bound on the average cost of \(\text{Enqueue}\)s at all processes; hence, it is the amortized cost.

\[\square\]
Theorem 69. The amortized time complexity of Dequeue among all complete, admissible executions of Algorithm 6 is no more than $d + \epsilon$. □

Proof. The worst case is when the execution has no invoked Enqueue; then all Dequeue are slow, and the amortized time complexity is the same as the worst-case time complexity, $d + \epsilon$. □
APPENDIX B
SAMPLE OF DERIVED ALGORITHMS

See Sections 4.1.1 and 4.1.2 for the procedures on deriving these and other similar algorithms from relaxed stack algorithms.

Algorithm 7 Pseudocode for each process $p_i$ implementing an Out-of-Order $k$-Relaxed Priority Queue, where $k \geq n.$

1: HandleEvent INSERT(val)
2:  if localFastInsertsActive < $\tau_c - 1$ then
3:   send (insert$_f$, val, ⟨localTime, i⟩) to all
4:   setTimer(ε, ⟨insert$_f$, val, ⟨localTime, i⟩⟩, respond)
5:   localFastInsertsActive++
6:  else
7:   send (insert$_s$, val, ⟨localTime, i⟩) to all
8:   setTimer(ε, ⟨insert$_s$, null, ⟨localTime, i⟩⟩, respond)
9: HandleEvent EXTRACTMAX
10:  if $lPQ$.peekByLabel($p_i$, safeRegion()) $\neq \perp$ or $lPQ$.size() < $k - \tau n$ then
11:    let ret := $lPQ$.peekByLabel($p_i$, safeRegion())
12:    ret.extracted := true
13:    send (extractMax$_f$, ret, ⟨localTime, i⟩) to all
14:    setTimer(ε, ⟨extractMax$_f$, ret, null⟩, respond)
15:  else
16:    send (extractMax$_s$, null, ⟨localTime, i⟩) to all
17:    setTimer(ε, ⟨extractMax$_s$, 1, ⟨localTime, i⟩⟩, respond)
18: HandleEvent RECEIVE (op, val, ts) FROM $p_j$
19:    Pending.insert((op, val, ts))
20:    setTimer($d + u$, (op, val, ts), execute)
21:    if op == extractMax$_f$ or op == extractMax$_s$ then
22:      extractMaxesInPending++
23:    if op == extractMax$_f$ then
24:      resetVolatility(val)
25:    HandleEvent EXPIRETIMER((op, val, ⟨*, j⟩), respond)
26:    if op == extractMax$_f$ then return val
27:    else if op == extractMax$_s$ then
28:  flushReservedQueueFront(j)
29:  if reservedQueues[j].peek ≠ ⊥ then
30:    reservedQueues[j].peek().extracted := true
31:  return reservedQueues[j].peek()
32:  else if val · ε − [2d − (l − τ)ε] < ε then
33:    setTimer(min{ε, 2d − (l − τ)ε − val · ε}, (pop_s, val + 1, (*, j)), respond)
34:  else return ⊥
35:  else if op == insert_s and localFastInsertsActive == τc − 1 then
36:    setTimer(min{ε, 2d − (τc − 1)ε − val · ε}, (push_s, val + 1, (*, j)), respond)
37:  else return ACK
38:  HandleEvent EXPIRETIMER((op, val, ts), execute)
39:  while ts ≥ Pending.min() do
40:    ⟨op′, val′, ts′⟩ := Pending.extractMin()
41:    executeLocally(op′, val′, ts′)
42:    cancelTimer(⟨op′, val′, ts′⟩, execute)
43:  function EXECUTELOCALLY(op, val, (time, j))
44:    if op == insert_f or op == insert_s then
45:      lPQ.insert(val)
46:      if lPQ.size() ≤ k − τn then
47:        lPQ.label(argmin_pₙ | lPQ.setByLabel(pₙ) ∩ safeRegion(), val)
48:      else if val ∈ topSet(lPQ.labeledSize()) then
49:        if ∃elem ∈ safeRegion() s.t. P(elem) < P(val) then
50:          lPQ.label(lPQ.getFromTop(k − τn + 1).label, val)
51:          setVolatility(elem, k − τn + 1)
52:        else lPQ.label(dummy, val)
53:        lPQ.label(null, lPQ.getFromTop(k + min{|lowerRegion()|, popsInPending} + 1))
54:      if j == i and op == insert_f then localFastInsertsActive−−
55:    else
56:      if j ≠ i then flushReservedQueueFront(j)
57:      if op == extractMax_f then let toRemove := val
58:      else let toRemove := reservedQueues[j].dequeue()
59:      if toRemove.higher ≠ null then resetVolatility(toRemove.higher)
60:      extractMaxesInPending−−
61:      if toRemove ∈ safeRegion() then
62:        if volatileRegion() ∩ lPQ.topSet(k − τn) ≠ ∅ then
63:          let toRelabel := lPQ.latestBySet(volatileRegion())
64:          resetVolatility(toRelabel)
else let toRelabel := lPQ.getFromTop(k − τn + 1)

lPQ.label(j, toRelabel)

reservedQueues[j].enqueue(toRelabel)

lPQ.remove(toRemove)

if lPQ.labeledSize() < k then lPQ.label(dummy, lPQ.peekByLabel(null))

function SAFEREGION

return [topSet(k − τn) ∪ topSetBySet(extractMaxesInPending, lowerRegion())] \ latestSetBySet(extractMaxesInPending, volatileRegion())

function VOLATILEREGION

return {elem ∈ lPQ.topSet(k − τn) : elem.tvolatile > localTime − (2d + 2u + ϵ)}

function LOWERREGION

return {elem.lower : elem ∈ volatileRegion()}

function SETVOLATILITY(elem, other)

elem.tvolatile := localTime

elem.lower := other

elem.lower.higher := elem

function RESETVOLATILITY(elem)

elem.tvolatile := −∞

elem.lower.higher := null

elem.lower := null

function FLUSHRESERVEDQUEUEFRONT(j)

let front := reservedQueues[j].peek()

while ¬lPQ.contains(front) or front.extracted or front.label ≠ pj do

reservedQueues[j].dequeue()

let front := reservedQueues[j].peek()

Algorithm 8 Pseudocode for each process $p_i$ implementing an Out-of-Order $k$-Relaxed Queue, where $k \geq n$.

1: **HandleEvent** ENQUEUE(val)

2: send (enq, val, ⟨localTime, i⟩) to all

3: setTimer(ϵ, ⟨enq, null, ⟨localTime, i⟩⟩, respond)

4: **HandleEvent** DEQUEUE

5: if lQueue.peekByLabel($p_i$) ≠ ⊥ then

6: let ret := lQueue.peekByLabel($p_i$)

7: ret.dequeued := true

8: send (deq_f, null, ⟨localTime, i⟩) to all

9: setTimer(ϵ, ⟨deq_f, ret, null⟩, respond)
```java
10:     else
11:         send (deqₛ, null, ⟨localTime, i⟩) to all
12:         setTimer(ε, ⟨deqₛ, 1, ⟨localTime, i⟩⟩, respond)
13:     HandleEvent RECEIVE (op, val, ts) FROM pj
14:         Pending.insert((op, val, ts))
15:         setTimer(u + ε, ⟨op, val, ts⟩, execute)
16:     HandleEvent EXPIRE TIMER((op, val, ⟨∗, j⟩), respond)
17:         if op == deq_f then return val
18:         else if op == deqₛ then
19:             if lQueue.peekByLabel(pj) ̸= ⊥ then
20:                 let ret := lQueue.peekByLabel(pj)
21:                 ret.dequeued := true
22:                 return ret
23:             else if val · ε − [(d + ε) − lε] < ε then
24:                 setTimer(min{ε, (d + ε) − lε − val · ε}, ⟨deqₛ, val + 1, ⟨∗, j⟩⟩, respond)
25:             else
26:                 setTimer(d + ε − val · ε}, ⟨deqₛ, null, ⟨∗, j⟩⟩, respond)
27:         else if op == deqₛ2 then
28:             return lQueue.deqByLabel(pj)
29:         else return ACK
30:     HandleEvent EXPIRE TIMER((op, val, ts), execute)
31:         while ts ≥ Pending.min() do
32:             ⟨op′, val′, ts′⟩ := Pending.extractMin()
33:             executeLocally(op′, val′, ts′)
34:             cancelTimer((op′, val′, ts′), execute)
35:         function EXECUTELOCALLY(op, val, ⟨∗, j⟩)
36:             if op == enq then
37:                 lQueue.enq(val)
38:             if lQueue.size() ≤ k then
39:                 lQueue.label(argminₚₖ |lQueue.setByLabel(pₖ)|, val)
40:             else
41:                 lQueue.deqByLabel(pj)
42:             if lQueue.size() ≥ k then
43:                 lQueue.label(argminₚₖ |lQueue.setByLabel(pₖ)|, lQueue.peekByLabel(null))
```

**Algorithm 9** Pseudocode for each process $p_i$ implementing a Lateness $k$-Relaxed Queue, where $k ≥ n$. 

1: HandleEvent ENQUEUE(val)
2: send (enq, val, ⟨localTime, i⟩) to all
3: setTimer(ϵ, (enq, null, ⟨localTime, i⟩)), respond
4: HandleEvent DEQUEUE
5: if localDeqsInvoked < l_s then
6: localDeqsInvoked++
7: let ret := lQueue.peekByLabel(p_i)
8: ret.dequeued := true
9: send (deq_f, ret, ⟨localTime, i⟩) to all
10: setTimer(ϵ, (deq_f, ret, null), respond)
11: else send (deq_s, null, ⟨localTime, i⟩) to all
12: HandleEvent RECEIVE (op, val, ts) FROM p_j
13: Pending.insert((op, val, ts))
14: setTimer(2u, (op, val, ts), execute)
15: if op == deq_f then val.dequeued := true
16: HandleEvent EXPIRETIMER((op, val, ts), respond)
17: if op == deq_f then return val
18: else return ACK
19: HandleEvent EXPIRETIMER((op, val, ts), execute)
20: while ts ≥ Pending.min() do
21: ⟨op', val', ts'⟩ := Pending.extractMin()
22: executeLocally(op', val', ts')
23: cancelTimer(⟨op', val', ts'⟩, execute)
24: function EXECUTELOCALLY(op, val, ⟨∗, j⟩)
25: if op == enq then
26: lQueue.topAvailableByLabel(p_i).canDeqByLabel := true
27: lQueue.enq(val)
28: lQueue.label(argmin_p_h | lQueue.setByLabel(p_h)|, val)
29: lQueue.topAvailableByLabel(p_i).canDeqByLabel := false
30: else
31: if op == deq_f then
32: if j == i then localDeqsExecuted++
33: if val == lQueue.getFromTop(1) then
34: localDeqsInvoked := localDeqsExecuted
35: localDeqsExecuted := 0
36: lQueue.remove(val)
37: else
38: if localDeqsExecuted == l_s then
localDeqsInvoked − localDeqsExecuted

localDeqsExecuted := 0

let ret := lQueue.deq()

if ret.label == \textit{p}_i then
    lQueue.peekByLabel(\textit{p}_i).canDeqByLabel := false

else
    if \textit{j} == \textit{i} then
        localDeqsInvoked++
    else
        if \textit{j} == \textit{i} then return

Algorithm 10 Pseudocode for each process \textit{p}_i implementing a Restricted/Windowed \textit{k}-Relaxed Queue, where \(k \geq n\).

1: \texttt{HandleEvent ENQUEUE}(val)
2: send (\textit{enq}, val, (\textit{localTime}, \textit{i})) to all
3: \texttt{setTimer}(\epsilon, (\textit{enq}, \textit{null}, (\textit{localTime}, \textit{i})), \texttt{respond})

4: \texttt{HandleEvent DEQUEUE}
5: if \texttt{lQueue.peekByLabel}(\textit{p}_i) \neq \bot then
6: let ret := \texttt{lQueue.peekByLabel}(\textit{p}_i)
7: ret.dequeued := true
8: send (\textit{deq}_f, \textit{null}, (\textit{localTime}, \textit{i})) to all
9: \texttt{setTimer}(\epsilon, (\textit{deq}_f, ret, \textit{null}), \texttt{respond})
10: else send (\textit{deq}_s, \textit{null}, (\textit{localTime}, \textit{i})) to all
11: \texttt{HandleEvent RECEIVE} (\textit{op}, val, ts) \texttt{FROM} \textit{p}_j
12: \texttt{Pending.insert}((\textit{op}, val, ts))
13: \texttt{setTimer}(2\epsilon, (\textit{op}, val, ts), \texttt{execute})
14: if \textit{op} == \textit{deq}_f then \texttt{val.dequeued} := true
15: \texttt{HandleEvent EXPIRETIMER}((\textit{op}, val, (\textit{*, j})), \texttt{respond})
16: if \textit{op} == \textit{deq}_f then return \texttt{val}
17: else return \texttt{ACK}
18: \texttt{HandleEvent EXPIRETIMER}((\textit{op}, val, ts), \texttt{execute})
19: while ts ≥ \texttt{Pending.min()} do
20: (\textit{op}', val', ts') := \texttt{Pending.extractMin()}
21: \texttt{executeLocally}(\textit{op}', val', ts')
22: \texttt{cancelTimer}(\textit{op}', val', ts'), \texttt{execute})
23: \textbf{function} \texttt{EXECUTELOCALLY}(\textit{op}, val, (\textit{*, j}))
if \( op == \text{enq} \) then

\[
\text{lQueue.topAvailableByLabel}(p_i).\text{canDeqByLabel} := \text{true} \\
\text{lQueue.enq}(\text{val}) \\
\text{if } \text{lQueue.size()} \leq k \text{ then } \text{lQueue.label}(\arg\min_{p_h} |\text{lQueue.setByLabel}(p_h)|, \text{val})
\]

\[
\text{lQueue.topAvailableByLabel}(p_i).\text{canDeqByLabel} := \text{false}
\]

else

\[
\text{if } op == \text{deq}_f \text{ then} \\
\quad \text{if } j == i \text{ then } \text{localDeqsExecuted}++
\]

\[
\quad \text{if } \text{val} == \text{lQueue.getFromTop}(1) \text{ then} \\
\quad \quad \text{localDeqsInvoked} -= \text{localDeqsExecuted} \\
\quad \quad \text{localDeqsExecuted} := 0 \\
\quad \text{let } \text{ret} := \text{lQueue.peek()}
\]

else

\[
\quad \text{if } \text{localDeqsExecuted} == l_s \text{ then} \\
\quad \quad \text{localDeqsInvoked} -= \text{localDeqsExecuted} \\
\quad \quad \text{localDeqsExecuted} := 0 \\
\quad \text{let } \text{ret} := \text{lQueue.peekByLabel}(p_j)
\]

\[
\quad \text{if } j == i \text{ then return } \text{ret}
\]

\[
\text{ret.dequeued} := \text{true} \\
\text{ret.unresolved} := \text{true} \\
\text{if } \neg\text{ret.canDeqByLabel} \text{ and } \text{ret.label} == p_i \text{ then} \\
\quad \text{lQueue.peekByLabel}(p_i).\text{canDeqByLabel} := \text{false}
\]

\[
\text{if } \text{ret} == \text{lQueue.getFromTop}(1) \text{ then} \\
\quad \text{while } \text{lQueue has unresolved elements do} \\
\quad \quad \text{lQueue.remove} \text{ (the topmost unresolved element)} \\
\quad \quad \text{if } \text{lQueue.size()} \geq k \text{ then} \\
\quad \quad \quad \text{lQueue.label}(\arg\min_{p_h} |\text{lQueue.setByLabel}(p_h)|, \text{lQueue.peekByLabel}(\text{null}))
\]
REFERENCES


