TOWARDS UNDERSTANDING RELAXATIONS OF DISTRIBUTED DATA STRUCTURES

A Dissertation

by

EDWARD L. TALMAGE

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Chair of Committee,	Jennifer L. Welch
Committee Members,	Nancy Amato
	Andreas Klappenecker
	Alex Sprintson
Head of Department,	Dilma Da Silva

May 2018

Major Subject: Computer Science

ABSTRACT

Since computers are widespread and interconnected, the study of computing has expanded to encompass problems arising from concurrency and the need for coordinated effort between different computing entities. Essential among those problems is providing efficient means for multiple, distributed processes to operate on the same data, while guaranteeing that the results of their actions make sense and are useful. In this dissertation, we explore a method of improving the efficiency of operations on shared data.

The method we explore in this work is *relaxation* of a data type. Intuitively, relaxation consists of adding a limited amount of non-determinism to the specification of a data type. This allows multiple legal actions from a particular state. In some cases, we can associate these different actions with concurrent operations on shared data, allowing multiple processes to act without coordinating at that step. Since communication between different physical locations is relatively expensive reducing the need for coordinating communication can significantly increase the rate at which processes can execute operations on shared data.

After giving practical definitions of a few specific relaxations from the literature, we proceed in three steps. First, we provide implementations and analysis of algorithms for FIFO queues under several of these relaxations, showing the performance benefits relaxation provides. We then analyze the computational power of relaxed queues, to understand what we have given up to achieve improved performance. Finally, we compare the relaxation model to the study of weakened consistency conditions, a common approach in the literature. We show a partial correspondence between the models, allowing us to use existing tools to analyze relaxations.

To conclude, we step away from relaxations and provide a set of heuristics for determining the consensus number of a data type, which is a measure of its computational strength. While it is an undecidable problem in general, in specific cases our tools may allow system designers to recognize whether or not a specific type provides the strength they need for their particular distributed application.

DEDICATION

To Sara

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professors Jennifer Welch [advisor], Nancy Amato, and Andreas Klappencker of the Department of Computer Science and Engineering and Professor Alex Sprintson of the Department of Electrical and Computer Engineering.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a Graduate Research Assistantship under Dr. Jennifer Welch funded by NSF grants 1526725 and 0964696, as well as a Graduate Teaching Fellowship from the Texas A&M University College of Engineering and a Graduate Teaching Assistantship from the Texas A&M University Department of Computer Science and Engineering.

TABLE OF CONTENTS

Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION

With the increase in data collection and storage ability of the last decade, computational problems are growing extremely quickly. To enable the kind of bigger computing required for these problems, we distribute tasks to large sets of computers, which are often geographically-dispersed. This allows us to apply large amounts of computational power to a single problem, while balancing the load of maintenance and upkeep of the computers. However, distributed computation is in many ways more complex than local and has its own set of unique challenges and foibles.

In this dissertation, we work towards providing tools that make distributed computing practical and easy to use. Particularly, we explore the properties of distributed data structures, which are fundamental tools in distributed programming. By abstracting basic tasks of storing, sharing, and interacting with data needed by multiple processes, we can remove the complications of coordinating concurrent operations from a programmer's burden. We explore some possibilities of distributed data structures, from implementation to specification, speed and computational power.

1.1 Relaxed Data Types

Most of this work focuses on relaxed data types. In essence, relaxing a data type adds a precise, limited amount of non-determinism, which allows different processes to perform some concurrent operations with less synchronization than is required for fully deterministic data types. Since communication delays are often vastly larger than those of local computation, this translates to significant performance improvements. For example, one type of relaxed FIFO queue allows each $Dequeue$ to return one of the k oldest elements currently in the queue, instead of the exact oldest. This can allow up to k $Dequeue$s to be performed without any synchronization between processes.

Chapter 3 explores implementations of relaxations of FIFO queues. We present distributed algorithms to implement two versions of a relaxed queue. We then give lower bounds on the amortized time of any implementation of an unrelaxed FIFO queue which show our algorithms have better performance, in an amortized sense, than any possible algorithm for an unrelaxed

queue. Following those, we prove lower bounds on algorithms implementing these relaxed queues. These show, first, that our algorithms were asymptotically optimal and, second, that increasing the relaxation parameter gives continual improvements to performance. This confirms that we should be looking for applications which can use relaxed data types, since they can achieve better performance than the corresponding unrelaxed types.

Intuitively, relaxation gives up some strength to allow greater efficiency. In Chapter 4, we use the notion of consensus numbers to characterize the amount of computational strength an augmented queue loses in relaxation. The consensus number of a data type, which is defined as the largest number of processes which can use objects of that type to solve the consensus agreement problem (see Section 2.4), is a classic measure of the computational strength of data types in asynchronous, failure-prone systems. We consider three relaxed versions of augmented queues, which have a $Peek$ operation as well as $Enqueue$ and $Dequeue$. Each operation can be relaxed by an integer parameter, so we have an infinite 3-dimensional space of relaxed queues for each relaxation. By directly proving the consensus numbers of a few specific relaxations and some lemmas relating the consensus numbers of different choices of relaxation parameters, we find exact consensus numbers for every point in these relaxation spaces. Our results show that even a slight change in relaxation can significantly reduce the computational power of the type. This shows that it behooves a developer using a relaxed data type to choose a relaxation type and parameters very carefully to ensure the data type maintains the required computational strength.

Another historically common approach to improving the performance of distributed data structures is to weaken the required consistency condition. Consistency conditions specify how concurrent behavior relates to sequential data type specifications. By weakening the constraints on concurrent behavior, more distributed executions are considered acceptable, and more efficient algorithms become possible. We show that data type relaxations are a subset of consistency conditions. That is, relaxations and some weakened consistency conditions are different ways to express the same set of allowable behaviors. In Chapter 5, we show correspondences between specific conditions and use existing tools from the literature to compare the strength of some relaxations

by comparing their corresponding consistency conditions. This ability to use whichever model is more convenient is one direct, practical benefit of the correspondence.

1.2 Sensitivity

One of the primary concerns when working with distributed data types, relaxed or unrelaxed, is their computational strength, most often represented by the type's consensus number. In Chapter 6, we define the notion of *sensitivity*, which captures which part or parts of the prior history of operations on the data structure a process can learn about in a single operation. We then prove consensus numbers for data types with operations in one of several classes defined by sensitivity. This allows a user to find a data type's consensus number simply by determining its sensitivity class. This can be much easier than directly finding a consensus number by giving an algorithm to solve consensus among a certain number of processes and an impossibility proof showing that no such algorithm exists for any larger number of processes. Our method may thus help a developer who needs to quickly determine the computational power of a given data type.

2. DEFINITIONS AND MODEL *

In this chapter, we formally define the concepts of data types and relaxations as we will use them throughout the dissertation. We also discuss the models of computation in which we will work.

2.1 Specifying Data Types

An *Abstract Data Type* specifies an interface for interacting with data, and defines how the data object will behave. Data type specifications consist of the possible operations which a process may invoke and a set of sequences of operation instances which specifies all possible return values an operation response may have, given an invocation and a sequence of past operations. We here consider only objects which have sequential specifications, as relaxation of tasks without sequential specifications (see, e.g., [1, 2]) has not yet been defined. We follow the definitions in [3] but modified to encompass nondeterminism (and thus relaxation).

Definition 1. *An* Abstract Data Type *consists of*

1. *A set OPS of operations and the sets $args(OP)$ of valid arguments and $rets(OP)$ of valid return values for each $OP \in OPS$. An* instance *of an operation OP, denoted $OP(arg, ret)$, contains the argument(s) arg and the value(s) returned, ret. In a sequential environment,*

*Parts of the material in this chapter are reprinted from the following papers:

E. Talmage and J.L. Welch, "Improving average performance by relaxing distributed data types," in *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings* (F. Kuhn, ed.), vol 8784 of *Lecture Notes in Computer Science*, pp. 421-438, Copyright 2014 by Springer.

E. Talmage and J.L. Welch, "Generic proofs of consensus numbers for abstract data types," in *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France* (E. Anceaume, C. Cachin, and M. G. Potop-Butucaru, eds.), vol. 46 of *LIPIcs*, pp. 32:1-32:16, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

E. Talmage and J.L. Welch, "Anomalies and Similarities among consensus numbers of variously-relaxed queues," in *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings* (A.E. Abbadi and B. Garbinato, eds.), vol. 10299 of *Lecture Notes in Computer Science*, pp. 191-205, Copyright 2017 by Springer.

E. Talmage and J.L. Welch, "Relaxed data types as consistency conditions," in *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings* (P.G. Spirakis and P. Tsigas, eds.), vol. 10616 of *Lecture Notes in Computer Science*, pp. 142-156, Copyright 2017 by Springer.

instances are indivisible, but we will consider them as a distinct invocation and matching response in the distributed setting.

When either $args(OP)$ or $rets(OP)$ contains no values, we write an instance formally as $OP(arg, -)$ or $OP(-, ret)$ and for convenience condense the notation to $OP(ret)$ or $OP(arg)$, as appropriate.

2. *A set $\mathcal{L}$ of* legal *sequences of operation instances which satisfies two properties:*

   - Prefix Closure: *If a sequence ρ is in $\mathcal{L}$, then every prefix of ρ is also in $\mathcal{L}$.*
   - Completeness: *If a sequence ρ is in $\mathcal{L}$, then for every operation $OP \in OPS$ and every argument arg in $args(OP)$, there is a response $ret \in rets(OP)$ such that $\rho \cdot OP(arg, ret)$ is in $\mathcal{L}$.*

We use *state* of an object to refer to the equivalence class of operation sequences which allow the same set of extending sequences as the sequence of instances which have been executed on that object. That is, we say that two such sequences π and ρ are *equivalent*, denoted $\pi \equiv \rho$, if for any sequence σ where either $\pi \cdot \sigma$ or $\rho \cdot \sigma$ is legal, then $\rho \cdot \sigma$ or $\pi \cdot \sigma$ is also legal, respectively, and a state is an equivalence class of such sequences. We classify operations by whether they change a shared object's state, return information about it, or both. Every operation on a shared object must be either an *accessor*, which returns some information dependent on the state of the object, a *mutator*, which changes the state of the object, or both, which we call a *mixed* operation. Operations which are neither accessors nor mutators would be constants or no-ops, and are thus not useful operations on a shared object.

Definition 2. *An operation OP of an abstract data type T is a* mutator *if there exists a legal sequence ρ of instances of operations of T and an instance op of OP such that $\rho \not\equiv \rho \cdot op$. An operation OP is an* accessor *if there exist legal sequences ρ, ρ' of instances of operations of T and an instance aop of OP such that $\rho \cdot aop$ is legal, but $\rho' \cdot aop$ is not legal.*

An operation which is both an accessor and a mutator is a mixed *operation. An operation*

which is a mutator but not an accessor, or an accessor but not a mutator, is a pure *mutator or accessor, respectively.*

For example, in a ***Read-Modify-Write (RMW)*** register, $Read$ is a pure accessor, $Write$ is a pure mutator, and $Read\text{-}Modify\text{-}Write$ is a mixed operation. In a FIFO queue augmented with $Peek$, $Enqueue$ is a pure mutator, $Dequeue$ is a mixed operation, and $Peek$ is a pure accessor. A data type does not need to have all three kinds of operations, as seen in a $Read/Write$ register or FIFO queue without $Peek$.

Note that removing all instances of pure accessors from a sequence of operation instances π does not change the state represented, so we denote this equivalent sequence containing only mutator instances as $\pi|_m$. We similarly use $\pi|_{args}$ to denote the sequence of arguments to the operation intances in π.

2.1.1 Relaxed Queue Specifications

We will consider four different types of relaxation introduced in [4] and re-formulated for relaxing queues in [5, 6]. Each relaxed operation has a parameter specifying the maximum amount of relaxation allowed, either for each instance of the operation or bounding the number of consecutive operation instances which can behave differently than the unrelaxed operation. The Out-of-Order k-relaxation allows each operation instance to take effect up to k places out of order. For example, a $Dequeue$ can return any of the first k elements at the head of a queue, instead of only the first. The Lateness k-relaxation merely requires that at least one in every k instances must behave as the unrelaxed version, while the other instances may disregard ordering. The Restricted Out-of-Order k-relaxation is the intersection of the previous two relaxations, requiring that consecutive instances which behave in an out-of-order fashion are increasingly near to the correct order. The Stuttering k-relaxation allows some mutator instances to fail to change the state, requiring that at least one in every k instances must behave as the unrelaxed version.

We will next formally define these relaxations on FIFO queues augmented with a $Peek$ operation, but first we must note that each type will have three different relaxation parameters, one for each operation. Each of these parameters may be in the set $\mathbb{Z}^+ \cup \{*, \emptyset\}$, which we will denote as

$\mathbb{Z}^*$. A $\emptyset$ parameter, equivalent to a 0 in [7], means that the operation is not supported, while we consider that $* > x, \forall x \in \mathbb{Z}^+$. That is, $*$-relaxed is infinitely relaxed, and such operations have no ordering constraints. For technical reasons, we will define $\emptyset > * > x, \forall x \in \mathbb{Z}^+$.

We assume that all arguments to queue operations are unique (accomplishable by timestamps). We can represent the state of a relaxed queue by the sequence of elements which have been inserted (by $Enqueue$) but not yet removed (by $Dequeue$), denoting one end as the *head* and the other as the *tail*. In an unrelaxed queue, $Enqueue(val)$ appends val to the tail of the queue, while $Dequeue$ and $Peek$ return the value at the head of the queue, with $Dequeue$ also removing that element from the queue. $Peek$ and $Dequeue$ may return a special symbol $\perp$ if the queue appears to contain no elements (relaxation may allow the queue to appear empty even when it is not). We assume that $\perp$ is not a possible input to any operation. Formally:

Definition 3. *A* queue *over a set of values V is a data type with two operations:*

- $Enqueue(val, -), val \in V$*; intuitively, adds element val and has no return value*
- $Dequeue(-, val), val \in V \cup \{\perp\}$*; intuitively, removes an element and returns it, and has no argument*

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

(C1) Every argument to an instance of $Enqueue$ is unique[1].

(C2) Every return value of a $Dequeue$ instance is unique.

(C3) Every non-$\perp$ value which an instance of $Dequeue$ returns is the argument to a previous instance of $Enqueue$.

(C4) If ρ is a legal sequence of operation instances, then $\rho \cdot Dequeue(-, val), val \neq \perp$, is legal iff $Enqueue(val, -)$ is the first $Enqueue$ in ρ which does not have a matching $Dequeue(-, v)$ in ρ. Furthermore, $\rho \cdot Dequeue(-, \perp)$ is legal iff every $Enqueue(val, -)$ in ρ has a matching $Dequeue(-, val)$ in ρ.

[1] As mentioned previously, this can easily be achieved by timestamping arguments.

For the Out-of-Order, Lateness, and Restricted Out-of-Order relaxations, we first give a formal definition with only $Dequeue$ relaxed, then a less formal definition which relaxes all three operations. The first reason for this is ease of understanding, to minimize potentially confusing formality. The second reason is that when we give implementations of some of these relaxed queues in Chapter 3, we only need to relax $Dequeue$, becuase results in [3] show that $Enqueue$ can always be fast.

To introduce the notation we need, consider first an unrelaxed FIFO queue. The queues with relaxed $Dequeue$s use some of conditions *(C1)-(C3)*, but each modifies condition *(C4)* to give a different set of legal return values for $Dequeue$, and one does not use *(C2)*. The altered versions of *(C4)* allow a larger set of possible return values.

The Out-of-Order relaxation, instead of requiring a $Dequeue$ to return the oldest element, allows any of the k oldest elements as a return value for a $Dequeue$.

Definition 4. *An* Out-of-Order k-relaxed queue *satisfies (C1)-(C3) from Definition 3, and the following condition:*

(C4) If ρ is a legal sequence of operation instances, then $\rho \cdot Dequeue(-, val)$, is legal iff there are fewer than k distinct val's such that $Enqueue(val', -)$ precedes $Enqueue(val, -)$ in ρ and there is not a matching $Dequeue(-, val')$ in ρ. Furthermore, $\rho \cdot Dequeue(-, \perp)$ is legal iff there are fewer than k $Enqueue(val', -)$'s in ρ without matching $Dequeue(-, val')$'s in ρ.

Relaxing $Peek$ is exactly the same, with the constraing that $Peek$ never removes an element. Relaxing $Enqueue$ just considers in what position the operation places its argument, which may or may not be the tail. We then have the general Out-of-Order relaxation of a queue:

Definition 5. *An Out-of-Order relaxed queue with parameters $a, b, c \in \mathbb{Z}^*$, which we denote as $OQueue[a, b, c]$, provides three operations, as follows:*

- $Enqueue[a](val)$ *adds val to the OQueue such that at most $a - 1$ elements already in the OQueue are nearer the tail than val*

- $Dequeue[b]()$ *removes and returns one of the first* b *elements at the head end of the OQueue; if there are fewer than* b *elements in the OQueue,* $Dequeue[b]()$ *may return* $\perp$
- $Peek[c]()$ *returns, without removing, one of the first* c *elements at the head end of the OQueue; if there are fewer than* c *elements in the OQueue,* $Peek[c]()$ *may return* $\perp$

$Enqueue[\emptyset]$, $Dequeue[\emptyset]$ *and* $Peek[\emptyset]$ *are no-ops.*

For the next two relaxations, we need the concept of *lateness*, which is a measure of how many consecutive operations of a specific type have been out of order. Define $lateness(OP[k])$ for a finite sequence ρ of operations instances on a relaxed queue as the number of instances of $OP[k]$ appearing in ρ after the latest instance of $OP[k]$ that behaved as the unrelaxed version would. That is, the number of $Enqueue[a]$ instances since the last one which put an element at the tail, the number of $Dequeue[b]$ instances since the last which removed the head, or the number of $Peek[c]$ instances since the last which returned the head.

The next relaxation, Lateness, does not impose any restriction on how close to the top an element which a $Dequeue$ returns must be, but instead simply enforces that each operation's lateness never exceeds that operation's relaxation parameter. For $Dequeue$, this means the head is returned by at most the kth $Dequeue$ after it became head.

Definition 6. *A* Lateness k-relaxed queue *satisfies* (C1)-(C3) *from Definition 3, and the following condition:*

(C4) If ρ *is a legal sequence of operation instances, then* $\rho \cdot Dequeue(-, val)$ *is legal iff every* $Enqueue(val', -)$ *preceding* $Enqueue(val, -)$ *has a matching* $Dequeue(-, val')$ *in* ρ *or there are fewer than* $k-1$ *instances* $Dequeue(-, val')$ *that follow the first* $Enqueue(val'', -)$ *which does not have a matching* $Dequeue(-, val'')$ *in* ρ*.*

Further, $\rho \cdot Dequeue(-, \perp)$ *is legal iff there are fewer than* $k-1$ *instances* $Dequeue(-, val')$ *that follow the first* $Enqueue(val'', -)$ *without a matching* $Dequeue(-, val'')$ *in* ρ *or every* val' *such that* $Enqueue(val', -)$ *is in* ρ *has a matching* $Dequeue(-, val')$ *in* ρ*.*

Again, the definition of a relaxed $Peek$ is practically identical to that of a relaxed $Dequeue$ and a relaxed $Enqueue$ is symmetric.

Definition 7. *A Lateness relaxed queue with parameters $a, b, c \in \mathbb{Z}^*$, denoted $LQueue[a, b, c]$, provides three operations, as follows:*

- $Enqueue[a](val)$ *inserts* val *at an arbitrary location in the LQueue, while maintaining* $lateness(Enqueue[a]) < a$
- $Dequeue[b]()$ *removes and returns any element in the LQueue, or* $\perp$*, while maintaining* $lateness(Dequeue[b]) < b$
- $Peek[c]()$ *returns, without removing, any element in the LQueue or* $\perp$*, while maintaining* $lateness(Peek[c]) < c$

$Enqueue[\emptyset]$, $Dequeue[\emptyset]$ *and* $Peek[\emptyset]$ *are no-ops.*

A Restricted Out-of-Order relaxed queue keeps the requirement of an LQueue that at least one in every k consecutive instances of a given operation must behave as if unrelaxed, but also requires every operation instance to approximately respect the ordering of an unrelaxed queue. Thus, it can be seen as the intersection of the last two definitions. It allows an instance of $Dequeue$ to return any of the first k elements at the head of the queue, as fixed in time when last the single oldest element was returned. Thus, at least once every k instances of $Dequeue$, the true top element must be returned.

Definition 8. *A* Restricted Out-of-Order k-relaxed queue *satisfies (C1)-(C3) from Definition 3, and the following condition:*

(C4) If ρ is a legal sequence of operation instances, $\rho \cdot Dequeue(-, val), val \neq \perp$, is legal iff, in the suffix ρ' of ρ which starts at the first $Enqueue(val', -)$ which does not have a matching $Dequeue(-, val')$ in ρ, $Enqueue(val, -)$ is among the first k instances of $Enqueue$.

$\rho \cdot Dequeue(-, \perp)$ is legal iff there are fewer than k instances $Enqueue(val', -)$ in ρ'.

With all three operations relaxed:

Definition 9. *A Restricted Out-of-Order relaxed queue with parameters $a, b, c \in \mathbb{Z}^*$, which we denote as $RQueue[a, b, c]$, provides three operations, as follows:*

- $Enqueue[a](val)$ *adds val to the queue such that at most $(a-1) - lateness(Enqueue[a])$ elements already in the RQueue are nearer the tail than val*
- $Dequeue[b]()$ *removes and returns one of the first $b - lateness(Dequeue[b])$ elements at the head end of the RQueue; if there are fewer than $b - lateness(Dequeue[b])$ elements in the RQueue, $Dequeue[b]()$ may return $\perp$*
- $Peek[c]()$ *returns, without removing, one of the first $c - lateness(Peek[c])$ elements at the head end of the RQueue; if there are fewer than $c - lateness(Peek[c])$ elements in the RQueue, $Peek[c]()$ may return $\perp$*

$Enqueue[\emptyset]$, $Dequeue[\emptyset]$ and $Peek[\emptyset]$ are no-ops.

Note that if an operation's relaxation parameter is $*$ then, in each type of relaxation, all ordering constraints are gone, since at any particular point in time, there will be a finite number of elements in the queue, and lateness will be finite. Thus, $Enqueue[*]$ can put its argument in any location in the queue and $Dequeue[*]$ and $Peek[*]$ may return any element of the queue, regardless of the type of relaxation.

The stuttering relaxation, the last we will define here, has a very different flavor than the other relaxations. Instead of constraining ordering properties of an operation, it allows mutators to execute without actually changing the simulated state of the shared queue. Instead, up to k times, the operation may return as if it completed, in the case of a mixed operation like $Dequeue$ returning a value, but not changing the state.

Definition 10. *A* stuttering k-relaxed queue *satisfies (C1) and (C3) from Definition 3, and the following condition:*

(C4) If ρ is a legal sequence of operation instances, then $\rho \cdot Dequeue(-, val), val \neq \perp$ is legal iff there is no $Dequeue(-, val')$ with $val' \neq val$ such that either $Enqueue(val', -)$ is in ρ after $Enqueue(val, -)$ or $val' = \perp$, and there are fewer than k copies of $Dequeue(-, val)$ in ρ.

$\rho \cdot Dequeue(-, \perp)$ is legal iff every $Enqueue(val', -)$ in ρ has at least one corresponding $Dequeue(-, val')$.

Again, with all three operations relaxed. Note that pure accessors, like $Peek$, cannot be stuttering-relaxed, since they do not affect the object's state to start with.

Definition 11. *A Stuttering relaxed queue with parameters $a, b, c \in \mathbb{Z}^*$, denoted $SQueue[a, b, c]$, provides three operations, as follows:*

- *$Enqueue[a](val)$ attempts to add val to the tail of the queue such that at most $a - 1$ consecutive $Enqueue[a]$ instances do not change the SQueue's state*
- *$Dequeue[b]()$ returns the first element at the head end of the SQueue and at least one in every b consecutive $Dequeue[b]$ instances removes the element it returns; if there are no elements in the queue, $Dequeue[b]()$ returns $\perp$*
- *$Peek[c]()$ returns, without removing, the element at the head end of the SQueue, $\perp$ if there is none.*

$Enqueue[\emptyset]$, $Dequeue[\emptyset]$ and $Peek[\emptyset]$ are no-ops.

We will use natural reductions of notation to increase readability, such as denoting $Enqueue[1]$ as $Enqueue$, etc., since this is an unrelaxed operation. To specify the actual behavior of a particular $Enqueue[a]$ instance, we will also use the notation $Enqueue_i^t(x)$ to denote an $Enqueue$ instance executed by process p_i which places x immediately head-ward of the tail-most t elements.

2.1.2 General Data Type Relaxations

We here present definitions of the above relaxations for arbitrary data types, not just for queues. We restate these definitions purely in terms of legal sequences of operation instances, where [4]

combined equivalence classes of such sequences to develop a state machine notation. We choose to focus on these relaxations, comparing them to consistency conditions in Chapter 5, since a number of authors [5, 8, 7, 9, 10, 11] have considered implementations and analyses of these and similar relaxations.

First, we consider the Out-of-Order relaxation. The definition of this relaxation does not immediately appear to have anything to do with ordering, but when instantiated on operations in ordered data structures such as $Dequeue$ in queues and Pop in stacks, it causes those operations to return an element up to k places out of order. One way to think about this is to imagine that by deleting operation instances in the past, we are making the current instance act as if it is in a different place in the permutation of all instances.

Note that [4] defines the k-Out-of-Order relaxation to allow either deleting or inserting up to k operation instances. Some operations, though, could have arbitrary behavior if arbitrary operation instances may be added to the history. For example, $Dequeue$ and Pop, if $Enqueue(x)$ or $Push(x)$, for arbitrary x, is added such that $Dequeue$ or Pop returns x. To avoid this, we restrict our attention to out-of-order with respect to deleting past instances.

Definition 12 (k-Out-of-Order Relaxed ADT)**.** *Given any ADT T and an integer $k \geq 0$, a k-Out-of-Order relaxation of T, called T', is defined as follows:*

1. $OPS(T') = OPS(T)$
2. *A sequence Π is legal if for every instance op where $\Pi = \pi \cdot op \cdot \rho$, there is some sequence $u \cdot v \cdot w, |v| \leq k$, which is a minimum-length sequence equivalent in T to π, and there exists a sequence x, where*

   a. *$u \cdot w$ is legal in T and minimum-length among the set of sequences equivalent to it in T,*

   b. *$u \cdot w \cdot op$ is legal in T, and*

   c. - *$u \cdot w \cdot op \equiv x \cdot w$ and $\pi \cdot op \equiv x \cdot v \cdot w$ or*
      - *$u \cdot w \cdot op \equiv u \cdot x$ and $\pi \cdot op \equiv u \cdot v \cdot x$.*

Intuitively, an instance op is allowed after some prefix π if some contiguous portion of the prefix can be ignored. The relaxation does not want to consider past actions which have since been undone, such as an overwritten write or removed element, so we replace π with a minimum-length sequence equivalent to it ($u \cdot v \cdot w$). We then delete up to k consecutive mutator instances (v), making $u \cdot w \cdot op$ legal in the base type. Now, $u \cdot w \cdot op$ being legal in T means that $\pi \cdot op$ is legal in a k-Out-of-Order relaxation T' of an ADT T, but we need to specify what effect op had. We do this by saying that the set of sequences legal in T' after $\pi \cdot op$ is the same set as those legal after reinserting the deleted sequence of instances ($x \cdot v \cdot w$ or $u \cdot v \cdot x$, as appropriate).

In this and other relaxations, we refer to T, the type from which the relaxation is defined, as the *base type*.

The next relaxation we consider is *Lateness*. This name comes because one way to view the relaxed data type is that operations may act as Out-of-Order, each for any finite relaxation parameter, except that each time an instance does not satisfy the specification of the base type, we increase a lateness counter. That counter can never exceed k, and resets when an instance acts by the specification of the base type. Thus, we can have instances arbitrarily far from the base type's behavior, but are guaranteed that at least one in every k consecutive instances behaves normally. For example, a relaxed $Dequeue$ may return and remove any element in the queue, as long as one in every k $Dequeue$s returns the head.

Definition 13 (k-Lateness Relaxed ADT)**.** *Given any ADT T and an integer $k \geq 1$, a k-Lateness relaxation of T, T', is defined as follows:*

1. $OPS(T') = OPS(T)$
2. *A sequence Π of operation instances is legal in T' if for every instance op such that $\Pi = \pi \cdot op \cdot \rho$, there exists $l \geq 0$ such that $\pi \cdot op$ is legal by the semantics of an l-Out-of-Order relaxed T, and at least one in every k consecutive mutator instances in $\Pi|_m$ must have $l = 0$.*

Finally, we consider a relaxation with a different flavor. Instead of allowing operations to act slightly incorrectly, this relaxation allows some mutator instances to have no effect on the state

of the shared object. That is, some mutators may "stutter" on the current object state, failing to change it. Here, we only require that some fraction of mutator instances successfully change the object, while others may fail to take effect. All instances must still return a value that is legal based on the current state of the object. To do this, we track the subsequence of mutator instances in the schedule that do not stutter. This subsequence, represented by the π'_is, is the history that determines the next operation instance's behavior. For example, a stuttering counter may hold the same value after up to k consecutive $increment()$ instances before increasing. The π'_i consists only of those $increment$ instances which actually increased the counter's value.

Definition 14 (k-Stuttering Relaxed ADT)**.** *Given any ADT T and an integer $k \geq 1$, a k-Stuttering relaxation of T, T' is defined as follows:*

1. $OPS(T') = OPS(T)$
2. *A sequence $\Pi = op_1 \cdot op_2 \cdots$ is legal if every op_i, with $\Pi = \pi_i \cdot op_i \cdot \rho_i$, op_i returns a value such that $\pi'_i \cdot op_i$ is legal in T, where π'_i is a sequence of mutator instances such that*
   (a) *$\pi'_1 = \varepsilon$, the empty sequence,*
   (b) *$\pi'_i \in \{\pi'_{i-1}, \pi'_{i-1} \cdot op_{i-1}\}$ for $i > 1$, and*
   (c) *π'_i includes at least one of every k consecutive mutators in π_i.*

2.2 System Models

In this dissertation, we use two different models of computation to investigate different questions. These models focus attention on the details relevant to each particular problem. Recall that one of our goals is to hide real-world features of distributed systems, such as messages, from the end user, so the message-passing and shared-memory models can be seen as the environments below and above, respectively, an implementation of a shared data type. In both models, we assume a set $\Pi = \{p_0, \ldots, p_{n-1}\}$ of processes, which are arbitrary computing elements.

2.2.1 Partially Synchronous, Message Passing

We model each process in Π as a state machine. There are three kinds of events that can trigger a transition of the state machine for a process: the receipt of a message, a local timer going off, or the invocation of an operation instance. A *step* of a process is a 6-tuple (s, T, C, M, R, s'), where s is a state of the process (the old state), T is a trigger event, C is the local clock value (a real number), M is a set of messages (to be sent), R is either $\emptyset$ or an operation instance response, and s' is a state of the process (the new state), such that M, R, and s' are the result of the transition function operating 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 of the state machine;
- the old state of each step after the first one equals the new state of the previous step;
- each timer in the old state of each step has a value that does not exceed the clock time of the step;
- if the trigger of a step is xoa timer going off, then the old state of the step has a timer whose value is equal to the clock time for the step
- clock times of steps are increasing, and if the sequence is infinite then they increase without bound;
- at most one operation instance is pending at a time

A *timed view* is a view with a real number, called "real time", associated with each step. There must exist a real number c such that, for each step, the difference between the clock time and the real time is exactly c (the "offset" of the process' local clock from real time).

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 matching 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 d, u, and ϵ, if

- every received message has delay in the range $[d-u, d]$ and if a message is sent but not received, then the recipient's last step is at real time less than $t+d$, where t is the real time when the message is sent;
- for all processes p_i and p_j, $|c_i - c_j| \leq \epsilon$, where c_i is the clock offset of p_i and c_j is the clock offset of p_j.

We assume that any message from a process to itself is simulated as taking the minimum message delay $d-u$.

We consider only algorithms which are *Eventually Quiescent*: Every complete admissible run with a finite number of operations is finite (i.e., every view is finite).

2.2.2 Asynchronous, Shared Memory

For our second model, consider an asynchronous, shared-memory model of computation on n processes. We split operation instances into separate invocations and responses. Processes interact by invoking operations, with arguments, on shared objects. Some time after an invocation, the object responds, giving the process a return value. Computation takes the form of *schedules*. A schedule of a data type T is a collection of sequences, one per process, of alternating invocations and responses of operations of T, each occurring at some real time and with each response of the same operation as the previous invocation. Each process' sequence is either infinite or ends in an operation response. In a schedule, we call two operation instances at different processes *overlapping* if the real time of one instance's invocation is between the real times of the invocation and response of the other instance. A schedule implies a partial order, called the *schedule order*, on non-overlapping instances, where an instance that returns before a second is invoked, in real time, precedes it, while overlapping instances are not ordered with respect to each other.

2.3 Consistency Conditions

Since data type specifications are inherently sequential, we need some way to relate a schedule of a distributed system, which is inherently concurrent, to those specifications. A consistency condition specifies what concurrent schedules are *legal* on a given data type. Formally, a consistency condition C is the union, over all data types T, of the sets of schedules legal on T under C. When discussing a consistency condition in conjunction with a particular data type, we will implicitly consider only the subset of schedules for that type. This definition overloads the term "legal" to refer to schedules which correspond, by the consistency condition, to legal sequences on the given data type. Equality of consistency conditions is set equality between sets of legal schedules [12].

As an example, we define *Linearizability* [13], which is used throughout the literature in combination with relaxed data types, as it is the most intuitive.

Definition 15 (Linearizability)**.** *A schedule E on a data type T is legal under* linearizability *if there exists a permutation Π of all operation instances in E such that (1) If an instance op precedes another instance op' in the schedule order, then op precedes op' in Π, and (2) Π is legal, according to the sequential specification of T.*

Weaker consistency conditions may allow some reordering with respect to the schedule order. For example, k-Atomicity for $Read/Write$ registers, introduced in [14], allows $Read$ operations to get a "stale" value, possibly missing some updates which overlap or even immediately precede the $Read$ instance in the schedule order. This staleness is bounded by the constant k, ensuring that the behavior is not arbitrary. In practice, the values "missed" can reflect $Write$ instances which the process invoking the $Read$ has not yet heard about. [14] gives probabilistic results showing that only requiring k-Atomicity can lead to implementations with higher proportions of operations which succeed, meaning that processes do not need to retry as often, improving performance.

All the consistency conditions we consider require *liveness*: In every complete, admissible run of an implementation of an abstract data type, every operation invocation has a matching response and every response has a matching invocation.

2.4 Consensus Numbers

To classify the computational power of shared data types, we use the *consensus* problem [15]. The consensus problem is for each of n processes to either crash or in a finite amount of time, starting with an input value in $\{0, 1\}$, agree on (or *decide*) and return the same output value as all other deciding processes, such that the decided value was some process' input.

Formally, the consensus problem is defined as follows: Every process has an initial *input* value $v \in \{0, 1\}$. After that, if it is correct, it will *decide* a value $d \in \{0, 1\}$. Once a process decides a value, it cannot change that decision. Further, all correct processes must satisfy three conditions:

- Termination: All correct processes eventually decide some value
- Agreement: All correct processes decide the same value d
- Validity: All correct processes decide a value which was some process' input

An abstract data type T can *implement* consensus if there is an algorithm in the given model which uses objects of T (plus registers) to solve consensus. The *consensus number*, introduced in [15], of an abstract data type is the largest number of processes n for which there exists an algorithm to implement consensus among n processes using objects of that data type. If there is no such largest number, we say the data type has consensus number ∞.

To prove a lower bound on a type's consensus number, we merely exhibit an algorithm which uses objects of that type to solve consensus among some number of processes. For an upper bound, we use the technique of *valency*, as in [15], and its extensions to non-deterministic types from [16]. We here re-state several concepts and lemmas from these papers, as well as [17], which allow us to streamline our proofs.

A *configuration* of an algorithm consists of the local states of all processes and the states of all shared objects. An *initial configuration* is one where every process is in an initial local state and every shared object has an initial state, as specified by the algorithm. We say that two configurations C and D are *indistinguishable* to a process p_i if p_i has the same local state and all shared objects have the same state in C and D.

Process p_i takes *step* (C, op_i, C'), where C and C' are configurations we call the old and new configurations of the step, if it executes an atomic operation instance op_i on a shared variable V. V and op_i's operation and argument are specified by p_i's state in C. This is said to be an *enabled* step. The resulting configuration C' differs from C only in the local state of p_i, according to the algorithm, and V's state, according to its type. We call C' a *child configuration* of C and use the notation $C \cdot op_i$ to denote the child configuration C'. Note that for each configuration C, there is at least one enabled step for each process. There may be more than one enabled step for a single process if the algorithm executes a nondeterministic operation. For example, a relaxed $Enqueue$ may lead to several different child configurations depending on where the argument is placed in the queue.

An *execution* of an algorithm A is an infinite sequence of steps, starting from an initial configuration, with the new configuration of each step equal to the old configuration of the next step. Processes that take only a finite number of steps are said to crash. The requirement that executions are infinite implies that at least one process does not crash. If a process terminates the algorithm successfully, we say that it triggers an infinite series of no-op steps at that process. A *reachable configuration* is one that appears in some execution.

Let C be a configuration reachable by some prefix E of an execution of a consensus algorithm A. Consider all executions E' which are extensions of E. A must terminate, so in each E', some value is decided. Let $vals(C)$ be the set of values decided in all E's. We call C *bivalent* if $vals(C) = \{0, 1\}$, *1-valent* if $vals(C) = \{1\}$, and 0-valent if $vals(C) = \{0\}$. We call C *critical* if it is bivalent, but every child configuration of C is univalent.

Lemma 1 ([15, 16])**.** *Every critical configuration has child configurations with different valencies which are reached by different processes acting on the same shared object, which is not a register. Further, every enabled step in a critical configuration must be a mutator.*

Lemma 1's claim that steps leading to different valencies must exist at different processes is trivial for deterministic types, since each process can have only one enabled step. With nondeterministic types, a single process may have multiple enabled steps from a single configuration.

Here, the lemma follows from the fact that there must be at least one 0-valent child configuration and at least one 1-valent child configuration. If these are not at different processes but both at the same process, then the valency of a step by some other process can be neither 0 nor 1, contradicting the definitions of valency and critical configurations.

Lemma 2 (Extended from [17])**.** *A consensus algorithm*

- *always has an initial bivalent configuration and*
- *must have a critical configuration in every execution*

Lemma 3 (Univalency Lemma, implicit in [15])**.** *If two univalent configurations are indistinguishable to a process, they have the same valency.*

3. IMPLEMENTING RELAXED QUEUES *

3.1 Introduction and Related Work

In this chapter, we explore the possible performance benefits of relaxed data structures. We focus on the elapsed time for operations when the shared object is implemented in a message-passing system with bounded message delays and approximately synchronized clocks. In contrast, [4], which introduced these relaxations, considered shared memory implementations of relaxed shared objects. To our knowledge, we are the first to consider message-passing implementations of the relaxations.

First, we prove that for a general class of operations, the worst-case elapsed time must be at least d, the maximum message delay. We then show that for three of the relaxations we consider (Lateness, Restricted-Out-of-Order, and Stuttering), the $Dequeue$ operation of the FIFO queue data type falls into this class and thus must take at least d time. This lower bound indicates that, with respect to worst-case time for operations, there is marginal gain, at best, from these relaxations, as recent work [3] has shown that an unrelaxed FIFO queue can be implemented with worst-case time for $Dequeue$ at most $d + \epsilon$, where ϵ is the maximum skew between local clocks.

In light of this negative result regarding worst-case time for $Dequeue$s, we next consider *amortized* time, in the hope that relaxed data types would require expensive synchronization less frequently. To show this benefit, we use aggregate analysis, dividing the sum of all operation times by the number of opreations in a run. As a first step, we focus on shared queues. We consider two relaxations from [4], applied to $Dequeue$. Each relaxation has an integer parameter $k \geq n$, where n is the number of processes. We present an algorithm for implementing an Out-of-Order k-relaxed queue in which the amortized time for $Dequeue$ is $d/\lfloor \frac{k}{n} \rfloor + \epsilon$. We also present an algorithm for implementing a Restricted-Out-of-Order k-relaxed queue in which the amortized time

*Parts of the material in this chapter are reprinted from
E. Talmage and J.L. Welch, "Improving average performance by relaxing distributed data types," in *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings* (F. Kuhn, ed.), vol 8784 of *Lecture Notes in Computer Science*, pp. 421-438, Copyright 2014 by Springer.

for $Dequeue$ is $(2d + \epsilon)/\lfloor \frac{k}{n} \rfloor + \epsilon$. In both cases, the amortized time for the $Dequeue$ operation is significantly below the worst-case lower bound of d, and decreases as k increases. In contrast, we show that the best possible amortized time for $Dequeue$ in an unrelaxed queue must be at least $d(1 - \frac{1}{n})$, indicating that relaxation does pay when considering amortized time.

We further show a lower bound of $d/\lfloor \frac{k}{n} \rfloor$ on the amortized time for $Dequeue$ for the same two relaxations, still for $k \geq n$, which indicates that one of our algorithms is optimal while the other is within a factor of two of optimal. Our upper and lower bounds on amortized $Dequeue$ time imply that there is an *inherent* performance benefit achievable by increasing k for these two forms of relaxation, as the lower bounds for any fixed value of k are larger than the corresponding upper bounds for sufficiently greater values of k. In contrast, the results in [18],[4] show performance improvements for shared memory implementations, based on experimental analyses of specific algorithms; no lower bounds are shown.

Unlike prior work proving lower bounds on the time complexity of operations (e.g., [19, 20, 21, 3]), ours are for nondeterministic data types. Nondeterminism is harder to deal with, as one cannot always rely on some operation returning a certain value. In our proofs, we take care to argue that the object can be "boxed into a corner" under certain circumstances, so that there is only one possible right answer.

Table 3.1 summarizes the known bounds on the elapsed time for $Dequeue$. Section 3.2 states some further model assumptions we need in this chapter. In Section 3.3, we prove lower bounds on the worst-case time for operations. Our two algorithms and their average-cost analyses are presented in Section 3.4. Section 3.5 contains our lower bounds on the average-cost time for $Dequeue$s, and we conclude in Section 3.6.

3.2 Correctness Condition

We are interested in developing linearizable algorithms. Recall from Chapter 2, we also require that algorithms satisfy a liveness conditions, that there is a bijection between operation invocations and responses in any complete, admissible run.

The *worst-case time complexity* of operation OP, denoted $|OP|$, is defined as the the maximum

Table 3.1: Bounds on $Dequeue$ Time Complexity

	Worst Case Cost		Average Cost	
	Lower Bound	Upper Bound	Lower Bound	Upper Bound
FIFO Queue	$d + \min\{\epsilon, u, \frac{d}{3}\}$ [3]	$d + \epsilon$ [3]	$d(1 - \frac{1}{n})$ (Sec 5.1)	$d + \epsilon$ [3]
Out-of-Order	?	$d + \epsilon$ [3]	$\frac{d}{\lfloor k/n \rfloor}$ (Sec 5.2; $k < n^2$)	$\frac{d}{\lfloor k/n \rfloor} + \epsilon$ (Sec 4.2; heavily-loaded)
Lateness	d (Sec 3)	$d + \epsilon$ [3]	?	?
Restricted-Out-of-Order	d (Sec 3)	$d + \epsilon$ [3]	$\frac{d}{\lfloor k/n \rfloor}$ (Sec 5.3)	$\frac{2d+\epsilon}{\lfloor k/n \rfloor} + \epsilon$ (Sec 4.3; heavily-loaded)
Stuttering	d (Sec 3)	$d + \epsilon$ [3]	?	?

over every instance of OP in every complete admissible run, of the real time that elapses between the the invocation of the instance and its response. The *amortized time complexity* of operation OP in is the least upper bound, over every complete admissible run R, of the sum over all complete instances of OP in any prefix of R of the real time that elapses between the invocation and response of the instance, divided by the total number of instances of OP in the prefix of R.

3.3 Worst-Case Lower Bound

First, we will show a lower bound on the worst case time complexity for a class of operations which includes some of the relaxed $Dequeue$s defined in Section 2.1.1. This lower bound is nearly equal to the upper bound given in [3] for arbitrary data types. This shows that there is negligible, if any, benefit from relaxation, with regard to this complexity measure.

To show this bound, we consider runs carefully structured so that the sequential specification of the data type gives tight limits on what values are legal to return. By simultaneously invoking multiple operation instances, we can use an indistinguishability argument to show that at least one of the instances must delay returning long enough to learn about another instance.

Definition 16. *Define an operation OP to be* non-repeatable with respect to ρ *if there exists a sequence of operation instances ρ and an instance $op = OP(arg, ret) \in OP$ such that $\rho \cdot op$ is*

legal and no $ret' \neq ret$ is a legal return value for $\rho \cdot OP(arg)$, but $\rho \cdot op \cdot op$ is not legal.

Theorem 1. *In any distributed shared memory implementation A, if there is an admissible run with linearization of its operation instances ρ and operation OP which is non-repeatable with respect to ρ, then $|OP| \geq d$.*

Proof. Let ρ and op be a sequence of operation instances and an instance of op, as in Definition 16. Construct three runs, as follows:

- R: Some process p_0 sequentially invokes and returns to each operation in ρ. Let t be the real time when the last instance returns.
- R_0: From time 0 to time t, R_0 is identical to R. After time t, all message delays are d. At time t, have p_0 invoke $OP(arg)$. By the definition of non-repeatability, that instance must return the value ret defined in Definition 16.
- R_1: From time 0 to time t, R_1 is identical to R. After time t, all message delays are d. At time t, p_1 invokes $OP(arg)$, instead of p_0. This instance must return ret to this invocation, by the definition of non-repeatability.

Finally, define run R exactly as both R_0 and R_1, except that at time t, both p_0 and p_1 invoke $OP(arg)$. Call these two instances op_0 and op_1, respectively. By the assumption that $|OP| < d$, we know that op_0 and op_1 must both return by real time $t + d$. Thus, they cannot learn about each other, since messages caused by either instance take d time in transit. Thus, R_0 and R' are indistinguishable to p_0 through the return of op_0 and R_1 and R' are indistinguishable to p_1 through the return of op_1. op_0's return value must then be ret, and op_1's also must. But now, the only linearizations of run R' are $\rho \cdot op_0 \cdot op_1$ and $\rho \cdot op_1 \cdot op_0$, since all instances in ρ overlap with no other instances. But these are both equal to $\rho \cdot op \cdot op$, which is illegal, by definition. Thus, $|OP| \geq d$. □

We next show that several versions of relaxed $Dequeue$ satisfy the hypothesis of Theorem 1.

Lemma 4. *For any algorithm implementing a queue with a Lateness k-relaxed $Dequeue$, there is some admissible run with linearization ρ such that $Dequeue$ is non-repeatable with respect to ρ.*

Proof. Consider the following sequence on a Lateness k-relaxed queue: $\rho = Enqueue(1,-) \cdot Enqueue(2,-) \cdots Enqueue(k+2,-) \cdot Dequeue(-,2) \cdot Dequeue(-,3) \cdots Dequeue(-,k)$. This sequence is legal, since lateness is always less than k, $Enqueue$ arguments and $Dequeue$ return values are unique, and $Dequeue$s return the arguments of previous $Enqueue$s. At the end of ρ, lateness is $k-1$, so if the next operation is $Dequeue$, then it must return 1, to reset lateness. Thus, $\rho \cdot Dequeue(-,1)$ is legal and $\rho \cdot Dequeue(-,x)$ is not legal for any $x \neq 1$. Further, $\rho \cdot Dequeue(-,1) \cdot Dequeue(-,1)$ is not legal, since every non-$\perp$ return value of a $Dequeue$ instance must be distinct, by condition *(C2)* of the definition of a Lateness k-relaxed queue. Thus, $Dequeue$ is non-repeatable. □

Corollary 1. *In any implementation of a Lateness k-relaxed queue, $|Dequeue| \geq d$.*

Lemma 5. *For any algorithm implementing a queue with a Restricted Out-of-Order k-relaxed $Dequeue$, there is some admissible run with linearization ρ such that $Dequeue$ is non-repeatable with respect to ρ.*

Proof. As in the proof of Lemma 4, consider the operation instance sequence $\rho = Enqueue(1,-) \cdot Enqueue(2,-) \cdots Enqueue(k+2,-) \cdot Dequeue(-,2) \cdot Dequeue(-,3) \cdots Dequeue(-,k)$, but this time on a Restricted Out-of-Order k-relaxed queue. This sequence is legal, since $Enqueue$ arguments and $Dequeue$ return values are unique, each $Dequeue$ returns the argument of a previous $Enqueue$, and the return value is always among the first k arguments to an $Enqueue$ after $Enqueue(1)$. At the end of ρ, $Enqueue(1)$ is the first unmatched $Enqueue$ instance, and the only one of the k $Enqueue$s in the suffix of ρ starting with $Enqueue(1)$ which has not had its argument returned by a $Dequeue$. Thus, $\rho \cdot Dequeue(-,1)$ is legal, and $\rho \cdot Dequeue(-,x)$ is not legal for any $x \neq 1$. Further, $\rho \cdot Dequeue(-,1) \cdot Dequeue(-,1)$ is not legal, since no two $Dequeue$ instances may return the same value. Thus, $Dequeue$ in a Restricted Out-of-Order k-relaxed queue is non-repeatable. □

Corollary 2. *In any implementation of a Restricted Out-of-Order k-relaxed queue, $|Dequeue| \geq d$.*

The arguments used so far in this section to show a lower bound of d on the worst-case time complexity for relaxed versions of the $Dequeue$ operation of relaxed queues can be generalized to operations that remove elements from a set of elements. Consider any data type which maintains a set of current elements and has at least two operations, one to add elements and one to remove. Suppose further that the remove operation cannot remove any single element more than once. Finally, constrain the set of legal sequences of operation instances so that repeatedly invoking the remove operation must eventually remove every element in the set. Then we can show that the remove operation is non-repeatable with respect to some operation sequence, and thus the remove operation has worst-case time complexity at least d.

3.4 k-Relaxed Algorithms for Queues

We have shown that, with regard to the metric of worst-case operation time, there is no useful gain from relaxation of some common data types. This is due to the fact that distributed storage must still synchronize itself at times. But, in a relaxed data type, the required coordination may not be quite as close, so synchronization may not be required as often. We give two algorithms which exploit this lesser synchronization requirement to achieve better amortized operation cost, where the improvement scales with the degree of relaxation.

3.4.1 Local Variables

We specify the local variables our algorithms will use. Both algorithms use the same local variables, with the addition of $available$ fields on $lQueue$ elements and $heads_j$ arrays for Algorithm 3-4.

- $clean$: Boolean, initially true
- $lQueue$: Local copy of data structure, initially empty. Values have two associated fields: a $label$ field which is initially $null$ and can hold a process id and a Boolean $available$, initially true. Behavior is an extensions of a local (non-distributed) FIFO queue. Operations:

- $enq(val)$: inserts val
- $deqByLabel(p_j)$: removes and returns headmost (oldest) element labeled p_j, $\perp$ if none exists
- $peekByLabel(p_j)$: returns, without removing, headmost element labeled p_j, $\perp$ if none exists
- $deqBySet(S)$: removes and returns headmost element in $lQueue$ which is also in the set S
- $peekBySet(S)$: returns, without removing, the headmost element in both $lQueue$ and S
- $contains(val)$: returns true if val is in $lQueue$, false otherwise
- $size()$: returns current number of elements
- $sizeByLabel(p_j)$: Returns number of elements with label p_j
- $unlabeledSize()$: returns current number of unlabeled elements
- $tail()$: returns, without removing, the last element added
- $remove(val)$: removes val
- $label(p_j, val)$: label val with p_j
- $labelOldest(p_j, x)$: labels the oldest x elements with p_j

- $Pending$: Priority queue to hold operation instances, keyed by timestamp; initially empty. Supports standard operations $insert(val, ts)$, $min()$, $extractMin()$
- $heads_j[], 0 \leq j < n$: Arrays of data elements of size n, initially empty

We will use the parameter l, defined as $\lfloor \frac{k}{n} \rfloor$ throughout this section.

3.4.2 Out-of-Order Relaxed Queues

First, we give an algorithm for an Out-of-Order k-relaxed queue. This algorithm introduces the basic idea behind our later algorithm for Restricted Out-of-Order k-relaxed queues, and is

presented in Algorithms 1 and 2. This algorithm assumes $k > n$, and gives improved amortized performance over algorithms for unrelaxed queues, increasing as k increases by multiples of n. The algorithm is designed to gracefully degrade performance as it runs out of elements, since a k-relaxed Dequeue on a queue with fewer than k elements is not very meaningful. Instead, there will be an effectively lower k (down to a minimum of n) until the size of the queue grows sufficiently. This also allows us to use fast Enqueues at all times.

The algorithm is inspired by the algorithm from [3]. To allow quick returns of most operation instances, giving good amortized performance, we distribute the headmost k elements of the queue, which are legal to return at any given time, evenly among the processes. Each process can quickly return those elements assigned to it, then must synchronize to obtain more. When a process needs to return an element, due to a $Dequeue$, it returns the headmost element labeled with its own id. If there are no elements so labeled, then the process will not return until it has waited long enough to learn about concurrent and recent operations at other processes, effectively synchronizing, as every $Dequeue$ did in the algorithm of [3].

When a process tries to $Dequeue$, but has no local elements available and must synchronize, as part of its operation, it labels more elements for itself. No more than k elements are ever labeled, and for exactly k to be labeled, each process must have l elements labeled[1]. Thus, since the current process has no labeled elements, it is safe to claim more, up to a total of l, since then there will be at most k elements labeled, so every future operation returning a labeled element will return a legal element, according to the relaxation.

Before any elements are dequeued (while $clean$ is true), $Enqueue$ operations label up to k elements in round-robin fashion. This allows the first $Dequeue$ invoked to return quickly, since it will find elements labeled with its invoking process. After a $Dequeue$ is invoked, we mark the queue as dirty ($clean = false$) and no longer label elements during $Enqueue$s, because round-robin order may not be maintained if $Dequeue$s are not invoked evenly across all processes. This maintains good average performance in executions which may only perform a few $Dequeue$s.

[1] Or $l + 1$ elements when k is not an exact multiple of n, and elements have been labeled by $Enqueue$s.

Algorithm 1 Code for each process p_i to implement a queue with Out-of-Order k-relaxed $Dequeue$, where $k \geq n$ and $l = \lfloor k/n \rfloor$.

```
1:  HandleEvent ENQUEUE(val)
2:      send (enq, val, ⟨localTime, i⟩) to all
3:      setTimer(ϵ, ⟨enq, val, ⟨localTime, i⟩⟩, respond)
4:  HandleEvent DEQUEUE
5:      if lQueue.peekByLabel(p_i) ≠ ⊥ then
6:          ret = lQueue.peekByLabel(p_i)
7:          ret.available = false
8:          send (deq_f, ret, ⟨localTime, i⟩) to all
9:          setTimer(ϵ, ⟨deq_f, ret, null⟩, respond)
10:     else
11:         send (deq_s, null, ⟨localTime, i⟩) to all
12: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, respond)
13:     if op == deq_f then Generate response for Dequeue with return value val
14:     else Generate response for Enqueue with return value ACK
15: HandleEvent RECEIVE (op, val, ts) FROM p_j
16:     Pending.insert(⟨op, val, ts⟩)
17:     setTimer(u + ϵ, ⟨op, val, ts⟩, execute)
18: HandleEvent EXPIRETIMER(⟨op, val, ts⟩, execute)
19:     while ts ≥ Pending.min() do
20:         ⟨op′, val′, ts′⟩ = Pending.extractMin()
21:         executeLocally(op′, val′, ts′)
22:         cancelTimer(⟨op′, arg′, ts′⟩, execute)
```

▷ continued

When there are fewer than k elements left in the queue, a synchronizing $Dequeue$ will act as if k were $lQueue.size$. This means that it labels fewer elements for itself, allowing even performance across all processes. This behavior is adopted, as having a k larger than the current size of the queue means that every element is legal to return.

3.4.2.1 Out-of-Order Relaxation Correctness

Throughout this section, line numbers refer to Algorithms 1 and 2.

Let $ts(op)$ denote the $(localTime, i)$ pair, called a ***timestamp***, associated with an operation instance in line 2, 8, or 11.

Construction 1. *Define the permutation π of operation instances in a complete, admissible run of*

Algorithm 2 Continuation of algorithm implementing a queue with an Out-of-Order k-relaxed $Dequeue$.

```
23: function EXECUTELOCALLY(op, val, ⟨∗, j⟩)
24:     if op == enq then
25:         lQueue.enq(val)
26:         if clean == true and lQueue.size() ≤ k then
27:             let a = (lQueue.size() − 1) mod n
28:             lQueue.label(p_a, val)
29:     else
30:         clean = false
31:         if op == deq_f then lQueue.remove(val)
32:         else
33:             if lQueue.peekByLabel(p_j) ≠ ⊥ then
34:                 ret = lQueue.deqByLabel(p_j)
35:             else
36:                 ret = lQueue.deqByLabel(null)
37:             labelElements(j)
38:             if j == i then Generate response for Dequeue with return value ret
39:         if lQueue.size() == 0 then clean = true
40: function LABELELEMENTS(j)
41:     y = lQueue.unlabeledSize()
42:     lQueue.labelOldest(p_j, x), where x = min{l, ⌊lQueue.size()/n⌋, y}
```

Algorithm 1-2 as the order given by sorting by $ts(op)$ for each instance op.

We first show that all processes execute operation instances on their local copy of the shared queue in timestamp order and that Construction 1 will respect the real-time partial order of non-overlapping operation instances. These two facts allow us to prove that timestamp order is a legal linearization of all instances in a run.

Lemma 6. *Each process locally executes all operation invocations ($\langle op, arg, ts\rangle$) in timestamp order.*

Proof. Suppose we have two $Enqueue$ or $Dequeue$ invocations $op_1 = \langle opName_1, arg_1, ts_1\rangle$ and $op_2 = \langle opName_2, arg_2, ts_2\rangle$, with $ts(op_1) < ts(op_2)$. If some process p_i locally executes op_2 before op_1, then op_1 cannot have been in p_i's $Pending$ queue when it executed op_2, by lines 18-22. But op_2 cannot be executed less than $d + \epsilon$ time after it was invoked, by line 17, unless the $execute$

timer for some invocation op_3 with timestamp larger than op_2 went off. That can only occur at least $d + \epsilon$ after op_3's invocation.

Because the maximum difference between local clocks is ϵ, op_1's invocation can be at most ϵ real time after op_2's, since $ts(op_1) < ts(op_2)$. Then p_i will learn about op_1, and insert it into its $Pending$ queue at most $d + \epsilon$ real time after op_2's invocation. This would imply that if p_i locally executes op_2 at its own $execute$ timer, then it has op_1 in its $Pending$ queue, and would locally execute op_1 before op_2, by the code in the $execute$ timer handler.

Similarly, if p_i locally executes op_2 when op_3's $execute$ timer expires, op_1 could have been invoked at most ϵ real time after op_3 and still have a smaller timestamp. Thus, p_i will learn about op_1 no later than $d + \epsilon$ after op_3's invocation, which means that p_i has op_1 in its $Pending$ queue when it locally executes op_3, which is also when it locally executes op_2, contradicting the $execute$ timer handler. Thus, every process executes op_1 before op_2. □

Since all processes locally execute all invocations in the same order, we can argue that their local copies of the shared queue, represented by their $lQueue$ variables, take on the same sequence of states.

Lemma 7. *After locally executing a prefix ρ of the sequence π given by Construction 1, the $lQueue$ variable of every process is identical, excluding $available$ fields.*

Proof. The only place the algorithm edits $lQueue$ outside `executeLocally` is in line 7, which only changes available fields, and in the `labelElements` function, which is only called from inside `executeLocally`. Every process has the same sequence of calls, including arguments, to `executeLocally`, which is itself deterministic, so we need only show that each call to `executeLocally` yields the same result at every process, despite possible differences in available fields.

The only places within a call to `executeLocally` whose behavior may depend on available fields are lines 34, 36, and 41, the last in the `labelElements` function. Lines 36 and 41 do not actually change behavior based on available fields in this algorithm, since only labeled elements

are ever marked unavailable (line 7), and labels are never removed. Since these lines only interact with unlabeled elements, their behavior is independent of available fields.

Line 34 would have different behavior at different processes if an element labeled p_j was marked unavailable at some processes but not others while the current slow $Dequeue$, deq, is locally executed at each process. This could only happen, though, if p_j had invoked a fast $Dequeue$ which marked the element as unavailable. This must have happened since deq was invoked, since by definition of a slow $Dequeue$, there could not have been any elements labeled p_j at deq's invocation. But slow $Dequeue$s do not respond until they are locally executed, so p_j could have invoked no such fast $Dequeue$. Thus, there cannot be an element labeled p_j which is marked unavailable at some processes but not others when they each locally execute a slow $Dequeue$, and line 34 will behave the same way at every process in the local execution of the same instance.

Since the behavior of `executeLocally` does not change with the possible variations in available fields between processes, the $lQueue$ variables of each process will be the same, excluding available fields, after locally executing the same sequence of operation instances. □

Lemma 7 implies that for each $Dequeue$, every process removes the same return value from its $lQueue$, since they must remove the value from their $lQueue$ they percieve the $Dequeue$ as returning and the $lQueue$s of all processes are in the same state after locally executing the same sequence of operation instances.

Now, we can consider the parts of the definition of linearizability. We show that our constructed ordering respects the real-time order of instances, and that it is legal, by the sequential specification of an Out-of-Order k-relaxed queue to show that the algorithm is a correct, linearizable implementation.

Lemma 8. *Construction 1 respects the real-time order of non-overlapping operation instances.*

Proof. Suppose on the contrary that there are two operation instances op_1 and op_2 with $ts(op_1) < ts(op_2)$ but op_2 returns before op_1's invocation. By the response timers in lines 3 and 9 and the message delay assumptions, every instance takes at least ϵ time to respond. Thus, op_2 must have

been invoked more than ϵ before op_1. Since local clocks are within ϵ of each other, the local time at op_2's invocation must be less than the local time at op_1's invocation, so $ts(op_2) < ts(op_1)$, a contradiction. □

Theorem 2. *For any complete, admissible run of Algorithm 1-2, the permutation π given by Construction 1 is legal by the specification of an Out-of-Order k-relaxed queue.*

Proof. By Lemmas 6, 7, the sequence of local executions at each process is the same sequence as π defined in Construction 1, and Lemma 7 further implies that the $lQueue$s at all processes are the same after the same prefix of π is locally executed and all processes remove the same value for each $Dequeue$ instance. Thus, we need only argue that the value which processes choose to remove and return for a particular $Dequeue$ is legal, in the sense that the prefix of π ending with that $Dequeue$ instance is legal, by the specification of an Out-of-Order k-relaxed queue.

To do this, we argue that each process' execution of the algorithm maintains a *labeling invariant*. We prove that every element x in a process' $lQueue$ which has a non-$null$ label after locally executing a prefix ρ of π is an acceptable return value for a $Dequeue$, meaning $\rho \cdot Dequeue(-, x)$ is legal on an Out-of-Order k-relaxed queue.

Our proof is by induction on local execution of π, the sequence of all operation instances in an arbitrary complete, admissible run. At each step of the induction, we must show two things: first, that the return value chosen yields a legal sequence; second, that the labeling invariant still holds. The base case is an empty sequence, which trivially satisfies legality and the labeling invariant. We then assume that a prefix ρ of π is legal and consider local execution of an instance op, such that $\rho \cdot op$ is a prefix of π. We proceed by cases on the operation of instance op.

$Enqueue$: In the sequential specification of the queue, an $Enqueue$ is always legal. We thus have only to show that the labeling done by $Enqueue$s satisfies the labeling invariant. If $clean == true$ when processes locally execute op, this follows, since the code labels in round-robin fashion (line 27), and only labels while there are no more than k elements in the queue (line 26).

Locally executing a $Dequeue$ will set the $clean$ variable to false at all processes(line 30), so $Enqueue$s locally executed after a $Dequeue$ will no longer label elements until $clean = true$

again. Since we assumed that after local execution of ρ, for any labeled element x, the extended sequence $\rho \cdot Dequeue(-, x)$ is legal, and since an $Enqueue$ cannot make a value unacceptable for a $Dequeue$ to return, by the specification of an Out-of-Order k-relaxed queue, the labeling invariant holds after $\rho \cdot op$.

$Dequeue$: Suppose op is a $Dequeue$. We first show that when a process locally executes op, it chooses a return value which satisfies *(C2)-(C4)* of Definition 4, implying that $\rho \cdot op$ is legal.

(C2) op returns a value that was stored in $lQueue$ and removes it. (Lines 6, 31, 34, 36). Since elements are only added to $lQueue$ by instances of $Enqueue$ (line 25), *(C1)* guarantees that the values in $lQueue$ are unique.

No two $Dequeue$ instances can return the same value. Any value ret chosen as a $Dequeue$'s return value is immediately either removed from $lQueue$ (lines 34, 36) or marked as unavailable (line 7) in the $lQueue$ of the process with which ret is labeled, preventing another $Dequeue$ instance invoked at that process from returning ret. No existing labels are ever changed, since line 42 only labels unlabeled elements and labels are not set anywhere else, so no $Dequeue$ at another process p_j can return ret, since it only returns elements labeled p_j (lines 6, 34, 36). Thus, the return values of $Dequeue$ instances are unique.

(C3) This follows similarly, since $Dequeue$ will return a value from $lQueue$, which can only have been put there by an instance of $Enqueue$ (Line 25) which is previous in π, since each process executes instances of $Enqueue$ and $Dequeue$ in timestamp order, the order specified in π.

(C4) If there is an element labeled p_i in $lQueue$ when op is invoked at process p_i, then op will return the oldest such element after a delay of only ϵ (Line 9). When op is invoked, p_i has locally executed some prefix ρ' of ρ. By the labeling invariant, then, every labeled element can be returned to a $Dequeue$ occurring in π immediately after ρ'. op may not be immediately after ρ' in π, but once an element is labeled, its label never changes, since labels are only

edited in line 42, which only labels unlabeled elements. Thus, after ρ is locally executed, the element returned for op will still be labeled, so $\rho \cdot op$ is legal.

If there is no element labeled p_i at op's invocation, it is a slow $Dequeue$. There are two possible ways for a slow $Dequeue$ to choose its return value (Lines 34 and 36). In the first, some other operation has labeled elements between the invocation and execution of this $Dequeue$. It will then choose a labeled element as its return value during local execution, so by the labeling invariant, it returns a legal value.

In the second case, the $Dequeue$ will return an unlabeled element, possibly $\perp$. If this happens, we know there are fewer than k labeled elements, since the algorithm only ever labels a maximum of k elements (Lines 26 and 42, and the definition of l), including at least one at each process and there are currently none labeled p_i.

To see that returning the headmost unlabeled element in $lQueue$ yields a legal sequence, note that since instances of $Enqueue$ are locally executed in the order specified by π, the headmost elements in $lQueue$ will be those whose $Enqueue$s appear earliest in π. Since instances of $Dequeue$ are also executed in the order given by π and remove their return values from $lQueue$, the elements in $lQueue$ are the arguments of $Enqueue$s which appear previously in π and do not have corresponding $Dequeue$s, in order. Thus, the headmost k elements in $lQueue$ are legal to return by ***(C4)***, so it is legal for the $Dequeue$ to return the headmost unlabeled element when there are no elements labeled p_i. If the headmost unlabeled element is $\perp$, there are fewer than k $Enqueue$ instances without corresponding $Dequeue$ instances appearing previously in π, so $\perp$ is a legal return value.

It only remains to show that $Dequeue$s maintain the labeling invariant. A fast $Dequeue$ does not label any new elements, and no elements cease to be legal after a $Dequeue$ because it decreases the number of $Enqueue$s without a matching $Dequeue$ in π, as referenced in ***(C4)*** for Out-of-Order queues. Thus, the labeling invariant continues to hold after a fast $Dequeue$.

For slow $Dequeue$s, we see that when a $Dequeue$ labels elements (line 37), it only labels up to l elements from $lQueue$, for itself. Since elements are inserted into $lQueue$ in timestamp order of $Enqueue$s, which is the order the $Enqueue$s appear in π, and are removed by $Dequeue$s, also in order matching π, the headmost k elements in $lQueue$ are the first k elements with an $Enqueue$ in π but no corresponding $Dequeue$, and are thus legal to return. Since new labels are always applied to the headmost unlabeled elements of $lQueue$, if there are fewer than k labels, the headmost unlabeled element is legal to return. Thus, labeling it maintains the labeling invariant. □

Theorem 3. *Algorithm 1-2 implements an Out-of-Order k-relaxed queue.*

3.4.2.2 Out-of-Order Relaxation Performance

Definition 17. *We will call a run* heavily loaded *if every $Dequeue$ is linearized after a prefix of π in which there are at least k more instances of $Enqueue$ than instances of $Dequeue$.*

Theorem 4. *The amortized time complexity of $Dequeue$ in any heavily-loaded complete, admissible run of Algorithm 1-2 is no more than $\frac{d}{l} + \epsilon$, where $l = \lfloor k/n \rfloor$.*

Proof. Consider the view of a single process p_i. When p_i first invokes a $Dequeue$, there are l elements in p_i's copy of $lQueue$ labeled p_i, since the first k elements enqueued are labeled in round-robin fashion when enqueued. Thus, the first l $Dequeue$s p_i performs will be fast, taking ϵ time each.

The $(l+1)$th $Dequeue$ will have to synchronize, since there will be no elements in p_i's $lQueue$ labeled p_i. After this slow $Dequeue$, there will be either l elements labeled p_i, or $l-1$ if the slow $Dequeue$ returned an element labeled p_i. The pattern of $l-1$ or l fast $Dequeue$s, which only require local computation, followed by one slow $Dequeue$ which synchronizes will then repeat. The average cost of each repeat of this pattern is no more than

$$\frac{l\epsilon + d + \epsilon}{l} = \frac{d}{l} + \epsilon$$

We can upper bound the average cost of any prefix of the infinite repetition of this pattern of fast

and slow $Dequeue$s by the maximum average cost of a single copy of the pattern, since a prefix will have the highest average cost when it ends with the slow $Dequeue$ at the end of the pattern.

Since this is an upper bound on the average cost of operations at any process p_i, the amortized cost of operations at all processes will also be bounded above by $\frac{d}{l} + \epsilon$. □

While the specification of an Out-of-Order k-relaxed queue allows any number of $Dequeue$ instances to return $\perp$ when there are fewer than k elements in the queue, this may not be practically very useful. Instead, Algorithm 1-2 provides a stricter guarantee in order to gracefully degrade performance when the queue is nearly empty. This provides more intuitive behavior, making the algorithm potentially more useful. We next describe the time complexity of $Dequeue$ in the general case, when there may be fewer than k elements in the queue.

Definition 18. *The* effective l, denoted l_e, *of a portion of a time interval in a complete, admissible run is set to*

1. $lQueue.size()/n$, *when a* $Dequeue$ *locally executes when* $clean$ *is true (line 30), or*
2. $\min\left\{l, \left\lfloor \frac{lQueue_i.size()}{n} \right\rfloor, lQueue_i.unlabelled\_size()\right\}$, *when a slow* $Dequeue$ *labels elements (line 42)*

and remains until a new effective l *is set.*

Thus, every $Dequeue$ instance in a linearization has an effective l determined by the state of the simulated queue at its local execution. We next show that each process' $Dequeue$s maintain amortized performance determined by their effective l. The proof is very similar to the proof of Theorem 4, using the fact that there are at most l_e elements labeled p_i at any time.

Theorem 5. *Consider a complete, admissible run of Algorithm 1-2. During a time interval in which the effective* l *is* l_e*, the amortized time complexity of* $Dequeue$*s is at most* $\frac{d}{l_e} + \epsilon$.

3.4.3 Restricted Out-of-Order Relaxed Queues

We now present an algorithm which implements a queue with Restricted Out-of-Order k-relaxed Dequeues, for $k \geq n$. The pseudocode appears as Algorithms 3 and 4. This algorithm

uses the idea of locally distributing the oldest elements in the queue to allow processes to return quickly several times, before they must take time to synchronize their state with other processes. In addition, the algorithm uses the synchronizing operations to guarantee $Dequeue$s return the head with sufficient frequency. Doing this imposes extra cost on some operations, because they effectively may be forced to "steal" the head element from another process. The algorithm still has good amortized performance for sufficiently large k, and performance which improves monotonically as k increases.

The algorithm assigns elements to different processes by labeling them with process ids. The correctness argument depends on an invariant of the labeling: every element which has a label in the local state of a process is legal for an instance of $Dequeue$ by that process to return. Further, labeled elements will only be returned by the process with whose id they are labeled, unless another process goes through an expensive synchronization process to steal them. Thus, if a process finds an element labeled with its own id, it can generally return it quickly without waiting to coordinate with other processes.

If a $Dequeue$ does not find any elements labeled with its invoking process' id, then it must find another element to return, making it a slow $Dequeue$, since this will be expensive and require synchronization. A slow $Dequeue$ ensures that the head element in the simulated queue is removed, either by itself or by another, concurrent slow $Dequeue$. When a process does return the head element to a $Dequeue$, by the definition of a Restricted Out-of-Order k-relaxed queue the headmost k elements in the simulated queue are now legal to return, so the process labels them. Thus, after a process executes a slow $Dequeue$, there will be more elements labeled with its id, if there are enough elements currently in the queue.

To ensure the head is returned, a process p_i which invokes a slow $Dequeue$ notifies all other processes of the operation instance. Each other process p_j will mark the element labeled p_j which is nearest the head of the simulated queue (*headmost*) as unavailable for fast local return and broadcast it to all, marking it as being relevant to a slow $Dequeue$ invoked at p_i. We call this element the *local head* for p_j. Timers in the algorithm are set such that every process will receive

every other process' local head before it tries to execute the slow $Dequeue$. Since, if there are any labeled elements, processes label elements from the head of the queue without skipping, then the head in the queue will be some process' local head. Then when a slow $Dequeue$ is executed, it will return the head of the entire queue, unless another, concurrent slow $Dequeue$ has already returned it. In this case, the later slow $Dequeue$ need not worry about returning the global head or labeling elements, and can return any of the local head elements, since they are reserved by their processes.

Since $Dequeue$s synchronize as needed when the simulated queue empties, $Enqueue$s do not need to synchronize. Thus, we always have fast $Enqueue$s.

3.4.3.1 Restricted Out-of-Order Relaxation Correctness

Throughout this section, line numbers refer to Algorithms 3 and 4.

We must show that the algorithm gives return values for operation instances which allow there to be a permutation of the instances which respects both the real-time order of instances and the sequential specification of the relaxed data type. We construct a permutation of the complete operation instances the algorithm executes and show that it meets these requirements.

Construction 2. *Define the sequence π of operation instances in a complete, admissible run of Algorithm 3-4 as the order given by sorting by $ts(op)$ for each instance op.*

First, we will show that Construction 2 respects the real-time order of non-overlapping operation instances. Then, by showing that all processes locally execute all operation instances in the order given by π in Construction 2 and that after locally executing the same prefix of π, their local variables are the same, we show that the algorithm chooses a legal return value for every instance, implying that the algorithm is correct.

Lemma 9. *As defined in Construction 2, the sequence π respects the real-time order of non-overlapping operation instances.*

Proof. Every operation takes at least ϵ time to return, by the timers in lines 3 and 9 and the message delay, and the difference between any two processes' local clocks is upper bounded by ϵ. Thus, if

Algorithm 3 Code for each process p_i to implement a queue with Restricted Out-of-Order k-relaxed $Dequeue$s for $k \geq n$, where $l = \lfloor k/n \rfloor$.

1: **HandleEvent** ENQUEUE(val)
2: send $(enq, val, \langle localTime, i\rangle)$ to all
3: $setTimer(\epsilon, \langle enq, val, \langle localTime, i\rangle\rangle, respond)$
4: **HandleEvent** DEQUEUE
5: **if** $lQueue.peekByLabel(p_i) \neq \perp$ **then**
6: $x = lQueue.peekByLabel()$
7: $x.available = false$
8: send $(deq\_f, x, \langle localTime, i\rangle)$ to all
9: $setTimer(\epsilon, \langle deq\_f, x, \langle localTime, i\rangle\rangle, respond)$
10: **else** send $(deq\_s, null, \langle localTime, i\rangle)$ to all
11: **HandleEvent** RECEIVE (op, val, ts) FROM p_j
12: $Pending.insert(\langle op, val, ts\rangle)$
13: **if** $op = deq\_s$ **then**
14: clear $heads_j$
15: $head = lQueue.peekByLabel(p_i)$
16: $head.available = false$
17: send $(head, j)$ to all
18: $setTimer(d + u + \epsilon, \langle op, val, ts\rangle, execute)$
19: **HandleEvent** RECEIVE (val, k) FROM p_j
20: $heads_k[j] = val$
21: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, respond$)
22: **if** $op == deq\_f$ **then** Generate $Dequeue$ response with return value val
23: **else** Generate $Enqueue$ response with return value ACK
24: **HandleEvent** EXPIRETIMER($\langle op, val, ts\rangle, execute$)
25: **while** $ts \geq Pending.min()$ **do**
26: $\langle op', val', ts'\rangle = Pending.extractMin()$
27: $executeLocally(op', val', ts')$
28: $cancelTimer(\langle op', arg', ts'\rangle, execute)$

▷ continued

operation instance op_1, invoked at process p_1 returns before instance op_2 is invoked at process p_2, then op_1 is invoked more than ϵ time before op_2, so p_1's local clock when op_1 is invoked is less than p_2's when op_2 is invoked, and we have $ts(op_1) < ts(op_2)$. Thus, timestamp order respects the real-time partial order of operation instances. Since π is sorted by timestamp order, Construction 2 respects the real-time order of non-overlapping operation instances. □

Algorithm 4 Continuation of algorithm implementing a queue with Restricted Out-of-Order k-relaxed $Dequeue$.

```
29: function executeLocally(op, val, ⟨∗, j⟩)
30:     if op == enq then
31:         lQueue.enq(val)
32:         if clean == true and lQueue.size() ≤ k then
33:             let a = (lQueue.size() − 1) mod n
34:             lQueue.label(p_a, val)
35:     else
36:         clean = false
37:         if op == deq_f then
38:             lQueue.remove(val)
39:         else if op == deq_s then
40:             if lQueue.peekBySet(heads_j) ≠ ⊥ then
41:                 ret = lQueue.deqBySet(heads_j)
42:             else if lQueue.peekByLabel(p_j) ≠ ⊥ then
43:                 ret = lQueue.deqByLabel(p_j)
44:             else
45:                 ret = lQueue.deqByLabel(null)
46:             if ∀x ∈ heads_j, lQueue.contains(x) == true then
47:                 labelElements()
48:             if ∄deq_s ∈ Pending and ∃heads_j[i] for some 0 ≤ j < n then
49:                 heads_j[i].available = true
50:             if j == i then Generate Dequeue response with return value ret
51:         if lQueue.size() == 0 then clean = true
52: function labelElements
53:     while lQueue.unlabeledSize() > 0 and
          ∃j ∈ [0, n − 1] s.t. lQueue.sizeByLabel(p_j) < min{l, ⌈lQueue.size()/n⌉} do
54:         let m = min_j{lQueue.sizeByLabel(p_j)}
55:         lQueue.label(p_m, lQueue.peekByLabel(null))
```

Theorem 6. *Each process running Algorithm 3-4 locally executes all operation instances in timestamp order.*

Proof. On invocation, each operation instance op_1 sends a message to all processes (lines 2, 8, 10). By our assumptions on message delay, each process receives this message between $d-u$ and d time later. Each receiving process (including the sender) then adds the operation to the local $Pending$ priority queue and sets a timer for $d+u+\epsilon$. When that timer expires, the process will execute op_1.

Thus, each operation instance is executed at every process no later than $2d + u + \epsilon$ real time after it is invoked. An instance can also be executed before that if the $execute$ timer for another instance, op_2, with larger timestamp expires sooner. But the $execute$ timer for op_2 cannot go off less than $2d + \epsilon$ time after op_2 was invoked, which happens if its message delay was a minimal $d - u$. By that time, since the maximum message delay is d, $ts(op_1) < ts(op_2)$, and we know that op_1 could have been invoked no more than ϵ later in real time than op_2, the executing process has op_1 and all instances with smaller timestamps in its $Pending$ queue, and will execute them, before it executes op_2.

Thus, every process locally executes all operation instances in timestamp order. □

We continue by showing that the algorithm's local execution of each operation instances is correct. That is, we must show that the return value makes the sequence of locally executed instances legal, and that local variables, $lQueue, clean, Pending,$ and the $heads_j$ arrays, are left in a state such that the next operation instance will also return correctly. Since each process executes instances in the order they appear in π, this will also show that π is a legal sequence of operation instances.

Lemma 10. *After locally executing any prefix π' of π, in any complete, admissible run E of Algorithm 3-4 where π is the permutation of operation instances in E given by Construction 2, every process has the same local view of the shared object, as specified by the local variables $lQueue, clean, Pending,$ and each $heads_j$.*

This lemma follows directly from the fact that every process executes operation instances in the same order and the determinism of the algorithm. We now need only show that the local execution of each instance returns a correct value and maintains desired invariants on the local variables so that future instances will also execute correctly.

Theorem 7. *In any complete, admissible run of Algorithm 3-4, the permutation π given by Construction 2 is legal by the specification of a queue with a Restricted Out-of-Order k-relaxed $Dequeue$.*

Proof. As before, we construct an inductive argument on local execution of instances in π. To do this, we again need a labeling invariant, that after any process has locally executed some prefix ρ of π, for every element x which has a non-$\perp$ label, $\rho \cdot Dequeue(-, x)$ is legal. We assume that a prefix ρ of π is legal and satisfies the labeling invariant, and show that the prefix $\rho \cdot op$ of π is also legal and satisfies the invariant. The base case is the empty sequence, where both claims hold trivially. Consider the possible operations of which op may be an instance:

$Enqueue$: $Enqueue$ does not have a return value, so $\rho \cdot op$ is legal.

When an $Enqueue$ is locally executed it places a new element at the tail of $lQueue$. If $clean$ is true (i.e. there have been no $Dequeue$s executed since $lQueue$ was last empty) and there are no more than k elements in the new state of $lQueue$, then the $Enqueue$ labels the element just enqueued in round-robin fashion. Since this labeling only occurs as long as no elements are removed from $lQueue$, this will result in a maximum of k elements labeled, $\lfloor k/n \rfloor$ elements labeled for each process (or $\lceil k/n \rceil$ at some processes, if k is not a multiple of n). Thus, by the specification of a Restricted Out-of-Order k-relaxed queue, if the next instance is a $Dequeue$ which return a labeled value, the sequence will be legal, and the invariant holds.

Otherwise, if $clean \neq true$, then $Enqueue$ will not change the labeling. Since an $Enqueue$ cannot make it illegal to return any element in the queue which was previously legal and we assumed the labeling invariant held after local execution of ρ, the invariant holds after local execution of $\rho \cdot op$.

$Dequeue$: We must show that op satisfies *(C2)-(C4)* in Definition 8. Say that op's invoking process is p_i.

(C2) Suppose op returns val. There are two cases to consider, depending on whether op is a slow $Dequeue$ or a fast $Dequeue$:

If op is fast, then because it marks val as unavailable immediately, no later $Dequeue$ invoked at p_i can return val. Further, no $Dequeue$ invoked at any other process can return val, because a $Dequeue$ at p_j can only return an element labeled p_i if p_i sends it to p_j to place

in p_j's $heads_i$ array, in line 41. p_i will not send val to p_j after marking it as unavailable, because it only sends available values in $lQueue$ (line 15).

If *op* is a slow $Dequeue$, then we know that no later fast $Dequeue$ will return val, since p_i no longer has val in its $lQueue$ and a fast $Dequeue$ at any p_j could only return an element labeled p_j in its $lQueue$ which had not been sent to other processes and marked unavailable. A later slow $Dequeue$ could not return val at any process, because every process executes operation instances in the same order and would have removed val from $lQueue$ when locally executing *op*, so val would not be in the executing process' $lQueue$ when locally executing the later $Dequeue$.

(C3) Every element returned by a $Dequeue$ is an element found in $lQueue$ (lines 6, 41, 43, 45). Since only $Enqueue$s add elements to $lQueue$, and they only add elements which were arguments (line 31), every element a $Dequeue$ returns was previously an argument of an $Enqueue$, and is unique by the assumption of unique arguments to $Enqueue$s.

(C4) When each process locally executes *op*, let $enq(head)$ be the first $Enqueue$ in ρ which has not had a corresponding $Dequeue$ executed. $enq(head)$ is well-defined, since every process locally executes operation instances in the same order.

If *op* finds an available element val labeled p_i when it is invoked, then it is a fast $Dequeue$ and quickly returns that element, marking it unavailable in p_i's $lQueue$. At this time, p_i has locally executed some prefix ρ' of ρ. By the labeling invariant, $\rho' \cdot op$ is legal. But once an element is labeled, it is never unlabeled, so val will still be labeled after ρ and the inductive hypothesis implies that $\rho \cdot op$ is legal. Processes do not label any elements when locally executing a fast $Dequeue$, and a $Dequeue$ does not, by the specification of a queue with a Restricted Out-of-Order k-relaxed $Dequeue$, make any element which could previously have been returned unacceptable to return, so the labeling invariant holds after $\rho \cdot op$.

If *op* does not find an element labeled p_i, then we have a slow $Dequeue$. There are three places the return value for a slow $Dequeue$ may be chosen:

Line 41 It may return an available element val labeled $p_j, j \neq i$ from $heads_i$. By the labeling invariant, returning this value yields a legal sequence of operation instances.

Line 43 By the time op executes, it is possible that another operation has labeled elements, so p_i returns an element val labeled p_i. Returning this element also yields a legal sequence, by the assumption that the execution prior to deq maintains the labeling invariant.

Line 45 In this case, op determines that either there are currently no labeled elements or that the last time a $Dequeue$ labeled elements, there were fewer than n elements in $lQueue$ and none were labeled p_i. op will then return the headmost unlabeled element in the queue. In the first case, this is the head of $lQueue$. This yields a legal sequence, since the specification always allows returning the head of the relaxed queue. In the second case, since fewer than $k > n$ elements were labeled, returning the first unlabeled element in $lQueue$ yields a legal sequence, since it was one of the first k $Enqueue$s after $enq(head)$. This is also the only time the algorithm may return $\perp$, which it will only do if it is legal, as that only happens if there have been fewer than k $Enqueue$s since $enq(head)$, because the last time elements were labeled was when a $Dequeue$ removed the previous head element.

In each case, we have that $\rho \cdot op$ is a legal sequence, by the specification of a queue with a Restricted Out-of-Order k-relaxed $Dequeue$. All that now remains for us to prove is that the labeling invariant holds after local execution of a slow $Dequeue$ by showing that when a $Dequeue$ labels elements, it labels only elements which yield a legal sequence if a $Dequeue$ returns them.

A $Dequeue$ only labels elements when every element in $heads_j$, where p_j invoked the $Dequeue$, is still in $lQueue$ (Line 46). This means that the $Dequeue$ has the headmost element (in $lQueue$) with each process' label. Since the actual head element of $lQueue$ is always labeled, if any are ($Enqueue$ adds to the bottom of $lQueue$; Lines 34 and 55), this means that the $Dequeue$ will return the head of $lQueue$. Then the headmost k elements in

the new state of $lQueue$ are legal to return, since their $Enqueue$s are the first k $Enqueue$s in π which do not have a corresponding $Dequeue$. We see that only the headmost k elements in $lQueue$ are labeled in Lines 52-55. Thus, the labeling invariant is always maintained, since only clean $Enqueue$s and slow $Dequeue$s label elements.

□

Theorem 8. *Algorithm 3-4 is a correct implementation of a queue with a Restricted Out-of-Order k-relaxed $Dequeue$.*

3.4.3.2 Restricted Out-of-Order Relaxation Performance

We show the following upper bound on the amortized cost of $Dequeue$s in Algorithm 3-4:

Theorem 9. *The amortized time complexity per $Dequeue$ in any heavily loaded, complete, admissible run of Algorithm 3-4 is no more than $\frac{2d+\epsilon}{l} + \epsilon$, where $l = \lfloor k/n \rfloor$.*

Proof. Because we consider heavily loaded runs, we know that before any $Dequeue$s are invoked, each process will have at least l data elements labeled with its id. Because there are no more than l elements labeled for each process after a $Dequeue$ labels, in some execution at least every $(l+1)$th $Dequeue$ invoked at each process must be a deq_s. The l elements labeled for each p_i, though, can each either be removed by deq_fs, or they may be returned at another process by that process' deq_s.

We use the accounting method of amortized analysis to bound the time complexity as follows: In counting the time of all $Dequeues$, we will charge each process ϵ for every element labeled for that process. We will charge a process $2d + \epsilon$ for a deq_s which is invoked at that process. In counting the total number of operations, we will count $Dequeue$s at the process whose id is the label on the returned element. It is possible for a $Dequeue$ to return an unlabeled element, and this scheme does not count that operation, but since we do count the cost, and are dividing by the operation count, this does not decrease the upper bound.

Thus, at each process, the average time is low until that process executes a deq_s. At that point, the average time since the last deq_s at that process is no more than $\frac{l\epsilon+(2d+\epsilon)}{l}$, since there

have been l elements with that process' label removed, and the cost to that process is ϵ for the first l and $2d + \epsilon$ for the deq_s. Thus, the average cost at every process, and thus the amortized cost of the algorithm, is no more than $\frac{2d+\epsilon}{l} + \epsilon$. □

We will later compare this to lower bounds on the performance of algorithms for queues with unrelaxed $Dequeues$ to show that this relaxation gives better amortized performance. Further, our bound decreases with increasing k, which shows that stronger relaxation of the data type specification allows better performance.

3.5 Lower Bounds on Amortized Time Complexity

For the final results in this chapter, we give lower bounds on the amortized time complexity of $Dequeue$ operations in queues. We show, first, that both of our algorithms give performance gains over unrelaxed queues, when we consider amortized time per operation. This verifies our intuition that a relaxed data type can allow higher performance by reducing the required frequency of synchronization between processes.

Next, we give lower bounds on the amortized cost of $Dequeue$s for algorithms implementing queues with relaxed $Dequeue$s. We show that our algorithm for Out-of-Order k-relaxed $Dequeue$ is approximately optimal (with an extra term of ϵ, the clock skew bound), for reasonable values of k. We then show a lower bound for the Restricted Out-of-Order k-relaxed $Dequeue$ which is approximately a factor of two less than the performance of our algorithm. Thus, we see that we have algorithms that are near-optimal for both of these intuitive relaxations, and implicitly for Lateness k-relaxed queues, as well, since a Restricted Out-of-Order k-relaxed queue is also Lateness k-relaxed.

Our proofs rely heavily on the indistinguishability of runs, and the fact that no element can be returned more than once. We construct runs in which any algorithm with better performance than the lower bound we wish to show must have multiple processes behave in such a way that more than one will return the same element, based on the information they have. This contradiction allows us to conclude that algorithms performing faster than the proposed lower bounds are impossible.

Throughout this section, we assume $k \geq n$, the range where our algorithms are useful.

3.5.1 Strict Queue Lower Bound

We first consider implementations of unrelaxed queues. Every $Dequeue$ must return the unique head element in the structure. The proof for the amortized cost is very similar to the proof for the worst case cost. As we forced one operation instance to wait to make sure that it was not removing the head a second time, so we can force multiple simultaneous operation instances to wait, giving a high amortized cost.

Theorem 10. *In any linearizable implementation of a (unrelaxed) queue, $Dequeue$ must take at at least $d\left(1 - \frac{1}{n}\right)$ time, amortized.*

Our amortized operation times given by the algorithms for the two relaxations were $\frac{d}{l}$ and $\frac{2d+\epsilon}{l}+\epsilon$, respectively. We can see that for $l \geq 1$ and $n > 2$, the first algorithm gives better amortized performance per operation than the lower bound for unrelaxed queues, and for $l \geq 2, n > 2$ and sufficiently small ϵ, the second algorithm also performs better. Further, as l (and thus k) increases, the algorithms' performance will continue to increase, leaving this lower bound farther and farther behind. This shows that our algorithms give a benefit over prior algorithms for unrelaxed queues, so we turn our attention to determining how close our algorithms are to optimal.

3.5.2 Out-of-Order Relaxation Lower Bound

Theorem 11. *Any algorithm implementing a queue with an Out-of-Order k-relaxed $Dequeue$ with $k < n^2$ must have an amortized time complexity for $Dequeue$ at least $\frac{d}{l}$, where $l = \lfloor \frac{k}{n} \rfloor$.*

Proof. Let A be any algorithm implementing an Out-of-Order k-relaxed queue, for $k < n^2$. Suppose that A guarantees that, in any complete, admissible run, the average cost per instance of $Dequeue$s is strictly less than $\frac{d}{l}$.

When $Dequeue$ is invoked, by the definition of the Out-of-Order k-relaxation, it may return any of the headmost k elements in the queue. This means that as soon as one operation instance is complete, there is an element that is legal for the next instance to return that was not previously

legal, assuming there are more than k elements in the queue. Thus, if a process executes r instances of $Dequeue$, without receiving any communication from any other process, all of the elements returned by those operation instances must have been in the original $k + r - 1$ headmost elements in the queue.

Define complete, admissible run E with all message delays d, and p_0 initially enqueueing $k + l$ elements sequentially, finishing by some real time t. At time t, every process begins invoking $Dequeue$s as fast as possible (as soon as the previous operation instance returns). They continue this behavior until time $t + d$.

Each process cannot know about the operations being executed at other processes until it receives a message about them. By our construction, this does not happen until time $t + d$. Thus, every process must act as if it is running alone in the interval of time from t to $t + d$. It must then complete at least $l + 1$ operation instances in the interval from time t to $t + d$, to maintain the guaranteed average performance of less than $\frac{d}{l}$. Further, since no process learns about at least the first $l + 1$ $Dequeue$s at any other, the argument in the previous paragraph says that all of these operation instances must return elements from the original $k + (l + 1) - 1 = k + l$ headmost elements in the structure.

But we now have at least $n(l + 1)$ instances of $Dequeue$ which must return elements out of a set of size $k + l$. Arithmetic shows that if $n > l$, or $n^2 > k$, then $n(l + 1) > k + l$. Then, by the pigeonhole principle, at least two of the operation instances must have returned the same value, contradicting the assumed correctness of A. Thus, we have the bound. □

This bound only holds for $l < n$, which is equivalent to $k < n^2$, but it is reasonable to think that at some point, having k significantly larger than the number of available processes ceases to be as useful in real-world systems, particularly if n is large. Another consideration is that this relaxation may not be the most useful in practice. When there are fewer than k elements in the structure, the specification allows returning $\perp$, indicating that the structure is empty, even though it may not be. Thus, there could be algorithms satisfying the specification of this relaxation which never return every element in the queue. Due to these limitations, we focus our attention next on

the Restricted Out-of-Order relaxation, which provides stronger guarantees and, as we have seen, is not asymptotically more costly to implement.

3.5.3 Restricted Out-of-Order Relaxation Lower Bound

Our last lower bound shows us that Algorithm 3-4 is less than a factor of two above the lower bound on amortized performance. Because a Restricted Out-of-Order k-relaxed $Dequeue$ satisfies the conditions of an Out-of-Order k-relaxed $Dequeue$, we could apply the previous lower bound to these operations as well. However, we next show a bound without the limitation of $k < n^2$.

Theorem 12. *Any algorithm which implements a queue with a Restricted Out-of-Order k-relaxed $Dequeue$ which guarantees an upper bound c on the amortized time complexity for $Dequeue$ at all times during any complete, admissible run must have $c > \frac{d}{l}$.*

Proof. Suppose that some algorithm A implementing a queue with a Restricted Out-of-Order k-relaxed $Dequeue$ guarantees an average time complexity of $c \leq \frac{d}{l}$ in all complete, admissible runs. We will show that there is a complete, admissible run in which A behaves illegally.

We define a series of complete, admissible runs. In all of the following runs, let all message delays be d, and assume that all runs begin with an identical sequence of $2k$ $Enqueue$s invoked at process p_0 and ending before time $t - d$, which leaves element $head$ at the head of the queue.

Run E_f: Starting at time t, all processes invoke $Dequeue$s as fast as possible as long as they will finish before time $t + d$. Because no process can know about what any other process is doing, each must complete at least l instances by time $t + d$ to meet the guaranteed average time complexity. Thus, since at least one in every k consecutive $Dequeue$ instances must return the head, some process returns $head$ to one of its $Dequeue$s.

Run E_{hole}: Let p_i be the process which returned $head$ in E_f. Have all processes $p_j, j \neq i$ behave exactly as in E_f, while p_i does nothing after time t. This run must exist because no p_j can tell the difference between this run and E_f before time $t + d$.

Run $E_{h,a}$: This run is identical to E_{hole} up to time $t+d$, but then an arbitrary process p_a invokes $Dequeue$s as fast as possible starting at time $t + d$ as long as the $Dequeue$ finishes before time

$t + 2d$. p_a must complete at least l $Dequeue$s invoked at or after time $t + d$ to meet the guaranteed average time. This means that there are at least k $Dequeue$s in the run, so one must return $head$. Because $E_{h,a}$ is identical to E_h until time $t + d$, it must be one of the new instances at p_a invoked between $t + d$ and $t + 2d$.

Run $E_{h,b}$: Defined exactly as $E_{h,a}$, but p_a does nothing after $t + d$ and an arbitrary process $p_b \neq p_a$ invokes $Dequeue$s from time $t + d$ to time $t + 2d$. Here, $head$ must be returned by a $Dequeue$ at p_b between times $t + d$ and $t + 2d$.

Run $E_{a,b}$: p_a behaves as in $E_{h,a}$, p_b behaves as in $E_{h,b}$, and all other processes behave the same as they do in both of those runs. By construction, prior to time $t + 2d$, $E_{a,b}$ and $E_{h,a}$ are identical with respect to p_a's state, steps, and knowledge. Similarly, $E_{a,b}$ and $E_{h,b}$ are identical up to $t + 2d$ with respect to p_b's state, steps, and knowledge. Thus, both p_a and p_b will return $head$ to some $Dequeue$, which is illegal, contradicting the assumed existence of algorithm A. □

3.5.4 Relaxed Stacks

The primary semantic difference between stacks and queues which affects concurrency is that $Push$ and Pop generally contend with each other, while $Enqueue$ and $Dequeue$ generally do not. Since our counterexamples do not have concurrent $Enqueue$ and $Dequeue$ instances, all three of the lower bounds in this section can be straightforwardly adapted to stacks replacing $Enqueue$ with $Push$ and $Dequeue$ with Pop, relaxing the semantics of Pop analogously to those of $Dequeue$. Thus, we achieve analogous results for relaxed stacks as we have shown for relaxed queues.

3.6 Conclusion

We have made an introductory exploration into the benefits of relaxing data types to achieve higher performance in message-passing systems. Based on the intuition that non-determinism in a data type could make a lower degree of synchronization between processes sufficient, we have shown that there is a benefit to be gained by relaxing. First, we showed that the worst-case operation time is not affected by relaxation for a general class of operations. This follows from the fact that there are still times when we must have synchronization to enforce coherent

behavior. Proceeding from there, we gave two algorithms for queues with relaxed $Dequeue$s which perform significantly better, in terms of amortized cost, than the worst-case lower bounds for strict data types, for sufficient levels of relaxation ($k \geq n$). These algorithms exploit the non-determinism in the data type specification to assign different legal elements to different users in such a way that each user will be able to run locally, and thus quickly, for a time before they must resynchronize. Even with somewhat more costly synchronizing operations, as in one of the algorithms, the amortized cost per operation instance is significantly below the worst-case cost. To formalize this, we show a lower bound on amortized time complexity of $Dequeue$ for unrelaxed queues. This bound is higher than the performance our algorithms achieve, showing that there is a strict performance gain from relaxation. We then show lower bounds on amortized time complexity of $Dequeue$ for relaxed queues. We see that, for moderate relaxation, one of our algorithms is optimal, and for any level of relaxation, the other is less than twice the lower bound. Both algorithms have performance which improves as k increases, achieving greater performance gains from greater relaxation.

4. CONSENSUS NUMBERS OF RELAXED QUEUES *

4.1 Introduction

Given relaxed data type specifications, as we have presented and implemented so far, we wish to formally analyze their computational power. In this chapter, we explore the ability of relaxed data types, exemplified by queues, to solve the asynchronous consensus problem among several processes which may crash. Solving this problem allows us to implement any other data type among those processes. Thus, the largest number of processes which can solve consensus using a given data type, called the *consensus number* of the type, is a measure of the data type's computational strength [15].

We consider the space of possible parameters for three different relaxations of queues: Out-of-Order, Lateness, and Restricted Out-of-Order. We extend the classical method of bivalency arguments [15, 17] to handle the non-determinism in relaxed data types. Using this expanded method, we prove consensus numbers directly for several base classes, and show how these imply useful bounds on the consensus numbers of other parameter values.

To generalize our results, we show how parameterization of the relaxation of the three operations on a queue gives a 3-dimensional space. In this space, we give lemmas based on those in [7] which allow us to extend bounds proved for certain points across infinite areas. This allows us to totally cover the space of possible relaxations with only a handful of results.

4.1.1 Related Work

Consensus numbers were defined by Herlihy in [15] and are the standard measure of the computational strength of a shared data type. He showed that in an asynchronous system, a consensus object among a certain number of processes can wait-free implement any other shared data type

*Parts of the material in this chapter are reprinted from

E. Talmage and J.L. Welch, "Anomalies and Similarities among consensus numbers of variously-relaxed queues," in *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings* (A.E. Abbadi and B. Garbinato, eds.), vol. 10299 of *Lecture Notes in Computer Science*, pp. 191-205, Copyright 2017 by Springer.

among those processes. Thus, if a shared object can implement consensus among n processes, it is "universal" among n processes and can implement any data type in that system.

Lo and Hadzilacos [16] showed that consensus numbers do not form a robust wait-free hierarchy, in that multiple types of low consensus number can combine to implement types of high consensus number, if non-deterministic types are allowed. It remains an open question whether this is true for any level of non-determinism, or what minimum level of non-determinism causes the hierarchy to collapse. For single-type implementations, those using objects of a single type to solve consensus, though, consensus numbers are still useful, even for non-deterministic types, such as relaxed queues. [16] also set up the mechanisms for proving upper bounds on consensus numbers of non-deterministic types, which we use here.

Shavit and Taubenfeld [7] began exploring the computational power of relaxed data types by proving consensus numbers for some relaxed queues. Specifically, they proved a selection of results for Out-of-Order relaxed queues, one of the relaxations specified in [4] and which we have considered in this dissertation. We extend their work, showing explicitly how their results extend to other possible Out-of-Order queues and proving results for Lateness and Restricted Out-of-Order relaxed queues as well.

Chen et al. [22] explored the edge-condition behavior of several shared objects, with respect to their consensus numbers. They showed that the consensus power of queues is different if a $Dequeue$ on an empty queue returns a unique $\perp$ value or breaks and can never be used again, and several other examples. While we do not explore different edge-condition behaviors in depth, we note that the results we obtain do depend on our assumptions about when a $Dequeue$ or $Peek$ can see an empty queue.

4.2 Characterizing the Space of Relaxed Queues

We will consider relaxations of augmented queues where any or all of the three operations can be relaxed. Recall from Section 2.1.1, that the relaxation parameter for each operation is taken from $\mathbb{Z}^*$, an extension of the positive integers. Thus, we can visualize the space of possible relaxed queues, for a given relaxation type, as a 3-dimensional lattice. We can thus state the following

general version of two lemmas from [7] and then reason about the space of consensus numbers of relaxed queues.

Lemma 11. *For* $t \in \{O, L, R\}$ *and* $a, b, c, a', b', c' \in \mathbb{Z}^*$ *such that* $a \leq a', b \leq b'$*, and* $c \leq c'$*,* $CN(tQueue[a, b, c]) \geq CN(tQueue[a', b', c'])$

Lemma 11 states that relaxing or increasing the relaxation of an operation, or disabling an operation, will not increase a type's consensus number. The less-relaxed version of the operation satisfies the definition of the more-relaxed version, so any consensus algorithm using the more-relaxed version will also work with the less-relaxed version of the operation. Similarly, any algorithm which does not use a particular operation will work if its underlying data type is replaced by a type which differs only in that it provides additional operations.

Lemma 11 allows us to prove consensus number bounds for a finite number of points in the relaxation space and immediately have either an upper or lower bound on the consensus strength of all points in the relaxation space. In the rest of the chapter, we will fill in the consensus numbers of all relaxations of the three types defined above. We will use standard techniques, with a few novel twists, to show the consensus numbers of a handful of specific relaxations and apply Lemma 11, as well as the next two lemmas relating the spaces of different relaxation types, to achieve results for all relaxation values.

Since we are considering different types of relaxation, we state the next lemma to show some points where the 3-dimensional spaces of consensus numbers for each relaxation type are the same. Disabled operations are no different in different types of relaxation and a relaxation parameter of 1 means that the operation is not relaxed. Finally, recall that all relaxation types are equivalent with parameter $*$, imposing no ordering constraints on the operation.

Lemma 12. *For* $a, b, c \in \{1, *, \emptyset\}$*,*

$$CN(OQueue[a, b, c]) = CN(LQueue[a, b, c]) = CN(RQueue[a, b, c])$$

Similarly, since an $RQueue[a, b, c]$ satisfies both the definition of an $OQueue[a, b, c]$ and of an

$LQueue[a, b, c]$, any algorithm using one of these relaxed queues will be correct if all of its relaxed queues are replaced with $RQueue[a, b, c]$s. We thus have the following lemma:

Lemma 13. *For $a, b, c \in \mathbb{Z}^*$,*

$$CN(RQueue[a, b, c]) \geq \max\{CN(OQueue[a, b, c]), CN(LQueue[a, b, c])\}$$

We end this section with the results for unrelaxed queues from [15]. The results stated in [15] are for $Queue[1, 1, 1]$ and $Queue[1, 1, \emptyset]$, respectively, but the algorithms apply exactly as stated to the below versions, which are more useful for determining the values of relaxed queues.

Theorem 13.

- $CN(Queue[1, \emptyset, 1]) = \infty$*, and thus,* $CN(Queue[1, b, 1]) = \infty, \forall b \in \mathbb{Z}^*$
- $CN(Queue[\emptyset, 1, \emptyset]) \geq 2$*, so* $CN(Queue[a, 1, c]) \geq 2, \forall a, c \in \mathbb{Z}^*$.

These theorems imply that any relaxation which provides a $Dequeue[1]$ operation will have consensus number at least 2 and any relaxation which provides $Enqueue[1]$ and $Peek[1]$ will have infinite consensus number. In the rest of the paper, we will show where the boundaries between infinite and finite consensus number are, and those between consensus number 1 and 2. This allows us to understand which relaxations have maximum computational power and which have no more power than a register.

4.3 Relaxations Are Not All Equivalent

Before we get into the details of exploring every possible relaxation, we draw attention to two particular interesting results. This also allows us to showcase the extended techniques we use for proving consensus numbers that are necessary for non-deterministic data types.

A large part of the motivation for determining the consensus number of relaxed queues is to ease the choice of data type to use in solving a particular problem. However, if the consensus numbers of relaxed queues were easily predictable, or always the same for every type of relaxation, it would

hardly be worth proving them all. However, we here show that different types of relaxation do, in fact, have different consensus numbers for the same relaxation parameters. This seems an intuitive result, but is not entirely obvious to verify.

We also observe that it is important to be completely familiar with the consensus numbers because they change suddenly. In this result, the choice of relaxation type determines whether the consensus number is 2 or ∞. We will shortly see that even increasing a single parameter by as little as 1 can have a similarly disastrous effect on the computational strength of a data type. This leads to the conclusion that it is imperative to fully understand the space of consensus numbers of relaxed queues.

The proof of the following theorem exemplifies the extra detail that is required for proving impossibility for non-deterministic data types. The number of cases which we must consider increases, to handle different possible choices for the non-determinism. The proof, particularly in Case 1b, also shows the extra leverage we get from non-determinism. If one branch of a non-deterministic possibility is enabled, we can argue that another is as well, and use that to show the desired result.

Theorem 14. *For* $a > 1 \in \mathbb{Z}^+$, $CN(RQueue[a,1,1]) = CN(OQueue[a,1,1]) = \infty$, *but* $CN(LQueue[a,1,1]) = 2$.

Proof. First, we show that $CN(RQueue[a,1,1]) = CN(OQueue[a,1,1]) = \infty$ by giving Algorithm 5, a consensus algorithm for any number of processes using one object of the weaker type $OQueue[a,\emptyset,1]$. By Lemma 11, this type will have at most as high a consensus number as $OQueue[a,1,1]$. Then, by Lemma 13, replacing the $OQueue[a,1,1]$ with an $RQueue[a,1,1]$ will not affect the algorithm's correctness, so the result holds for both types of relaxed queue, as claimed.

Because each $Enqueue[a]$ places its argument in one of the tailmost a positions of the OQueue, after a instances of $Enqueue[a]$ have completed, the head of the OQueue is unaffected by further $Enqueue[a]$s. By executing a instances itself, each process guarantees that it calls $Peek()$ after the head element is fixed. Thus, every process' $Peek()$ returns the same value and all process decide

Algorithm 5 Consensus algorithm using one (unnamed) object of type $OQueue[a, \emptyset, 1]$; code for process p_i with private input $input$

```
1: for a iterations do
2:     Enqueue[a](input)
3: end for
4: decide(Peek())
```

the same value in a finite number of steps. Further, the returned value was some process' input, because all items put in the OQueue were processes' inputs.

Next, we note that the consensus algorithm from [15] using unrelaxed queues will also work using an $LQueue[a, 1, 1]$, since it only requires the $Dequeue$ operation. This implies half our remaining result, that $CN(LQueue[a, 1, 1]) \geq 2$. Finally, we need to show that there is no wait-free consensus algorithm for $n \geq 3$ processes using $LQueue[a, 1, 1]$ objects.

Claim 1. $CN(LQueue[a, 1, 1]) \leq 2$

Assume that some algorithm A can solve consensus among $n \geq 3$ processes using only registers and $LQueue[a, 1, 1]$s. By the lemmas in Section 2.4, we know that there is some critical configuration in some execution of A. Let C be an arbitrary such critical configuration. We then know that all enabled steps access the same shared $LQueue[a, 1, 1]$ object and must all be mutators. We will consider the possibilities for the types of enabled steps. First, note that by the argument for traditional queues in [15], any two $Dequeue[1]$ instances must lead to the same valency. Thus, at least one process must have an $Enqueue[a]$ enabled. We consider two cases: either some process has an enabled $Dequeue[1]$, or all processes have an $Enqueue[a]$ enabled.

1. Some process has a $Dequeue$ enabled. WLOG, call that process p_1, its enabled step op_1, and suppose that $C \cdot op_1$ is 1-valent. Since C is bivalent and $Dequeue$ is deterministic, there must be another process with an enabled step that leads to a 0-valent configuration. WLOG, call that step op_0 by process p_0. Because two enabled $Dequeue$ steps lead to the same valency, op_0 must be an instance of $Enqueue[a]$, $op_0 = Enqueue_0^d(x)$, i.e. op_0 places element x at distance d from the tail of the LQueue. We divide the argument into cases based on whether

the LQueue on which enabled steps operate is empty in C.

(a) The LQueue is empty in C: Then $C \cdot op_1$ and $C \cdot op_0 \cdot op_1$ both leave the LQueue empty, so these two configurations are indistinguishable to p_2. This contradicts the univalency lemma, since they have different valencies.

(b) The LQueue has size $s > 0$ in C: We begin by claiming that any 0-valent $Enqueue[a]$, e.g. op_0, must insert x at the head of the LQueue. Assume to the contrary, that $op = Enqueue^t(x)$ is enabled in C with $t < s$ and $C \cdot op$ is 0-valent. op must also be enabled in $C \cdot op_1$, as op_1 is a $Dequeue$, so the lateness of $Enqueue[a]$ is the same in $C \cdot op_1$ as in C. Then $C \cdot op \cdot op_1$ is equal to $C \cdot op_1 \cdot op$, since the $Enqueue[a]$ and $Dequeue$ change different parts of the LQueue and thus do not affect each other. This is a contradiction, since $C \cdot op \cdot op_1$ is 0-valent, $C \cdot op_1 \cdot op$ is 1-valent, and a configuration cannot be both 0-valent and 1-valent.

By this argument, we know that $op_0 = Enqueue_0^d(x)$ must insert its element to the head of the LQueue, so $d = s$. Consider the possibilities for p_2's enabled step(s) in C. p_2 must have either an $Enqueue[a]$ or a $Dequeue$ enabled. If an $Enqueue[a]$ is enabled, then an $Enqueue[a]$ to the tail (which is not the head because the LQueue is not empty) must be enabled, by the definition of an LQueue. This $Enqueue[a]$ must lead to a 1-valent state by the same argument that op_0 is to the head. If p_2 has a $Dequeue$ enabled, it must also lead to a 1-valent state, as mentioned above. Let op_2 denote whichever of these two steps is enabled for p_2 in C.

The state of the LQueue is equal in $C \cdot op_0 \cdot op_1$ and in C, since in the first configuration, op_0 inserts its element at the head, so op_1 removes it. Then $C \cdot op_0 \cdot op_1 \cdot op_2$ and $C \cdot op_2$ are indistinguishable to p_2. But this contradicts the univalency lemma, since $C \cdot op_0 \cdot op_1 \cdot op_2$ is 0-valent and $C \cdot op_2$ is 1-valent.

2. No process has a $Dequeue$ enabled. We claim that for some process p_i, the step $op_i = Enqueue_i^0(x)$ is enabled and for some $p_j \neq p_i$ and $d \geq 0$, $op_j = Enqueue_j^d(y)$ is enabled,

such that $C \cdot op_i$ and $C \cdot op_j$ have different valencies. That is, there is an $Enqueue[a]$ to the tail of the LQueue which leads to a different valency than some other $Enqueue[a]$ instance to some location (possibly the same). This must be true, because otherwise, an $Enqueue[a]$ by p_j to *any* location must lead to the same valency as op_i, an $Enqueue[a]$ by p_i to *any* location must lead to the same valency as op_j, and since those are the same valency, all enabled steps by p_i and p_j lead to the same valency. To invalidate the claim, this must hold for any pair of processes, so all enabled steps by any of the three processes must lead to the same valency. But that implies that C is univalent, contradicting our assumption that it is critical, and therefore bivalent.

WLOG, assume $i = 0$ and $C \cdot op_0$ is 0-valent, while $j = 1$ and $C \cdot op_j$ is 1-valent. By the definition of an $LQueue[a, 1, 1]$, lateness will be 0 after op_0, since it enqueues to the tail of the LQueue. Thus, $op'_1 = Enqueue_1^{d+1}(y)$ is enabled in $C \cdot op_0$. Since op_0 enqueues an element at the head of the LQueue, it is always enabled. Thus, both $C \cdot op_0 \cdot op'_1$ and $C \cdot op_1 \cdot op_0$ are reachable configurations. These two configuration have the same shared state, but different valencies, so they are indistinguishable to p_2 and violate the univalency lemma.

Thus, in every possible execution from an arbitrary critical configuration C, we reach a contradiction, which implies that our assumed algorithm A cannot exist. □

4.4 Some Relaxations Lose all Power

Here, we give another example proof to demonstrate the other major technique by which we prove upper bounds on consensus numbers. The hiding technique used in this proof was introduced in [23] and is a formal and general version of a technique used to prove bounds for queues with relaxed $Peek$s in [7]. Here, we exploit the non-determinism of the relaxed data type to force certain return values at each process. If each process only sees its own actions after a critical configuration, then it must conclude that it is running alone. Since different processes' steps have different valencies, this leads to erroneous decision values, proving the impossibility result.

We also show that even a very slight relaxation, moving from a $Peek[1]$ to a $Peek[2]$, drops the consensus number of every type of relaxed queue we consider from ∞ to at most 2. This illustrates the ease with which a developer could use the wrong relaxation and lose all guarantees on computational power, unless all relaxations' consensus numbers are known.

Theorem 15. $CN(RQueue[1, \emptyset, c]) = 1, \forall 1 < c \in \mathbb{Z}^*$.

Proof. Assume there is an algorithm A which solves consensus among 2 processes using only registers and $RQueue[1, \emptyset, c]$ objects. From Section 2.4, we know that A must have a critical configuration. Let C be an arbitrary critical configuration. WLOG, we say that process p_0 has step op_0 enabled and $C \cdot op_0$ is 0-valent. Similarly, p_1 has op_1 enabled and $C \cdot op_1$ is 1-valent. Every enabled step in a critical configuration must be a mutator, so both op_0 and op_1 are instances of $Enqueue$ (since $Dequeue$ is disabled). Say that $op_0 = Enqueue(x_0)$ and $op_1 = Enqueue(x_1)$. Note that $Enqueue$ is not relaxed, so both of these instances will add an element at the tail of the RQueue.

Configurations C, $C \cdot op_0$, $C \cdot op_1$, $C \cdot op_0 \cdot op_1$, and $C \cdot op_1 \cdot op_0$ all have different shared state, so the univalency lemma does not help us. Instead, we argue that there are executions from $C \cdot op_0 \cdot op_1$ and $C \cdot op_1 \cdot op_0$ such that all instances of $Peek[c]$ return the same values in both executions, preventing the two processes from determining which execution they are in. We do this by exploiting the non-determinism of $Peek[c]$.

Before we consider the specific executions, we claim that the RQueue must be empty in C. If not, then because there is no way to remove elements from the RQueue, and it is always legal for a $Peek[c]$ to return the element at the head of the RQueue, there are executions in which every $Peek[c]$ does return the head. If the RQueue is not empty in C and every $Peek[c]$ returns the head in an execution from $C \cdot op_0$ and in an execution from $C \cdot op_1$, then neither process can tell which execution they are in, and will decide the same value in both, violating valency in one or the other.

Consider the following execution prefix from 0-valent configuration $C \cdot op_0 \cdot op_1$, which we call E_0:

1. Repeat the following steps until some process decides. As soon as any process decides, pause both processes.

2. Allow p_0 to run alone until it executes a $Peek[c]$, let that $Peek[c]$ return x_0 which is always legal because it is the head, then continue until it has a second $Peek[c]$ enabled.

3. Next, allow p_1 to similarly run alone until it has executed one $Peek[c]$, which returns x_1, and has a second enabled. The $Peek[c]$ can return x_1 because x_1 is the second element in the RQueue, $c \geq 2$, and p_0's $Peek[c]$ just returned the head, which resets lateness to 0, meaning that it is legal for a $Peek[c]$ to return the second element in the RQueue.

Consider a second execution prefix from the 1-valent configuration $C \cdot op_1 \cdot op_0$, which we call E_1:

1. Repeat the following steps until some process decides. As soon as any process decides, pause both processes.

2. Allow p_1 to run alone until it executes one $Peek[c]$, let that return x_1, and continue until it has a second $Peek[c]$ enabled. x_1 is the head element in the RQueue, so it is always legal for a $Peek[c]$ to return, and doing so resets the lateness.

3. Let p_0 then run alone until it executes a $Peek[c]$, returning x_0, and continue until it has a second $Peek[c]$ enabled. Similarly to E_0, x_0 is the second element and p_1 just reset the lateness, so x_0 is a legal return value to $Peek[c]$.

We now need to show that both processes will decide the same values in both executions.

Both processes receive the same return values to all operation instances in E_0 and E_1, so they each execute the same series of steps in both execution prefixes. If p_0 decides first in E_0, say after executing h instances of $Peek[c]$, then p_1 executes at least $h - 1$ instances of $Peek[c]$ in E_0. But since p_1 behaves the same way in both execution prefixes, it must also execute at least h $Peek[c]$ instances in E_1, since p_0 decides after at least h. Thus, even if p_1 executes exactly h $Peek[c]$s in E_1 and decides first, p_0 will only execute one $Peek[c]$ after p_1's last (which resets lateness to 0) before

deciding. p_0 can then receive the same return value (x_0) to that hth $Peek[c]$ instance extending E_1 as it did its hth $Peek[c]$ instance in E_0. Since p_0 will decide before executing another $Peek[c]$, it will decide the same value in both execution prefixes. That contradicts the fact that the two prefixes are from univalent configurations with different valencies. Similarly if p_1 decides first in E_1.

We have shown that p_0 cannot decide first in E_0, so p_1 must decide first in E_0, which implies that p_1 executes strictly fewer $Peek[c]$ instances than p_0. But we also know that p_1 cannot decide first in E_1, so the same argument shows that p_0 must execute strictly fewer $Peek[c]$ instances than p_1. This leads to a contradiction, showing that the assumed algorithm A cannot exist. □

We can then extend this result, to cover another column in the relaxation space for each type of relaxation, by the following lemma:

Lemma 14. $\forall t \in \{O, L, R\}, CN(tQueue[a, *, c]) \leq CN(tQueue[a, \emptyset, c]), \forall a, c \in \mathbb{Z}^*$

Proof. Suppose a consensus algorithm A exists for some relaxed queue $tQueue[a, *, c]$, with $t \in \{O, L, R\}$ and $a, c \in \mathbb{Z}^*$ among some number n of processes such that $CN(tQueue[a, \emptyset, c]) < n$. Then A must invoke $Dequeue[*]$ at some point in its execution, or it would also solve consensus using objects of type $tQueue[a, \emptyset, c]$, contradicting the assumption on $tQueue[a, \emptyset, c]$'s consensus number. But with a $Dequeue[*]$, every instance can return $\perp$ in each type of relaxation. Thus, from any initial configuration, there is an execution of A in which $Dequeue[*]$ is a no-op. If A can successfully solve consensus in this execution, then we can replace each instance of $Dequeue[*]$ with a constant function to generate an algorithm A' which can solve consensus using $tQueue[a, \emptyset, c]$ from the same initial state, a contradiction. □

Theorem 16. $\forall t \in \{O, L, R\}, CN(tQueue[a, *, c]) = 1, \forall a \in \mathbb{Z}^*, 1 < c \in \mathbb{Z}^*$

4.5 Filling the Space

All we have left to do is to prove upper and lower bounds on boundary cases. These are the cases where adjusting relaxation parameters changes the consensus number of the relaxed queue. Most upper bounds we need only to prove for RQueues, since by Lemma 13 an upper bound for

RQueues applies to both LQueues and OQueues. On the other hand, algorithms for either LQueues or OQueues give lower bounds for RQueues as well.

We do not present complete proofs for the impossibility results in this section, as they all closely follow the templates of the proofs given in Sections 4.3 and 4.4. We present consensus algorithms for lower bounds, but omit the proofs since they are completely standard. At the end of the section, we present Table 4.1, a graphical representation of the relaxation spaces for each relaxation type.

4.5.1 RQueue

For RQueues, we show that any relaxation of $Peek$ results in consensus number at most 2. This upper bound applies to the entire relaxation space of $RQueue[a, b, c]$s, except where $c = 1$. For that part of the space, we show that when a reaches $*$, then any further relaxation has consensus number at most 2, and if both a and b reach $*$, then the RQueue is no stronger for consensus than a register.

The result in Theorem 15 shows that when we have relaxed $Peek$s ($c > 1$), we drop from consensus number 2 to 1 when the relaxation of $Dequeue[b]$ reaches $b = *$. The following theorems, along with the result we will show in the next section for $OQueue[\emptyset, b, \emptyset]$, completely and precisely give the consensus numbers of any $RQueue[a, b, c]$ with $a, b, c \in \mathbb{Z}^*$.

Theorem 17. $CN(RQueue[1, 1, c]) \leq 2, \forall c > 1 \in \mathbb{Z}^*$

This theorem is proved with a hiding proof, similar to the proof of Theorem 15. In constructing the indistinguishable executions, we need only be careful of when the elements $Enqueue$d immediately after a critical configuration are $Dequeue$d.

The next two bounds both have proofs in the style of Theorem 14. Both bounds involve arguing that if one $Enqueue[*]$ is enabled, then $Enqueue[*]$s to other locations in the RQueue must also be enabled, and showing that a contradiction arises. To prove the second, show that $CN(RQueue[*, \emptyset, 1]) = 1$ and use Lemma 14. The third bound in Theorem 18 is implied by an algorithm for $LQueue[\emptyset, b, \emptyset]$, which we will show in the section on LQueues as Algorithm 6.

Theorem 18.

- $CN(RQueue[*, 1, 1]) \leq 2$
- $CN(RQueue[*, *, 1]) = 1$
- $CN(RQueue[\emptyset, b, \emptyset]) \geq 2, 1 < b < * \in \mathbb{Z}^*$

4.5.2 LQueue

By Lemma 13, upper bounds for RQueues also apply to LQueues, so we immediately have:

- $CN(LQueue[1, 1, c]) \leq 2, 1 < c \in \mathbb{Z}^*$
- $CN(LQueue[1, *, c]) = 1, 1 < c \in \mathbb{Z}^*$

To determine the consensus numbers of all other $LQueue[a, b, c]$s, we also need the following two bounds. The proof of the first shows that $CN(LQueue[a, \emptyset, 1]) = 1$, using the techniques of Theorem 14, then expands the result with Lemma 14. To prove the second, we simply demonstrate a consensus algorithm for 2 processes in Algorithm 6. Intuitively, we can see that the algorithm is correct since the definition of $Dequeue[b]$ on an LQueue requires that at least one in every b consecutive $Dequeue[b]$ instances returns the element at the head of the LQueue. Thus, one of the $Dequeue[b]$ instances will return the initial element, and the process which does not $Dequeue[b]$ that element will know the other process must have.

Theorem 19.

- $CN(LQueue[a, *, 1]) = 1, \forall a > 1 \in \mathbb{Z}^*$
- $CN(LQueue[\emptyset, b, \emptyset]) \geq 2, \forall b > 1 \in \mathbb{Z}^+$

4.5.3 OQueue

We have the following results from those for RQueues and the fact that an upper bound on the consensus number of an $RQueue[a, b, c]$ implies the same upper bound on $OQueue[a, b, c]$.

Algorithm 6 Consensus algorithm for process $p_i, i \in \{0, 1\}$ with input $input_i$, using one (unnamed) object of type $LQueue[\emptyset, b, \emptyset]$, initially containing $\top$, and two registers R_0, R_1

1: $R_i.write(input_i)$
2: **for** b iterations **do**
3: $\quad ret \leftarrow Dequeue[b]()$
4: $\quad$ **if** $ret == \top$ **then** $decide(input_i)$
5: **end for**
6: $decide(R_{1-i}.read())$

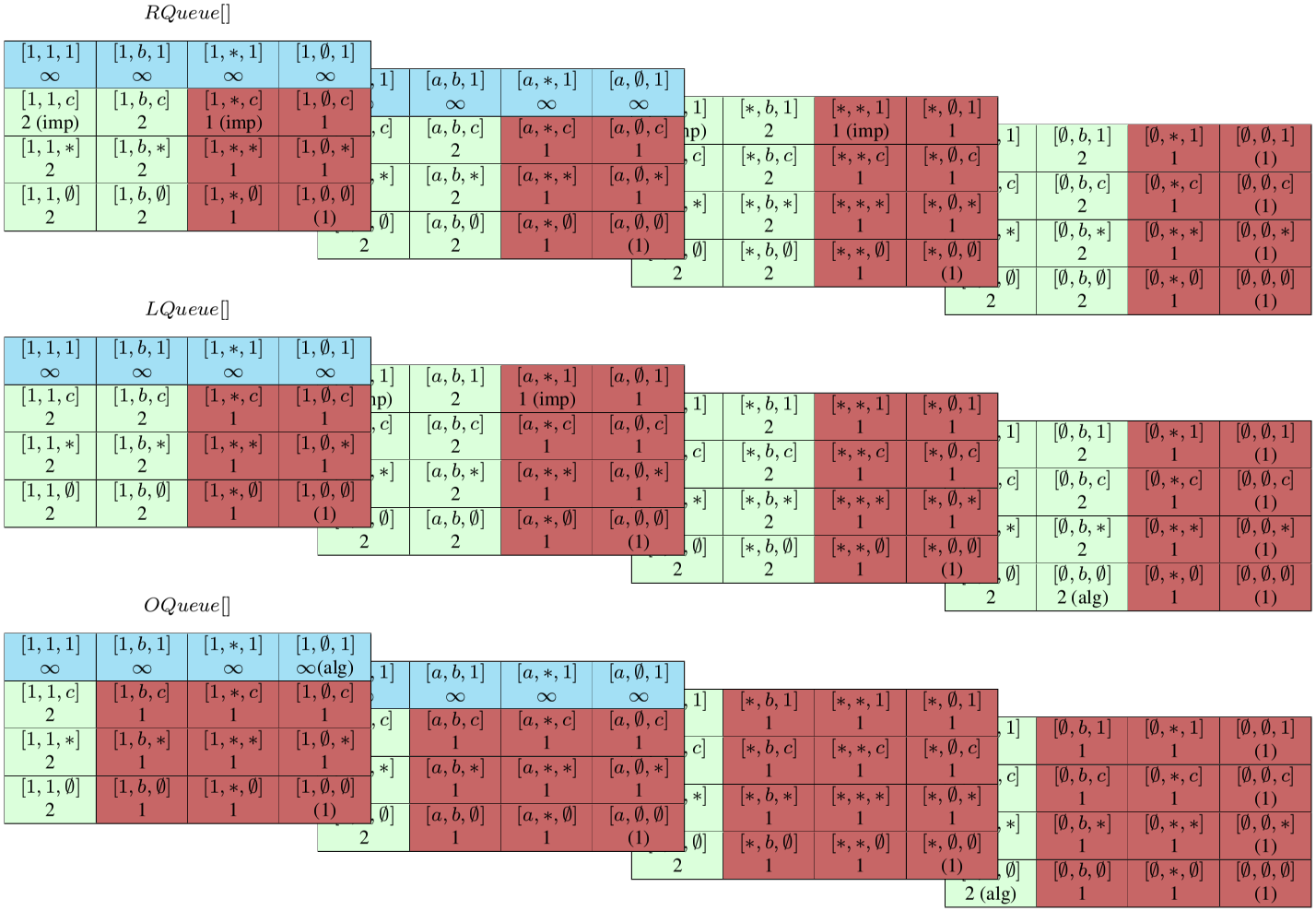
- $CN(OQueue[1, 1, c]) \leq 2, 1 < c \in \mathbb{Z}^*$
- $CN(OQueue[1, *, c]) = 1, 1 < c \in \mathbb{Z}^*$
- $CN(OQueue[*, 1, 1]) \leq 2$
- $CN(OQueue[*, *, 1]) = 1$

The following theorem determines the last of the consensus numbers of relaxed OQueues. The two bounds have very similar proofs, using the techniques of Theorem 14 applied to both $Enqueue[a]$ and $Dequeue[b]$, taking advantage of the non-determinism implying that multiple steps by a single process may be enabled in a single configuration.

Theorem 20.

- $CN(OQueue[1, b, c]) = 1, \forall 1 < b, c \in \mathbb{Z}^*$
- $CN(OQueue[*, b, 1]) = 1, \forall 1 < b \in \mathbb{Z}^*$

Table 4.1: Graphical Representation of Relaxation Space for Different Relaxation Types



4.5.4 Chart of Results

Finally, we give a graphical presentation of our results in Table 4.1. Recall that for each relaxation type, we have a 3-dimensional lattice. In the charts, we use a, b, c to indicate integers greater than 1, since it happens that within that range, consensus numbers do not change. Moving right in a grid increases the relaxation of $Dequeue$, moving down increases the relaxation of $Peek$, and moving front-to-back from one grid to the next increases the relaxation of $Enqueue$.

We mark cells with "(imp)" or "(alg)" to indicate an impossibility result or algorithm proved or restated in this chapter. Lemma 11 implies that consensus numbers must decrease while moving to the right or down within a single grid or moving back from one grid to the next. An algorithm, giving a lower bound on a consensus number, implies the same lower bound for all cells above, to the left, and in more-forward grids, since those cells have stronger and/or more operations. Cells containing "(1)" indicate vacuous data structures which do not have both an accessor and a mutator.

4.6 Conclusion

In this chapter, we have explored the space of parameterized relaxations for three related types of relaxed queues. We used a visualizable description of the three-dimensional parameter space of each relaxation to allow us to draw conclusions about every point in the space from a handful of carefully-chosen parameter choices.

Having determined the consensus number of each possible relaxation of these three types, we can draw interesting conclusions about what effect different amounts and types of relaxation have on the computational power of a data type. For instance, we note that for every relaxation type, only queues with an unrelaxed $Peek$ operation have infinite consensus number. Even the slightest relaxation of $Peek$ reduces the consensus number to 2 or less.

In fact, none of these relaxation types have consensus numbers between 2 and ∞. This means that, as far as computational guarantees are concerned, there is little purpose in using a slightly-relaxed queue. If performance is the primary concern, the degree of relaxation should be increased as much as possible, as that leads to the possibility of more efficient implementations of the data

type [5].

This work generalizes that in [7], which considers only Out-of-Order relaxed queues, which we refer to as $OQueue$s. This allows us to see the relationship between the strength of different relaxations, where the same parameters can lead to different consensus numbers, as shown in Section 4.3. We do note that we use a slightly different definition of $OQueue$ than that in [7]. They do not allow a non-empty relaxed queue with fewer than k elements to return $\perp$, indicating an empty queue. Under this definition, an $OQueue[*, *, \emptyset]$ is simply a multiset, allowing them to use the known fact that multiset's consensus number is 2.

Unfortunately, this does not match the definitions in the literature ([4, 5]), so the conclusions about increased performance from those papers do not hold. Intuitively, that definition restricts the relaxation of an almost-empty queue, making it behave as if it had smaller relaxation parameters. For this reason, we use the previous definitions, which do allow erroneous empty indicators, which leads to consensus number 1 for certain relaxations, such as $OQueue[*, *, \emptyset]$, where [7] had consensus number 2.

5. RELAXED DATA TYPES AS CONSISTENCY CONDITIONS *

5.1 Introduction and Related Work

It is important to provide the best possible guarantees on the behavior of data types under concurrent access to shared data while maintaining the efficiency of those interfaces. The study of *consistency conditions* considers what guarantees may be provided or required on the behavior of shared data objects under concurrent access. The strongest consistency condition, linearizability [13], requires that all operations on shared data appear to all processes as if they happened sequentially, in an order respecting the order of operations which do not overlap in real time. This makes program design and reasoning about program correctness relatively easy, as we are familiar with sequential program design and analysis. However, linearizability is generally expensive to implement, in terms of computation and communication delays [21, 20, 19, 5]. To avoid this cost, many weaker consistency conditions have been proposed (see [12] for a review of consistency conditions in the literature), allowing more concurrent executions while providing weaker guarantees on the behavior of shared objects. These can be implemented more efficiently than linearizability (e.g. [20]). Some work has been done to explore classes of data types which, when implemented under a weak consistency condition, give stronger behavioral guarantees that those of the consistency condition, e.g. [24].

In this chapter, we explore the relation of the weakened consistency condition and relaxation methods for improving the performance of shared data type implementations. We show that the combination of linearizability and some data type relaxations previously considered in the literature, namely k-Out-of-Order, k-Lateness, and k-Stuttering [4], can be alternately defined as consistency conditions. That is, the set of concurrent executions which are considered legal under

*Parts of the material in this chapter are reprinted from

E. Talmage and J.L. Welch, "Relaxed data types as consistency conditions," in *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings* (P.G. Spirakis and P. Tsigas, eds.), vol. 10616 of *Lecture Notes in Computer Science*, pp. 142-156, Copyright 2017 by Springer.

linearizability when working with the relaxed type is the same as the set of concurrent executions which are legal under the new consistency condition and the original, unrelaxed type. Conversely, we show, by the example of k-Atomicity, that some consistency conditions can be separated into linearizability and a data type relaxation.

This partial equivalence means that for several common relaxations and consistency conditions, the relaxation and consistency condition definitions are interchangeable. As an example of the use of this interchangeability, we use ideas from the large body of work comparing the strengths of different consistency conditions [25, 26, 12, 27] to show that the consistency conditions equivalent to k-Out-of-Order, k-Lateness, and k-Stuttering are distinct from similar previously known consistency conditions. This means that the relaxations are distinct points in the space of consistency conditions. For some particular data types, though, we show that k-Stuttering is a strengthening of k-Atomicity.

5.2 Converting Relaxations to Consistency Conditions

Relaxing data types and weakening consistency conditions have so far been largely separate methods of improving the performance of shared data types. In the next two sections, we show by example that some relaxed data types under linearizability can be equivalently defined as their base types under weaker consistency conditions and vice versa.

The basic idea is to think of both consistency conditions and relaxations as functions. Consistency conditions reduce concurrent schedules to one or more sequences of operation instances, which can be compared to the legal sequences of a given data type. We can view this as a function from the space of possible concurrent schedules to the power set of possible operation instance sequences. Data type relaxations take a sequence of operation instances and transform it to be legal by the base type's specification. This is a function from the space of possible operation instance sequences to itself.

Since the domain of relaxations is elements of the codomain of consistency conditions, we can compose the two "functions". Thus, the consistency condition can map a concurrent schedule to sequences that may not be legal by the base type, but then we may transform them by the rules of

a relaxation to be legal. Thus, both collapsing concurrency and allowing some variance from the base set of legal sequences can occur in the consistency condition.

Similarly, if a consistency condition requires a global ordering respecting the schedule order, then adds other conditions, we will show in Section 5.3 that we can split these conditions apart to have linearizability for the consistency condition and a relaxation of the original data type, while still allowing the same set of concurrent schedules.

We will now define several consistency conditions which are equivalent to the data type relaxations introduced in Section 2.1.2. The equivalence theorems follow from the fact that the set of linearizable schedules legal for the relaxed version of a data type is the equal to the set of schedules legal for the original data type and the weaker consistency condition.

First, we discuss the Out-of-Order relaxation. This enables operations to return values which are not legal by the specification of the base type T, but would be legal if a few other instances had not occurred. This larger set of possible return values can be accommodated in a consistency condition by including schedules where instances are not required to be legal by the definition of T, but are allowed some leeway.

Definition 19 (OutofOrderCC(k))**.** *A schedule of any ADT T satisfies* OutofOrderCC(k)*, for $k \geq 0$, if*

- *There exists a permutation Π of all operation instances in the schedule, which respects the schedule order of non-overlapping instances, and*
- *For every $op \in \Pi$, with $\Pi = \pi \cdot op \cdot \rho$, there is some sequence $u \cdot v \cdot w$, $|v| \leq k$, which is a minimal-length sequence equivalent in T to π, and there exists a sequence x, such that*
  - *a) $u \cdot w$ is legal in T and minimum-length among the set of sequences equivalent to it in T,*
  - *b) $u \cdot w \cdot op$ is legal in T, and*
  - *c) either*

– $u \cdot w \cdot op \equiv x \cdot w$ *and* $\pi \cdot op \equiv x \cdot v \cdot w$*, or*

– $u \cdot w \cdot op \equiv u \cdot x$ *and* $\pi \cdot op \equiv u \cdot v \cdot x$*.*

Theorem 21. *For $k \geq 0$, the set of schedules legal on a k-Out-of-Order relaxation of any ADT T under linearizability is the same as the set of schedules legal on T under OutofOrderCC(k).*

We can similarly define consistency conditions equivalent to k-Lateness or k-Stuttering relaxed versions of a type T under linearizability. Again, by rolling the relaxation into the consistency condition, we show by construction that the schedules legal on these relaxed data types under linearizability are just those legal on the base type under a weaker consistency condition.

Definition 20 (LatenessCC(k))**.** *A schedule of any ADT T satisfies* LatenessCC(k)*, for $k \geq 1$, if*

- *There exists a permutation Π of all operation instances in the schedule which respects the schedule order of non-overlapping instances, and*
- *For every $op \in \Pi$, with $\Pi = \pi \cdot op \cdot \rho$, there exists $l \geq 0$ such that $\pi \cdot op$ is legal by the semantics of an l-Out-of-Order relaxed T, and at least one in every k consecutive mutator instances in Π must have $l = 0$.*

Theorem 22. *For $k \geq 1$, the set of schedules legal on a k-Lateness relaxation of any ADT T under linearizability is the same as the set of schedules legal on T under LatenessCC(k).*

Definition 21 (StutteringCC(k))**.** *A schedule of any ADT T satisfies* StutteringCC(k)*, for $k \geq 1$, if*

1. *There exists a permutation $\Pi = op_1 \cdot op_2 \cdots$ of all operation instances in the schedule, respecting the schedule order of non-overlapping instances,*
2. *For every instance op_i in Π, let $\Pi = \pi_i \cdot op_i \cdot \rho_i$. op_i returns a value that such that $\pi'_i \cdot op_i$ is legal in T, where π'_i is a sequence of mutator instances such that*
   (a) $\pi'_1 = \varepsilon$
   (b) $\pi'_i \in \{\pi'_{i-1}, \pi'_{i-1} \cdot op_{i-1}\}$*, for* $i > 1$*, and*

(c) π_i' includes at least one of every k consecutive mutators in π_i

Theorem 23. *For $k \geq 1$, the set of schedules legal on a k-Stuttering relaxation of any ADT T under linearizability is the same as the set of schedules legal on T under StutteringCC(k).*

Theorems 21, 22 and 23 all hold by construction.

5.3 Consistency Condition to Relaxation

We have shown that we can convert familiar relaxations to consistency conditions. The interest in relaxed data types is largely founded on their ease of use and understanding, relative to consistency conditions. Ideally, then, any consistency condition would be representable as a relaxed data type. This does not seem to be true, at least for our current understanding of relaxed data types, as relaxed data type specifications are sequential, while consistency conditions may be inherently concurrent. Sequential specifications cannot use any notion of invoking process, while many consistency conditions explicitly refer to instances invoked by certain processes.

For example, sequential consistency requires that there exist a permutation of all operation instances that is legal, and in which all instances invoked at a particular process appear in the order in which they were invoked. Because a sequential specification does not know about multiple processes, it is not well-defined to require or guarantee that all instances invoked at a single process have some desired relation.

Despite this conclusion that the sets of relaxations and consistency conditions are not equivalent, in this section we will show that some consistency conditions can be equivalently expressed as relaxed data types. We consider a well-established consistency condition from the literature, and define a generic data type relaxation equivalent to it.

5.3.1 k-Atomicity

Aiyer et al. defined k-Atomicity in [14], however their definition only discusses registers. It was introduced in literature only concerned with registers and has not, to our knowledge, been generalized to other types. Since we are interested in arbitrary ADTs, we would like a more general definition. To do this, we generalize $Read$s to all pure accessors and $Write$s to all pure

mutators. It is not well-defined how mixed operations should behave under k-Atomicity. They should be allowed to return a value as if they were out of order, but then the mutations they cause could seemingly cause previous operation instances to be illegal. Given these issues, we will limit our definition of k-Atomicity to data types which have only pure operations.

Definition 22 (k-Atomicity)**.** *A schedule E on a data type T, which has only pure operations, is* k-atomic*, for $k \geq 0$, if there exists a permutation Π of all operation instances in E, respecting the schedule order of non-overlapping instances, such that for every accessor instance op, with $\Pi = \pi \cdot op \cdot \rho$, there exists a sequence π' obtained by removing up to k consecutive instances from the end of $\pi|_m$ such that $\pi' \cdot op$ is legal in T.*

We can now split this condition into two pieces. The first is the core of linearizability, that there is an ordering of all operation instances in the schedule that respects the schedule order. The second condition expands the set of legal sequences beyond the set of legal sequences specified by T. The consistency condition requires that the sequence of all instances is in the set defined by the second part. By moving the second part into the data type, relaxing the data type specification, we are left with linearizability for the consistency condition.

Definition 23 (k-Atomic-Equiv Relaxed ADT)**.** *Given any ADT T with no mixed operations and $k \geq 0$, a k-Atomic-Equiv relaxation of T is defined as follows:*

1. $OPS(T') = OPS(T)$
2. *$\mathcal{L}_{T'}$ is the set of sequences Π, where for each accessor instance op, with $\Pi = \pi \cdot op \cdot \rho$, there exists a sequence π' such that $\pi' \cdot op$ is legal in T, where π' is obtained by removing up to k consecutive instances from the end of $\pi|_m$.*

Theorem 24. *For $k \geq 0$, the set of schedules legal on a k-Atomic-Equiv relaxation of any ADT T with no mixed operations under linearizability is the same as the set of schedules legal on T under k-Atomicity.*

The theorem follows by definition.

Definition 23 is very similar to that of k-Out-of-Order, but they are not equivalent. Because it uses minimal equivalent sequences, a k-Out-of-Order relaxed data type cannot return a value which has been "deleted" from the data structure. For example, consider the following sequence: $Enqueue(1) \cdot Enqueue(2) \cdot Enqueue(3) \cdot Dequeue(1) \cdot Dequeue(x)$. In a 2-Out-of-Order queue, x could be either 2 or 3. On the other hand, a k-Atomic type can return historical values that have been deleted or overwritten, so if the sequence in the previous example were executed on a 2-Atomic-Equiv queue, x could also be 1.

In addition to k-Atomicity, [14] also introduces two more consistency conditions for registers, relaxing multi-writer versions of classic conditions from [28]. (See [29] for a discussion of some of the many ways to generalize regular and safe registers for multiple writers.) Both *k-regular* and *k-safe* registers distinguish between $Read$ instances which overlap with $Write$ instances and those which do not, allowing $Read$s overlapping a $Write$ to return the argument of some such concurrent $Write$, in the case of k-regularity, or any value in the domain of the register, in the case of k-safety.

It is interesting to note that k-Regularity and k-Safety, though very similar to k-Atomicity, cannot be directly converted into relaxed data types. This is because they allow operation instances to have different behaviors when they overlap with one or more mutators than when they do not overlap with any mutators. A sequential specification has no notion of concurrency, or overlapping operation instances, so cannot differentiate these two possibilities. Recent work, such as [2, 30, 31], has begun exploring the concept of tasks or objects which do not have sequential specifications. These more general definitions may be able to represent consistency conditions which sequential specifications cannot.

5.4 Placing New Consistency Conditions

We have shown that some data type relaxations can be expressed as consistency conditions. We would like to know how these conditions compare to known consistency conditions. They neither appear to be any common consistency conditions, nor do any of our new consistency conditions appear to be related to each other. In this section we prove that these intuitions are correct.

Recall that consistency conditions are just sets of legal schedules [12]. Thus, to compare the strength of different consistency conditions, we can compare the sets of schedules over all data types.

Definition 24. *Given two consistency conditions C and D, we say that C is* stronger *than D, and D is* weaker *than C, if for all data types T, every schedule legal under C and T is also legal under D and T. That is, the set of legal schedules under C, for all data types, is a subset of the set of schedules legal under D.*

If neither C is stronger than D nor D is stronger than C, we say C and D are incomparable*. If C is stronger than, but not equal to, D, we say that C is* strictly stronger *than D and D is* strictly weaker *than C.*

Our conditions are in the "version staleness-based" family of consistency conditions in [12], since these also have the requirements of linearizability. Thus, we will be comparing them to k-Atomicity, k-Regularity, and k-Safety, which are also version staleness-based. It is trivial to see that all of our conditions are weaker than Linearizability, since they start with the conditions of Linearizability, then allow some sequences that Linearizability does not.

First, we define generalized versions of k-Regularity and k-Safety, as we did for k-Atomicity. Because k-Regularity and k-Safety may behave exactly as k-Atomicity, we have the same restriction to data types without mixed operations.

Definition 25 (k-Regularity)**.** *A schedule E on a data type T with no mixed operations is* k-regular*, for $k \geq 0$, if there exists a permutation Π of all operation instances in E, respecting the schedule order of non-overlapping instances, such that for every instance op, $\Pi = \pi \cdot op \cdot \rho$,*

- *if op is a mutator or overlaps with no mutator instances, $\pi|_m \cdot op$ is legal by k-Atomicity, and*
- *if op is an accessor overlapping with at least one other mutator, there exists a sequence π' such that $\pi' \cdot op$ is legal in T, where π' is constructed either by deleting up to k instances from the end of $\pi|_m$ or by moving any subset of the mutator instances overlapping with op from after op in Π to before it and placing them in some order.*

Definition 26 (k-Safety)**.** *A schedule E on a data type T with no mixed operations is* k-safe*, for $k \geq 0$, if there exists a permutation Π of all operation instances in E, respecting the schedule order of non-overlapping instances, such that for every instance op of operation OP,*

- *if op is a mutator or overlaps with no mutator instances, $\pi|_m \cdot op$ is legal by k-Atomicity, and*
- *if op is an accessor overlapping with at least one other mutator, it may return any value in $rets(OP)$.*

First, we state the following theorem relating k-Atomicity, k-Regularity, and k-Safety. This theorem is well established in the literature for registers, and directly generalizes for our new definitions.

Theorem 25 ([14, 29, 12])**.** *For all $k \geq 0$, k-Safety is strictly weaker than k-Regularity which is strictly weaker than k-Atomicity, which is strictly weaker than Linearizability, in the domain of data types which do not have mixed operations.*

Theorem 25 claims the following two statements for each pair of consistency conditions C and D, with C claimed strictly weaker than D: First, for every data type T for which D is defined, every schedule legal under D and T is legal under C and T. Second, there is some data type S for which there is a schedule legal under C and S but not under D and S. The proof follows immediately from the definitions, since linearizable behavior is legal under k-Atomicity, k-atomic behavior is legal under k-Regularity, and k-regular behavior is legal under k-Safety.

We will next show that none of the three new consistency conditions we have defined are comparable to any of these three previously known conditions. If we can show that a consistency condition C does not contain (is not weaker than) k-Atomicity, then we immediately know that C is not weaker than either k-Regularity or k-Atomicity, because any point in k-Atomicity is also in the supersets k-Regularity and k-Safety. Conversely, if k-Safety does not contain C, then neither k-Regularity nor k-Atomicity can either, since they are subsets of k-Safety, so C is not stronger than any of the three.

Thus, by Theorem 25, to show a consistency condition C is incomparable with all of k-Atomicity, k-Regularity, and k-Safety, we choose a data type T and give a schedule which is legal under k-Atomicity and T, but not C and T, and a data type T' and give a schedule which is legal under C and T' but not under k-Safety and T. The proof of Theorem 26 uses this structure.

Theorem 26. *In the domain of data types which do not have mixed operations,*

1. *For all $k, l \geq 1$, OutofOrderCC(k) is incomparable with any of l-Safety, l-Regularity, and l-Atomicity.*

2. *For all $k \geq 2$ and $l \geq 1$, LatenessCC(k) is incomparable with any of l-Safety, l-Regularity, and l-Atomicity.*

3. *For all $k \geq 2$ and $l \geq 1$, StutteringCC(k) is incomparable with any of l-Safety, l-Regularity, and l-Atomicity.*

Proof. Throughout this proof, when we use instances of $Enqueue$ and $Peek$, we are referring to a restricted FIFO queue data type, which has no $Dequeue$, since that is a mixed operation. $Enqueue$ is a pure mutator, since it has no return value, and $Peek$ is a pure accessor, since it does not change the shared object.

1. OutofOrderCC(k):

   - To show that OutofOrderCC(k) does not contain l-Atomicity, consider the following sequential schedule of a register: $Write(1) \cdot Write(2) \cdot Read(1)$.

     For every $l \geq 1$, this schedule is legal under l-Atomicity, since the $Read$ can ignore the presence of the last preceding mutator instance, the $Write(2)$. This schedule is not legal under OutofOrderCC(k), for any $k \geq 1$, as the minimal-length equivalent sequence to $Write(1) \cdot Write(2)$ is simply $Write(2)$, and $Read(1)$ is not legal after any sequence obtained by deleting instances from this one-element sequence.

- To show that OutofOrderCC(k) is not contained in l-Safety, consider the following sequential schedule of a restricted FIFO queue: $Enqueue(1) \cdot Enqueue(2) \cdot Peek(2)$.

  This schedule is legal under OutofOrderCC(k), for every $k \geq 1$, since the prefix sequence $Enqueue(1) \cdot Enqueue(2)$ is a minimal-length sequence equivalent to itself, and then $Peek(2)$ is legal after the sequence $Enqueue(2)$ obtained by removing one mutator instance. This schedule is not legal under l-Safety, for any $l \geq 1$, since none of the instances are concurrent, and $Peek(2)$ is not legal after any sequence obtained by deleting consecutive mutators from the end of the preceding sequence. Thus, OutofOrderCC(k) is not a subset of l-Safety and is thus not stronger than any of l-Safety, l-Regularity, and l-Atomicity.

2. LatenessCC(k):

   - To show that LatenessCC(k) does not contain l-Atomicity, consider the following sequential schedule of a register: $Write(1) \cdot Write(2) \cdot Read(1)$.

     This schedule is legal under l-Atomicity, for $l \geq 1$, since the first two instances are legal in an unrelaxed register, and the $Read(1)$ is legal after the sequence obtained by ignoring the last previous mutator. This schedule is not legal under LatenessCC(k), $k \geq 2$, since the minimal-length equivalent sequence of $Write(1) \cdot Write(2)$ is $Write(2)$, so $Read(1)$ is not legal after any sequence obtained by deleting instances from a minimal sequence equivalent to the sequence of preceding instances.

   - To show that l-Safety does not contain LatenessCC(k), consider the following sequential schedule of a restricted FIFO queue: $Enqueue(1) \cdot Enqueue(2) \cdot Peek(2)$.

     This schedule is legal under LatenessCC(k), for $k \geq 2$, since the prefix sequence $Enqueue(1) \cdot Enqueue(2)$ is legal in a FIFO queue and $Peek(2)$ is legal after the sequence $Enqueue(2)$ obtained by removing one instance, which is allowed because $Enqueue(1) \cdot Enqueue(2)$ is a minimal-length equivalent sequence of itself. Under l-Safety this schedule is not legal, for any $l \geq 1$, because $Peek(2)$ is not concurrent

with any mutator and not legal after any sequence obtained by deleting instances from the end of $Enqueue(1) \cdot Enqueue(2)$.

3. StutteringCC(k):

   - To show that StutteringCC(k) does not contain l-Atomicity, consider the following sequential schedule of a register: $Write(1) \cdot Write(2) \cdot Read(1) \cdot Read(2)$.

     This schedule is legal under l-Atomicity, $l \geq 1$, since the first $Read$ instance may ignore the last previous mutator, while the second $Read$ may see it. For $k \geq 2$, this schedule is not legal under StutteringCC(k), as the first $Read$ may only return 1 if $Write(2)$ stuttered, but then no succeeding $Read$ can see the $Write(2)$.

   - To show that l-Safety does not contain StutteringCC(k), consider the following sequential schedule of a restricted FIFO queue: $Enqueue(1) \cdot Enqueue(2) \cdots Enqueue(k-1) \cdot Enqueue(k) \cdots Enqueue(k+l) \cdot Peek(k)$.

     This schedule is legal under StutteringCC(k), $k \geq 2$,[1] since the first $k-1$ $Enqueue$ instances in the prefix $Enqueue(k) \cdots Enqueue(k+l)$ may stutter, leaving $Peek(k)$ legal. This schedule is not legal under l-Safety, $l \geq 1$, since the $Peek$ is not concurrent with any mutator and ignoring up to l of the last previous mutators will not allow $Peek$ to return any value besides 1.

□

While our new consistency conditions are all incomparable to these similar existing conditions in general, we observe that for some specific data types, we may actually be able to compare them. We next show that for a certain class of data types, StutteringCC(k) is stronger than k-Atomicity. This class of types are those where all mutators are *overwriters*. An overwriter OP is an operation such that every sequence $\pi \cdot op$, $op \in OP$, is equivalent to the singleton sequence op [32, 33]. This means that the set of next operation instances which result in a legal sequence is determined

[1] Recall that StutteringCC(1) is merely linearizability.

entirely by the last previous mutator. For example, $Write$ on a register is an overwriter, since future operations only depend on the value of the most recent $Write$ instance, while $Enqueue$ on a FIFO queue is not an overwriter, as later operation instances can depend on $Enqueue$ instances prior to the most recent. At first, it may appear that all data types with an overwriter are essentially a $Read/Write$ register, but with other mutators and accessors which may only change or return a portion of the total state, there can many different data types with overwriters.

StutteringCC(k) and k-Atomicity both allow ignoring some recent mutator instances. The difference, which makes the two consistency conditions distinct, is that a stuttering instance must be ignored by all subsequent operation instances, while in k-Atomicity, instances may be ignored by some subsequent instances, but seen by others.

We show the slightly stronger result that, when all mutators are overwriters, StutteringCC(k) is stronger than $(k-1)$-Atomicity. $(k-1)$-Atomicity is always stronger than k-Atomicity, since ignoring up to the last $(k-1)$ previous mutator instances is a special case of ignoring up to the last k previous mutator instances. This gives us the immediate corollary that StutteringCC(k) is stronger than k-Atomicity, for types which only have overwriting mutators.

Theorem 27. *If all mutators in a data type T, which has no mixed operations, are overwriters, then for all $k \geq 1$, StutteringCC(k) on T is stronger than $(k-1)$-Atomicity on T.*

Proof. We show that any schedule which is legal under StutteringCC(k) is also legal under $(k-1)$-Atomicity. Consider any schedule E. Let Π be an ordering of all instances in E, which respects the schedule partial order of non-overlapping instances, as specified by the definition of StutteringCC(k). Let $\Pi = op_1 \cdot op_2 \cdots$. For each π'_i specified by the definition of StutteringCC(k), let m_i be the last mutator instance in π'_i. Because all mutators are overwriters, $\pi'_i \equiv m_i$. For each $op_i \in \Pi$, there cannot be more than $(k-1)$ mutator instances in Π strictly between m_i and op_i, by the definition of π'_i and m_i. Thus, by deleting up to $(k-1)$ of the last previous mutator instances before op_i in π, m_i will be the last mutator instance, and because it is a mutator, $op_1 \cdots m_i \equiv m_i \equiv \pi'_i$, so $op_1 \cdots op_i$ is legal under k-Atomicity. Thus, Π is legal under k-Atomicity. □

Finally, we show that the three new consistency conditions corresponding to data type relaxations we introduced in this chapter are incomparable to one another. This reflects the different approaches they take to relaxation. Given that they are seemingly orthogonal to one another, an interesting future direction is combining these conditions. Combining consistency conditions, and their components, is a common approach (e.g. [26]), and a combination of the Lateness and Out-of-Order relaxations appears as a distinct relaxation in previous works [4, 5]. It would be enlightening to compare the ease of definition and analysis for combining conditions either as data type relaxations or as consistency conditions.

Observe that in the proof of Theorem 28, as in that of Theorem 26, all the schedules we use as counterexamples are sequential. At first glance, this may appear odd, since the purpose of different consistency conditions is to handle concurrency in different ways. On further thought, though, this is actually natural, now that we have shown a correspondence between several of these consistency conditions and sequential (relaxed) data types. We are, in effect, showing that the different relaxations are distinct. We did not previously have formal tools for comparing them, though we could compare their effects on other measures, such as consensus numbers [7, 8]. Even though we cannot define relaxations equivalent to k-Regularity and k-Safety, because they are generalizations of k-Atomicity, the counterexamples for k-Atomicity are sufficient for the proof of Theorem 26.

We no longer restrict the set of data types considered, since these relaxations are defined for all data types.

Theorem 28. *Considered on all data types and for all $k \geq 1$ and $l, m \geq 2$, OutofOrderCC(k), LatenessCC(l), and StutteringCC(m) are all incomparable to one another.*

Proof.

- First, we compare OutofOrderCC(k) and LatenessCC(l), showing that neither condition contains the other.
  - The sequential schedule $Enqueue(1) \cdots Enqueue(l{+}2) \cdot Dequeue(2) \cdots Dequeue(l{+}2)$ is legal on a FIFO queue under OutofOrderCC(k), because the prefix sequence

$Enqueue(1) \cdots Enqueue(l+2)$ is legal in the base type, and for each of the following instances $Dequeue(x)$, we can obtain a π' such that $\pi' \cdot Dequeue(x)$ is legal by deleting $Enqueue(1)$ from the minimal-length equivalent sequence to the preceding sequence, which consists of all $Enqueue$ instances whose argument has not yet been returned. The schedule is not legal under LatenessCC(l) because there are l consecutive $Dequeue(x)$ instances, for none of which is $\pi' \cdot Dequeue(x)$ legal in a FIFO queue, when π' is a minimal-length equivalent sequence to the prior history, since all minimum-length sequences equivalent to some prefix of this schedule start with $Enqueue(1)$.

- The sequential schedule $Enqueue(1) \cdots Enqueue(k+2) \cdot Dequeue(k+2)$ is legal on a FIFO queue under LatenessCC(l), but not under OutofOrderCC(k). The prefix $Enqueue(1) \cdots Enqueue(k+2)$ is legal in the base type and of minimum length among equivalent sequences. By deleting a finite number, $k+1$, of consecutive mutators from the preceding sequence, we have $Enqueue(k+2) \cdot Dequeue(k+2)$ which is legal in the base type. Under OutofOrderCC(k), the schedule is not legal, because deleting up to k consecutive mutators from the prefix before the $Dequeue$ instance yields a sequence ending in $Enqueue(x) \cdot Enqueue(k+2)$, where $1 \leq x \leq k+1$, and appending $Dequeue(k+2)$ to such a sequence cannot give a sequence legal in a FIFO queue.

- Next, we compare OutofOrderCC(k) and StutteringCC(m):

  - The sequential schedule $Enqueue(1) \cdot Enqueue(2) \cdot Dequeue(2) \cdot Dequeue(1)$ on a FIFO queue is legal under OutofOrderCC(k), since removing $Enqueue(1)$, which is already minimal-length, from the preceding sequence gives $Enqueue(2) \cdot Dequeue(2)$, which is legal in a FIFO queue. The prefix of the first three instances is then equivalent to $Enqueue(1)$, and $Enqueue(1) \cdot Dequeue(1)$ is legal, so the entire sequence is legal. This schedule is not legal under StutteringCC(m), because π' for $Dequeue(2)$ must not

include $Enqueue(1)$, so no later π' may include $Enqueue(1)$ and 1 cannot be returned by a $Dequeue$. In other words, $Enqueue(1)$ stutters, having no effect, so the second $Dequeue$ instance cannot return 1.

- The sequential schedule $Enqueue(1) \cdot Dequeue(1) \cdot Dequeue(1)$ on a FIFO queue is legal under StutteringCC(m), but not under OutofOrderCC(k). In StutteringCC(m), π' for the first $Dequeue$ instance is $Enqueue(1)$, leaving out no previous instances. This $Dequeue$ instance stutters, having no effect, so π' for the second $Dequeue$ is $Enqueue(1)$, and thus $\pi' \cdot Dequeue(1)$ is legal.

  For OutofOrderCC(k), the minimal-length equivalent sequence of the schedule's prefix $Enqueue(1) \cdot Dequeue(1)$ is the empty sequence ε, and since $\varepsilon \cdot Dequeue(1)$ is not legal in a FIFO queue, the original schedule is not legal.

- Finally, compare LatenessCC(l) and StutteringCC(m):

  - The sequential schedule $Enqueue(1) \cdots Enqueue(m+2) \cdot Dequeue(m+2)$ on a FIFO queue is legal under LatenessCC(l) but not under StutteringCC(m), as above. For StutteringCC(m), the π' for the $Dequeue$ instance must contain at least one of every m consecutive mutator instances in the preceding sequence. This means that it must contain at least 1 $Enqueue(x)$, where $x < m+2$, since there are $m+1$ consecutive such instances. Thus, $\pi' \cdot Dequeue(m+2)$ is not legal, so this schedule is not legal under StutteringCC(m).

  - The sequential schedule $Enqueue(1) \cdot Dequeue(1) \cdot Dequeue(1)$ on a FIFO queue is legal under StutteringCC(m), as argued above, but is not legal under LatenessCC(l). The second $Dequeue$ instance must be legal after deleting a minimal-length sequence equivalent to the prefix $Enqueue(1) \cdot Dequeue(1)$. That is the empty sequence, though, and $\varepsilon \cdot Dequeue(1)$ is not be legal.

In each case, we have shown that the sets of schedules legal under each pair of consistency conditions are not related by subset or superset, so the three consistency conditions are pairwise

incomparable. □

5.5 Conclusion

In exploring the relation between relaxations for abstract data types and consistency conditions, we have shown that in several cases, the ideas in each may be expressed equivalently by the other. Specifically, we showed that the k-Out-of-Order, k-Lateness, and k-Stuttering relaxations may be equivalently expressed as consistency conditions and that the consistency condition k-Atomicity can be equivalently expressed as a relaxation. For each of these, we define the equivalent consistency condition or relaxation. We then explore how the newly-defined consistency conditions fit into the space of consistency conditions, related by the conditions' strength, by showing that they are distinct from several previously-known similar conditions.

In this work, we did not consider relaxing particular operations in a data type. It is possible, and common in the literature [4, 5], to relax the behavior of certain operations, while requiring that others behave as in the base type. In the case of per-operation relaxations, our result in Section 5.4 regarding data types where all mutators are overwriters would extend to all data types where all overwriting operations were k-Stuttering relaxed, greatly increasing their scope.

In the future, we need to define or quantify the space of possible data type relaxations, and maybe that of consistency conditions. This would allow more general conclusions about the relation of the two fields. For example, it seems that every data type relaxation can be expressed as a consistency condition, while only some consistency conditions can be expressed as relaxations. If we could formally show this, the space of relaxations would be a subset of the space of consistency conditions. One possible approach is to use the technique of combining consistency conditions to obtain stronger conditions to find new data type relaxations.

6. GENERIC PROOFS OF CONSENSUS NUMBERS FOR ABSTRACT DATA TYPES *

6.1 Introduction

Determining the power of shared data types to implement other shared data types in an asynchronous crash-prone system is a fundamental question in distributed computing. Pioneering work by Herlihy [15] focused on implementations that are both wait-free, meaning any number of processes can crash, and linearizable (or atomic). As shown in [15], this question is equivalent to determining the *consensus number* of the data types, which is the maximum number of processes for which linearizable shared objects of a data type can be used to solve the consensus problem. If a data type has consensus number n, then in a system with n processes, shared objects of this type can be used to implement shared objects of any other type. Thus, knowing the consensus number of a data type gives us a good idea of its computational strength.

We wish to provide tools with which it is easy to determine the consensus number of any given data type. So far, most known consensus number results are for specific data types. These are useful, since we know the upper and lower bounds on the strength of many commonly-used objects, but are of little or no help in determining the consensus number of a new shared data type. Further, even among the known bounds, there are some that seem similar, and even have nearly identical proofs of their bounds, but these piecemeal proofs for each data type give no insight into those relations.

6.1.1 Summary of Results

We define a general schema for classifying data types, based on their sequential specifications, which we call *sensitivity*. If the information about the shared state which an operation returns can be analyzed to extract the arguments to a particular subsequence of past operation instances, we

*Parts of the material in this chapter are reprinted from

E. Talmage and J.L. Welch, "Generic proofs of consensus numbers for abstract data types," in *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France* (E. Anceaume, C. Cachin, and M. G. Potop-Butucaru, eds.), vol. 46 of *LIPIcs*, pp. 32:1-32:16, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

say that the data type is sensitive to that subsequence. For example, a register is sensitive to the most recent write, since a read returns the argument to that write. A stack is sensitive to the last $Push$ which does not have a matching Pop, since a Pop will return the argument to that $Push$. We define several such classes in this chapter, such as data types sensitive to the kth change to the state, data types sensitive to the kth most recent change, and data types sensitive to the l consecutive most recent changes.

We show a number of bounds, both upper and lower, on the number of processes which can use shared objects whose data types are in these different sensitivity classes to solve wait-free consensus. Specifically, we begin by showing that information about the beginning of a history of operation instances of a shared data type allows processes to solve consensus for any number of processes. This is a natural result, since the ordering of operation instances on the shared objects allows the algorithm to break symmetry.

An augmented queue, as in [15], using $Enqueue$ and $Peek$ is such a data type, as $Peek$s can always determine what value was enqueued first, and all processes can decide that value. Other examples include a Compare-And-Swap (CAS) object using a function which stores its argument if the object is empty and returns the contents, without changing them, if it is not. Repeated applications of this operation have the effect of storing the argument to the first operation instance executed and returning it to all subsequent instances. There are data types which are stronger than this class which can learn the first event, such as types with operations which return the entire history of operation instances on the shared object, but our result shows that that strength is unneeded for consensus.

Next, we consider what happens if a data type has operations which depend on the last operation instances executed. We show that if a data type has only operations whose return values depend exclusively on one instance at a fixed distance back in history, then that data type can only solve consensus for a small, constant number of processes. If none of such a data type's operations can atomically both read and change the shared state, then the type has consensus number 1. If a data type's operations reveal some number l of consecutive changes to the shared state, then it can solve

consensus for l processes.

These data types model the scenario when there is limited memory. If we want to store a queue, but only have enough memory to store k elements, we can throw away older elements, yielding a data type sensitive to recent operations. A cyclical queue has such behavior, and with operations $Enqueue$ and $Peek$, where $Peek$ returns the kth-most recent argument to $Enqueue$, has consensus number 1. To solve consensus for more processes with a similar data type, we show that knowledge of consecutive past operations is sufficient. If instead of only one recent argument, we can discern a contiguous sequence of them, we can solve consensus for more processes. Using the same cyclical k-queue, if our $Peek$ operation is replaced with a $ReadAll$ which tells the entire contents of the queue atomically, we show that we can solve consensus for k processes. This parameterized result suggests a fundamental property of the amount of necessary information for solving consensus.

6.1.2 Related Work

Herlihy[15] first introduced the concepts of consensus numbers and the universality of consensus in asynchronous, wait-free systems. He showed that a consensus object could provide a wait-free and linearizable implementation of any other shared object. Further, he showed that different objects could only solve consensus for certain numbers of processes. This gives a hierarchy of object types, sorted by the maximum number of processes for which they can solve consensus. He also proved consensus numbers for a number of common objects.

Many researchers have worked to understand exactly what level of computational power consensus numbers represent, and when they make sense as a measure of computational power. Jayanti and Toueg [34] and Borowsky, et al. [35] established that consensus numbers of specific data types make sense when multiple objects of the type and R/W registers are used, regardless of the objects' initial states. Bazzi et al. [36] showed that adding registers to a deterministic data type with consensus number greater than 1 does not increase the data type's consensus number. Other work establishes that non-determinism can collapse the consensus number hierarchy [16, 37], that consensus is impossible with Byzantine faults [38], and what happens when multiple shared objects can be accessed atomically [39].

Ruppert [40] provides conditions with which it is possible to determine whether a data type can solve consensus. He considers two generic classes of data types, RMW types and readable types. RMW types have a generic Read-Modify-Write operation which reads the shared state and changes it according to an input function. Readable types have operations which return at least part of the state of the shared object without changing it. He shows that for both of these classes, consensus can be solved among n processes if and only if they can discern which of two groups the first process to act belonged to. This condition, called n-discerning, is defined in terms of each of the classes of data types. This has a similar flavor to our first result below, where seeing what happened first is useful for consensus. We define our conditions more directly as properties of the sequential specification of a shared object and also consider different perspectives on what previous events are visible.

Chordia et al. [41] have lower bounds on the number of processes which can solve consensus using classes of objects with definitions similar to [40]–the duration for which two operation orderings are distinguishable affects the objects' consensus power–using algebraic properties, as we do. These results are not directly comparable to those in [40], since they have different assumptions about the algorithms and exact data returned. [41] also does not provide upper bounds, on which we focus.

In another direction, Chen et al. [22] consider the edge cases of several data types, when operations' return values are not traditionally well-defined. An intuitive example is the effect of a $Dequeue$ operation on an empty queue, where it could return $\perp$ or return an arbitrary value, never return a useful value again, or a number of other possibilities. They consider a few different possible failure modes, and show that the consensus numbers of objects are different when they have different behaviors when the object "breaks" in such a case. These results are orthogonal to ours, as they primarily focus on queues and stacks, and assume that objects break in some permanent way when they hit such an edge case. We assume that there is a legal return value for any operation invocation, and that objects will continue to operate even after they hit such an edge case.

6.2 Sensitivity

We introduce the concept of *sensitivity* to classify operations. The sensitivity of a set of operations is a means of tracking which previous operations on a shared object cause a particular instance to return a specific value. Intuitively, an operation which has a return value will usually return a value dependent on some subset of previous operation instances. For example, a $Read$ on a register will return the argument to the last previous $Write$. On a queue, an instance of $Dequeue$ will return the argument of the first $Enqueue$ instance which has not already been returned by a $Dequeue$. We categorize operations by which previous instances (first, latest, first not already used, etc.) we can deduce, or "see", based on the return value of an instance of an accessor operation.

Definition 27. *Let OPS be a subset of the operations of a data type T. Let OPS_M denote the set of all mutators in OPS. Let S be an arbitrary function that, given a finite sequence $\rho \in \mathcal{L}_T$, returns a subsequence of ρ consisting only of instances of mutators.*

OPS is defined to be S-sensitive *iff there exist an accessor $AOP \in OPS$ and a computable function $decode : rets(AOP) \to \big($ the set of finite sequences over $\bigcup_{MOP \in OPS_M} args(MOP)\big)$ such that for all $\rho \in \mathcal{L}_T$, $arg \in args(AOP)$, and $ret \in rets(AOP)$ with $\rho \cdot AOP(arg, ret) \in \mathcal{L}_T$, $decode(ret) = S(\rho)|_{args}$.*

Definition 28. *A subset OPS of the operations of a data type T is* strictly S-sensitive *if for every $\rho \in \mathcal{L}_T$, every accessor AOP and every instance $AOP(arg, ret)$ with $\rho \cdot AOP(arg, ret) \in \mathcal{L}_T$, $ret = S(\rho)|_{args}$. That is, $AOP(arg, ret)$ gives no knowledge about the shared state except for $S(\rho)|_{args}$.*

An example, for which we will later show bounds on the consensus number, is *k-front-sensitive* sets of operations:

Definition 29. *A subset OPS of the operations of a data type T is* k-front-sensitive *for a fixed integer k if OPS is S-sensitive where $S(\rho)$ is the kth mutator instance in ρ for every $\rho \in \mathcal{L}_T$ containing only instances of operations in OPS which has at least k mutator instances.*

In an augmented queue (as in [15]), the operation set $\{Enqueue, Peek\}$ is k-front-sensitive by this definition, where $k = 1$, S returns the first mutator in a sequence of operation instances, the accessor AOP is $Peek$, and the $decode$ function is the identity, since the return value of $Peek$ is the argument to the first $Enqueue$ on the queue. In fact, this operation set is also strictly 1-front-sensitive, since the return value of an instance of $Peek$ is the argument to the single first $Enqueue$.

Note that we do not require that the set of sensitive operations is the entire set of operations supported by the shared object(s) in the system. There may be other operations. These extra operations do not detract from the ability of a sensitive set of operations to solve consensus, since an algorithm may just choose not to use any other operations. This means that our proofs of the ability to solve consensus are powerful. Impossibility proofs do not get this extra strength, as a clever combination of operations which are not individually sensitive in a particular way may allow stronger algorithms.

6.3 k-Front-Sensitive Data Types

We begin by proving a result that generalizes the consensus number of augmented queues. We observe that if all processes can determine which among them was the first to modify a shared object, then they can solve consensus by all deciding that first process' input. For, example, in an augmented queue, any number of processes can solve consensus by each enqueuing their input value, then using $Peek$ to determine which $Enqueue$ instance was first [15].

More generally, processes do not need to know which mutator instance was first, as long as they can all determine, for some fixed integer k, the argument of the kth mutator instance executed on the shared object. Thus, we have the following general theorem, which applies to either a mutator and pure accessor or to a mixed operation. An example (for $k = 1$) is an augmented queue, where $Peek$ returns the first argument ever passed to an $Enqueue$, requiring no decoding. Another similar example is a Compare-And-Swap operation which places a value into a shared register in an initial state and leaves any other value it finds in the object, leaving the argument of the first operation instance still in the shared object, and thus decodable at each subsequent operation. Generally,

for any k, a mixed operation which stores a value and returns the entire history of past changes satisfies the definition, since the first argument is always visible to later operations.

Theorem 29. *The consensus number of a data type containing a k-front-sensitive subset of operations is ∞.*

We give a generic algorithm (Algorithm 7) which we can instantiate for any k-front-sensitive set of operations (which has a mutator with at least two possible distinct arguments) to solve consensus among any number of processes and prove its correctness as a consensus algorithm. The mutator and accessor in the algorithm are not necessarily distinct operations.

Algorithm 7 Consensus algorithm for a data type with a k-front-sensitive subset of operations OPS, using a mutator OP and accessor AOP, each in OPS

```
1: for i = 1 to k do
2:     OP(input)
3: end for
4: result ← AOP(arg)                    ▷ Arbitrary argument arg
5: val ← decode(result)
6: decide(val)
```

Proof. We must show that this algorithm satisfies the three properties of a consensus algorithm.

- *Termination*: Each process performs a finite number of operation instances, never waiting for another process. Thus, even in a wait-free system, where any number of other processes may have crashed, all running processes will terminate in a finite length of time.

- *Validity*: By the definition of sensitivity, the decision value at each process will be an argument to a past mutator instance, and only processes' input values are passed as inputs to mutators on the shared object. Thus, each decision value is some process' input value, and is valid.

- *Agreement*: $decode(result)$ will return the argument to the kth mutator instance at all processes. Since each process completes k mutator instances before it invokes AOP, there are guaranteed to be at least k mutator instances preceding the instance of AOP in line 4. Thus, each process decides the same value.

No part of the algorithm or proof is constrained by the number of participating processes, which means that this algorithm solves consensus for any number of processes using a k-front-sensitive data object, so the consensus number of any shared object with a k-front-sensitive set of operations is ∞. □

6.4 Consensus with End-Sensitive Data Types

While data types which "remember" which mutator instance was first, or kth as above, are intuitively very useful for consensus, other data types can also solve consensus, though not necessarily for an arbitrary number of processes. As a motivating example, consider the difference in semantics and consensus numbers between stacks and queues, shown in [15]. Both store elements given them in an ordered fashion, and the basic version of each has consensus number 2. However, adding extra power to a queue in the form of a $Peek$ operation gives it consensus number ∞, while adding a similar operation Top to stacks does not give them any extra power.

If we view the difference between an augmented queue and an augmented stack in terms of sensitivity, $Enqueue$ and $Peek$ on a queue are front-sensitive, while $Push$ and Top on a stack are end-sensitive. That is, queues see what operation was first, while stacks see which was latest. When processes cannot tell how far in the algorithm other processes have gotten, though, due to asynchrony, knowing what operation was latest is not helpful for consensus, as another mutator instance could finish after some process decides, and that other process will see a different latest value. We explore generalizations of this problem and what power still remains in end-sensitive data types.

Unfortunately, the picture for data types with end-sensitive operations sets is more complex than that for front-sensitive types. Here, we have variations depending on exactly which part of

the end of the previous history is visible or partly visible to an accessor. It is also important that shared objects have a pure accessor, or some other means of maintaining the state of the object, or else every operation will change what future operations see, making it difficult or impossible to come to a consensus.

We begin with a symmetric definition to that in Section 6.3, but for recent operations instead of initial, and show that it is not useful for consensus. We then show that certain subclasses, which are sensitive to more than one past operation, have higher consensus numbers.

Definition 30. *A subset OPS of the operations of a data type T is* k-end-sensitive *for a fixed integer k if OPS is S-sensitive where $S(\rho)$ is the kth-last mutator instance in ρ for every $\rho \in \mathcal{L}_T$ consisting entirely of instances of operations in OPS and containing at least k mutator instances, and $S(\rho)$ is a null operation instance $\perp(\perp, \perp)$, if there are not at least k mutator instances in ρ.*

This definition does not lead to as simple a result as that for front-sensitive sets of operations. As we will show, there is no algorithm for solving consensus for n processes with an arbitrary k-end-sensitive set of operations, for $n > 1$. We will give a number of more fine-grained definitions, showing that different subsets of the class of k-end-sensitive operation sets range in power from consensus number 1 to consensus number ∞.

Consider a set of operations which is S-sensitive, where for all ρ, $S(\rho)$ is the entire sequence of mutator instances in ρ. This set of operations is both k-end-sensitive and k-front-sensitive, for $k = 1$. By the result from Section 6.3, we know that such a set of operations has consensus number ∞. A similar result holds for any k for which an operation set is k-front-sensitive. Thus, in this section, we will only consider operation sets which are not k-front-sensitive for any k to consider the strength and limitations of end-sensitivity independently.

6.4.1 k-End-Sensitive Types

Unlike front-sensitive data types, if a set of operations is strictly k-end-sensitive, for some fixed k, the data type does not have infinite consensus number. This is a result of the fact that the kth-last mutator instance is a constantly moving target, as processes execute more mutator instances. As we

will show, in an asynchronous system, if there are more than one or three processes in the system (depending on the operations in the set), operations can be scheduled such that the "moving target" is always obscured for some processes, so they cannot distinguish which process took a step first after a critical configuration, which prevents them from safely deciding any value. We formalize this in the following theorems.

Theorem 30. *For $k > 2$, any data type with a strictly k-end-sensitive operation set consisting only of pure accessors and pure mutators has consensus number 1.*

Proof. Suppose we have a consensus algorithm A for at least 2 processes, p_0 and p_1, using such an operation set. Consider a critical configuration C of an execution of algorithm A, as per Lemmas 1, 2. If p_0 is about to execute a pure accessor, p_1 will not be able to distinguish C from the child configuration $p_0(C)$ when running alone, by the definition of a pure accessor. Thus, it will decide the same value in the executions where it runs from either of those states, which contradicts the fact that they have different valencies. If p_1's next operation is a pure accessor, a similar argument holds.

Thus, both processes' next operations from configuration C must be mutators. Assume without loss of generality that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent. Then the states $C_0 = p_1(p_0(C))$ and $C_1 = p_0(p_1(C))$ are likewise 0-valent and 1-valent, respectively.

We construct a pair of executions, extending C_0 and C_1, in which at least one process cannot learn which configuration it is executing from. By the Termination condition for consensus algorithms, at least one process must decide in a finite number of steps, and since the two executions return the same values to the first process to decide, it will decide the same value after $p_1(p_0(C))$ as after $p_0(p_1(C))$, despite those configurations having different valencies. This is a contradiction to the supposed correctness of A, showing that no such algorithm can exist.

We construct the first execution, from C_0, as follows. Assuming for the moment that both processes continue to execute mutators (we will discuss below what happens when they don't), let p_0 run alone until it is ready to execute another mutator. Then pause p_0 and let p_1 run alone until it is also ready to execute a mutator, and pause it. Let p_0 run alone again until it has completed $k - 2$

mutator instances and is ready to execute another. Next, allow p_1 to run until has executed one mutator instance, and is prepared to execute a second. We then continue to repeat this sequence, allowing p_0 to run alone again for $k-2$ mutator instances, then p_1 for one, etc.

The second execution is constructed identically from C_1 except that after C_1, p_0 first runs until it has executed $k-3$ mutator instances and is ready to execute another, then p_1 executes a mutator instance. After that, the processes alternate as in the first execution, with p_0 executing $k-2$ mutator instances and p_1 executing one.

We know that each process, running alone from C_0 (or C_1), must execute at least $k-2$ mutator instances to be able to see what mutator instance was first after C, since we have a strictly k-end-sensitive set of operations, which means that any correct algorithm must execute at least that many mutator instances, since it must be able to distinguish $p_0(C)$ from $p_1(C)$. The way we construct the executions, though, we interleave the operation instances in such a way that each process sees only its own operation instances, and cannot distinguish these executions from running alone from C_0 (or C_1). It is an interesting feature of this construction that we do not force any processes to crash. In fact, we need both processes to continue running to ensure that they successfully hide their own operations from each other.

If we denote any mutator instance by m and any accessor instance by a, with subscripts to indicate the instance's invoking process and superscripts for repetition (in the style of regular expressions), we can represent these two execution fragments, restricted to the shared object operated on in configuration C, as follows:

$$m_0 \cdot m_1 \cdot \cdot a_0^* \cdot a_1^* \cdot (m_0 \cdot a_0^*)^{k-2} \cdot (m_1 \cdot a_1^*) \cdot (m_0 \cdot a_0^*)^{k-2} \cdots$$

$$m_1 \cdot m_0 \cdot \cdot a_1^* \cdot a_0^* \cdot (m_0 \cdot a_0^*)^{k-3} \cdot (m_1 \cdot a_1^*) \cdot (m_0 \cdot a_0^*)^{k-2} \cdots$$

Since the return value of each accessor instance is determined by the kth most recent mutator instance, all operations are pure, and operations are deterministic, we can see that corresponding accessor instances will return the same value in the two executions. Thus, neither process can

distinguish the two executions. This is true despite the possibility of operations on other shared objects. To discern the two runs, each process must determine which process executed an operation first after C, and that can only be determined by operations on this shared object. Thus, as long as the return values of operation instances on this object are the same, since the algorithm is deterministic, the processes will continue to invoke the same operations in the two runs, and will be unable to distinguish the two executions.

This interleaving of operation instances works as long as both processes continue to invoke mutators. Each process must decide after a finite time, though, so they cannot continue to invoke mutators indefinitely. When a process ceases to invoke mutators, we can no longer schedule events as before to continue hiding its past operation instances. There are two possible cases for which process(es) finish their mutator instances first in the two executions.

First, one process (WLOG p_0) may execute its last mutator instance before the other does in both executions. When p_0 executes its last mutator instance in each execution, let it continue to run alone until it decides. Since configuration C, it has only seen its own mutator instances, and since the data type is strictly k-end-sensitive and no more mutators are executed, will continue to see only its own past mutator instances in both executions. Thus, the two executions are identical for p_0 and it will decide the same value in both, contradicting their differing valencies.

Second, it may be that in one execution, p_0 executes its last mutator instance before p_1 does and in the other, p_1 executes its last mutator instance before p_0. Each process will follow the same progression of local states in both executions, so this case can only arise when p_0's last mutator instance in the first execution is the last in a block of $k-2$ mutator instance it executes while running by itself, and thus the first instance in such a block in the second execution. In the first execution, after p_0 executes its last mutator instance, let it run alone, as in the first case. In the second execution, after p_1 executes its last mutator instance, pause it, and allow p_0 to run alone, executing its last mutator instance and continuing until it decides. By the same argument as case 1, p_0 decides the same value in both executions, contradicting the fact that they have the same valency.

Thus, the assumed consensus algorithm cannot actually exist. □

If mixed operations are allowed, the above proof does not hold, as a mixed operation immediately after C will potentially have a different return value than it would in a different execution where there is an intervening mutator instance. We can show the following:

Theorem 31. *For $k > 2$, any data type with an operation set which is strictly k-end-sensitive has consensus number at most 3.*

Proof. Suppose we have a consensus algorithm A for at least 4 processes, p_0, p_1, p_2, and p_3, using such an operation set. Consider a critical configuration C of an arbitrary execution of algorithm A. If p_0 is about to execute a pure accessor, p_1 will not be able to distinguish C from the child configuration $p_0(C)$ when running alone, by the definition of a pure accessor. Thus, it will decide the same value in the executions where it runs from either of those states, which contradicts the fact that they have different valencies. If any other process' next operation is a pure accessor, a similar argument holds.

Thus, all four processes' next operations from configuration C must be mutators. Assume without loss of generality that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent. Then the states $C_0 = p_2(p_1(p_0(C)))$ and $C_1 = p_2(p_0(p_1(C)))$ are likewise 0-valent and 1-valent, respectively.

We construct two executions E_0 and E_1, extending C_0 and C_1, respectively. We design these such that p_0 and p_1 crash, while p_2 and p_3 cannot distinguish E_0 from E_1. By the Termination condition for consensus algorithms, at least one process must decide in a finite number of steps, and since no running process can tell with execution it is in, they will decide the same value in both. But since the two executions are of different valencies, this is a contradiction to the supposed correctness of A, showing that no such algorithm can exist.

We construct execution E_0 from C_0 as follows. p_0 and p_1 crash immediately in configuration C_0. Let p_2 run alone until it has completed $k-3$ mutator instances and is ready to execute another. If p_2 runs alone from either C_0 or C_1, it must execute at least $k-2$ more mutator instances on this shared object to be able to distinguish these two states, by the assumed sensitivity of the operation

set, so we know we can let it run this far. Next, allow p_3 to run until it executes one mutator instance, performs any other operations, and is ready to execute a second mutator instance. Now, let p_2 run alone again, for $k-2$ mutator instances and other interleaved operations, until it is prepared to execute a $(k-1)^{st}$ mutator instance. We now let p_3 execute another mutator instance and following accessors until it is ready to execute a second mutator. We then continue to repeat allowing p_2 to run alone again for $k-2$ mutator instances, then p_3 for one, etc.

The second execution is constructed similarly, from C_1, with p_0 and p_1 crashing in configuration C_1, except that p_2 continues from C_1 until it completes $k-4$ mutator instances, then p_3 executes one, and then they continue as above, with p_2 executing $k-2$ mutator instances in a row, followed by p_3 executing one, and so on.

If we denote any sequence consisting of a single mutator instance followed by any number of pure accessor instances by $block$, with subscripts to indicate the instances' invoking process and superscripts for repetition (in the style of regular expressions), we can represent these two executions, restricted to the shared object operated on in configuration C, as follows:

$$block_0 \cdot block_1 \cdot (block_2)^{k-2} \cdot block_3 \cdot (block_2)^{k-2} \cdot block_3 \cdots$$

$$block_1 \cdot block_0 \cdot (block_2)^{k-3} \cdot block_3 \cdot (block_2)^{k-2} \cdot block_3 \cdots$$

Since the return value of each accessor instance is determined by the k^{th} most recent mutator instance, we can see that each accessor instance will return the same value in each execution. Thus, neither p_2 nor p_3 can distinguish the two executions. This is true despite the possibility of operations on other shared objects as in the proof of Theorem 30.

We now need only to argue that we can schedule the operation instances of any algorithm in the patterns specified in these executions, until some process decides. Up to $n-1$ processes can crash, and do so at any time, so p_0 and p_1 can crash as specified. As long as both running processes continue to invoke mutators, we can schedule them as we wish, because the system is asynchronous. When one of these processes invokes its last mutator instance, pause the other

process. If the process which invoked its last mutator instance runs alone, it must decide in a finite number of steps, by the Termination requirement of consensus. It can access the shared object, but since the object state doesn't change, that will not give it any more ability to discern which of the two executions it is in. It will then decide the same value in each case, leading to a contradiction. □

6.4.2 1- and 2-End-Sensitive Types

The bounds in the previous section require $k > 2$, so we here explore what bounds hold when $k \leq 2$. We continue to consider strictly k-end-sensitive operations; we will consider operation sets with knowledge of additional operation instances (that is, with larger sensitive sequences $S(\rho)$) later.

We first consider the case $k = 1$, which implies that accessor operations can see the last previous mutator instance. If all operations are pure mutators or accessors, then it is intuitive that consensus would not be possible, since we could schedule operations such that each process only saw its own mutator instances. We show that this is, in fact, the case. This generalizes the bound that registers can only solve consensus for one process. If mixed operations are allowed, then a process can obtain some information about other operation instances, which we will show is enough to solve consensus for two processes, but no more. We know that this bound of 2 is tight, that is, no lower bound can be proved for the entire class, since $Test\&Set$, for example, is sensitive to only the last previous mutator instance and has consensus number 2 [15].

Theorem 32. *Any data type with a strictly 1-end-sensitive operation set with no mixed operations has consensus number 1.*

Proof. Suppose there is an algorithm A which solves consensus for such an operation set on at least 2 processes. Let C be a critical configuration. Assume WLOG that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent.

If at least one process, say p_1, is prepared to execute a pure accessor in configuration C, then $p_1(C)$ and C will only differ in the local state of p_1. If p_1 crashes immediately, p_0 will behave

the same in both executions, and decide the same value, which contradicts the fact that $p_0(C)$ is 0-valent.

Thus, both p_0 and p_1 must be ready in C to execute mutators. Consider the configurations $p_0(C)$ and $p_0(p_1(C))$. Since the operation set is sensitive to only the last mutator instance, if p_0 runs alone from each of these configurations, it will never be able to ascertain the presence of p_1's operation instance in the second configuration, and will decide the same value in either case. This again contradicts the different valencies of the configurations.

Thus, there cannot be a critical configuration in the execution of A, which means that there is an execution in which it will never terminate. Since consensus algorithms must terminate, A cannot exist. □

Theorem 33. *Any data type with a strictly 1-end-sensitive operation set has consensus number at most 2.*

Proof. Suppose there is an algorithm A which solves consensus for such an operation set on at least 3 processes. Let C be a critical configuration. Assume WLOG that $p_0(C)$ is 0-valent and $p_1(C)$ is 1-valent.

If at least one process, say p_1, is prepared to execute a pure accessor in configuration C, then $p_1(C)$ and C will only differ in the local state of p_1. If p_1 crashes immediately, p_0 will behave the same in both executions, and decide the same value, which contradicts the fact that $p_0(C)$ is 0-valent.

Thus, both p_0 and p_1 must be ready in C to execute mutators. Unless both processes are about to execute mixed operations, suppose WLOG that p_0 is about to execute a pure mutator. Consider the configurations $p_0(C)$ and $p_0(p_1(C))$. Since the operation set is sensitive to only the last mutator instance and p_0's operation is a pure mutator, if p_0 runs alone from each of these configurations, it will never be able to ascertain the presence of p_1's operation instance in the second configuration, and will decide the same value in either case. This again contradicts the different valencies of the configurations.

If both p_0 and p_1 are prepared to execute mixed operations in C, again consider the configurations $p_0(C)$ and $p_0(p_1(C))$. If we allow a third process p_2 to run alone from these two configurations, it will not be able to distinguish them, because the operation set is strictly 1-end-sensitive. Thus, p_2 will decide the same value in both cases, contradicting their different valencies.

Thus, there cannot be a critical configuration in the execution of A, which means that there is an execution in which it will never terminate. Since consensus algorithms must terminate, A cannot exist. □

Next, we consider $k = 2$. If the sensitive set of operations includes a pure accessor, we show that we can solve consensus for 2 processes. Here, unlike our other results, the presence or absence of a mixed operation does not seem to affect the strength for consensus. Instead, it is important to have a pure accessor, which can see the 2nd-last mutator without changing it, which makes it practical for both processes to see the same value.

Data types without a pure accessor seem to have less power than consensus, since it is impossible to check the shared state without changing it. This makes it very difficult for processes to avoid confusing each other. A similar argument to that for Theorem 31 provides an upper bound of $n \leq 3$ for this data type. We conjecture that it is lower($n = 1$), but do not yet have the tools to prove this formally.

For now, an upper bound on the consensus number of 2-end-sensitive operation types is an open question, but we conjecture that it will be 2, or perhaps 3 with mixed operations as for k-end-sensitive types with $k > 2$, above.

Theorem 34. *For $k = 2$, a data type containing a k-end-sensitive set of operation types which includes a pure accessor has consensus number at least 2, using Algorithm 8.*

Proof. The algorithm has no wait or loop statements, so it will always terminate in a finite number of steps. Similarly, processes always decide either their own input or a decoded input from the other process, and they only do the latter when there has actually been such a value put into the shared object, so the algorithms satisfy validity.

To prove agreement, consider the possible decision points in the algorithm. If one process passed the `if` statement, then it saw $\perp$ when reading the shared state. But one process' mutator instance must appear before the other's in the execution, which means that the second process, which accesses the object no sooner than it executes its mutator, would see that two operation instances (including its own) had completed, and would not decode $\perp$. Thus, it would fail the `if` condition, decoding the argument of the first mutator instance in the execution, which, because there are only ever two invoked, is that belonging to the first process, and they both decide the first process' input.

If both processes were in the `else` case, then both saw two mutator instances. Since only two mutator instances are ever invoked on the shared object, and they must appear in the same order to both processes, the processes would decode, and decide, the same value. Thus agreement is also satisfied, and this is a correct consensus algorithm for two processes. □

Algorithm 8 Consensus Algorithm for 2 processes using 2-end-sensitive set of operations using mutator OP and pure accessor AOP

```
1: OP(input)
2: val ← AOP()
3: if decode(val) = ⊥ then
4:     decide(input)
5: else
6:     decide(decode(val))
```

6.4.3 Knowledge of Consecutive Operations

End-sensitive operation sets which only allow a process to learn about one past operation are generally limited to solving consensus for at most a small constant number of processors. We now show that knowledge about several consecutive recent operation instances allows more processes to solve consensus. In effect, we are enlarging the moving target we discussed before. We will show that this does, in fact, allow consensus algorithms on more processes, as many as the size of

the target, or the number of consecutive operation instances we can decode. We will then show that when we know the last mutator instances that have happened, the bound is tight.

This is interesting because the consensus number is not affected by how old the visible operations are, as long as they are at a consistent distance in the past. That is, if we always know a window of history that is a certain fixed number of operation instances old (no matter what that number is), we can use it to solve consensus. Also interesting is the fact that the bound is parameterized. While knowing a single element of history can solve consensus for a constant number of processes, if we know l consecutive mutator instances in the history, we can solve consensus for l processes for any natural number l. Thus, knowing more consecutive elements always increases the consensus number.

We could use this to create a family of data types which solve consensus for an arbitrary number of processes, with a direct cost trade-off. If we maintain a rolling cache of several consecutive mutator instances, we trade off the size of the cache we maintain against the number of processes which can solve consensus. If we only need consensus for a few processes, we know we only need to maintain a small cache. If we have the available capacity to maintain a large cache, we can solve consensus for a large number of processes.

We begin by defining the sensitivity of these large-target operation sets and giving a consensus algorithm for them. In effect, the algorithm watches for the target to fill up, and as long as it is not full, can determine which process was first. Since we can only see instances as long as the target "window" does not overflow, this gives the maximum number of processes which can use this algorithm to solve consensus. We later show this number is tight, if there are no mixed operations.

Definition 31. *A subset OPS of the operations of a data type T is* l-consecutive-k-end-sensitive *for fixed integers l and k if OPS is S-sensitive where for every $\rho \in \mathcal{L}_T$, $S(\rho)$ is the sequence of l consecutive mutator instances in ρ, the last of which is the kth-last mutator instance in ρ. If there are not that many mutator instances in ρ, the missing ones are replaced by $\perp(\perp, \perp)$ in $S(\rho)$.*

Theorem 35. *Any data type with an l-consecutive-k-end-sensitive set of operations has consensus number at least l, using Algorithm 9.*

Algorithm 9 Consensus algorithms for l processes using an l-consecutive-k-end-sensitive operation set. (A) Using mutator OP and pure accessor AOP. (B) Using mixed operation BOP.

(A)

1: **for** $x = 1$ to k **do**
2: $OP(input)$
3: $vals[1..l] \leftarrow decode(AOP())$
4: let $m = \arg\min_{n \in 1..l}\{vals[n] \neq \perp\}$
5: **if** m exists **then**
6: $decide(vals[m])$
7: **end for**

(B)

1: **for** $x = 1$ to k **do**
2: $vals[1..l] \leftarrow decode(BOP(input))$
3: let $m = \arg\min_{r \in 1..l}\{vals[r] \neq \perp\}$
4: **if** m exists **then**
5: $decide(vals[m])$
6: **end for**
7: $decide(input)$

We will show that this is the maximum possible number of processes for which we can give an algorithm which solves consensus using any l-consecutive-k-end-sensitive operations set. We do this by considering a special case of that class, l-consecutive-0-end-sensitive with only pure operations, and showing that the bound is tight for it. As with most end-sensitive classes, a set of operations which satisfies the definition of l-consecutive-k-end-sensitive may also be sensitive to more, earlier operations, and thus have a higher consensus number. We will show a particular example of such an operation set, to show that there is more work to be done to classify end-sensitive data types.

Theorem 36 below shows an upper bound on the consensus number of strictly l-consecutive-0-end-sensitive operation sets. That is, operation sets in which accessors can learn exactly the last l mutator instances. To achieve this bound, we need to restrict ourselves to operation sets which have no mixed accessor/mutator operations. This is a strong restriction, but we will give an example showing that a mutator which also returns even a small amount of information about the state of the shared object can increase the consensus number of an operation set.

Theorem 36. *Any data type with a strictly l-consecutive-0-end-sensitive set of operations which has no mixed accessor/mutators has consensus number at most l.*

Proof. Assume we have a set of operations as specified in the theorem and an algorithm A which uses them to solve consensus for $l+1$ processes. Let C be a critical configuration, by Lemmas 1, 2,

which also imply that all processes must be about to execute an operation on the same shared object. We consider the possible operations which processes may be about to execute.

- A process p_i is about to execute a pure accessor, and $p_i(C)$ and $p_j(C)$ have different valencies, for some other process p_j:

  By the definition of a pure accessor, configurations $p_j(C)$ and $p_j(p_i(C))$ will have the same shared state, and p_j has the same state in both. Thus, if p_j runs alone from either of these configurations, it will decide the same value, contradicting their difference in valency, so this case cannot occur.

- All processes are prepared to execute mutators:

  Assume, WLOG, that $p_0(C)$ is 0-valent, and $p_1(C)$ is 1-valent. If each process $p_i, i \in \{0..l\}$ takes a step, in order, the resulting configuration is $p_l(p_{l-1}(...(p_1(p_0(C))...)$. However, since the set of operations is strictly l-consecutive-0-end-sensitive, p_l will not be able to distinguish this configuration from the one in which p_0 does not act, but all other processes execute their operations in the same order: $p_l(p_{l-1}(...(p_1(C)...)$. Thus, if p_l runs alone from either configuration, it will decide the same value in each case, which contradicts the fact that the first configuration is 0-valent and the second is 1-valent. Thus, this case cannot occur.

Since every possible set of ready operations at any critical configuration leads to a contradiction, we conclude that there is no such algorithm A to solve consensus for $l + 1$ processes using the specified set of shared operations. □

There are sets of operations which are strictly l-consecutive-0-end-sensitive, but have a mixed operation which returns information about the state of the object. We here give an example such set. Specifically, the mixed operation returns a (limited) count of the number of preceding mutator instances. Even this small amount of extra information is enough to increase the consensus power of a set of operations.

Consider an l-element shared cyclic queue with operations $Enq_l(x)$ and $ReadAll()$. $Enq_l(x)$ is a mixed accessor/mutator which adds x to the tail of the queue, discarding the head element if

there are more than l elements in the queue, and returning the number of Enq_l operation instances which have previously been executed, up to l. If more than l Enq_l operation instances have been previously executed, the return value will continue to be l. $ReadAll()$ is a pure accessor which returns the entire contents of the l-element queue. This is clearly a strictly l-consecutive-0-end-sensitive set of operations, since the return values of $ReadAll()$ and Enq_l depend on the last l $Enq_l(x)$ instances, but only the last l are visible to each instance of one of these. We show that it has consensus number at least $l + 1$ by giving Algorithm 10.

Algorithm 10 Algorithm for each process i to solve consensus for $l+1$ processes using a l-element cyclic queue with Enq_l and $ReadAll$

```
1: Write_i(input)                                   ▷ In a shared SWMR register
2: state ← Enq_l(i)
3: l_history ← (ReadAll())
4: if There are state values preceding i in l_history then
5:     decide oldest element in l_history
6: else
7:     j ← processor id not appearing in l_history
8:     decide Read_j()                              ▷ Value from p_j's SWMR register
```

The intuition for this algorithm is that all processes but one will be able to see which process was first. The variable $state$ will tell how many previous Enq_l instances processes have executed. If this is less than k, all previous Enq_l instances are visible, and the process can return the input of the first. If there have been k previous Enq_l instances, then we cannot see the first, but we know that there are at most $l+1$ processes and each executed only one Enq_l instandce, so the one process whose Enq_l instance we cannot see must have been first, and we decide that process' input.

This algorithm shows that mixed operations can give extra strength for consensus, beyond sensitivity, which is difficult to quantify. In general, mixed operations can not only give different return values based on the state of the shared object, but can alter the way they modify the object's state based on its previous state. This allows them to preserve any non-empty state, which means that it can keep a record of which process first modified the state, giving a front-sensitive data type,

which can solve consensus for any number of processes. For example, a $Read\text{-}Modify\text{-}Write$ operation can exhibit this behavior.

6.5 Conclusion

We have defined a number of classes of operations for shared objects, and explored their power for solving consensus. First, we generalized, with an intuitive result, the common understanding that knowing what process acted on a shared object first, or in some fixed position relative to the start of the execution order, allows a consensus algorithm for any number of processes. We then considered what might be possible if only knowledge about recent operation instances, instead of initial instances, is available.

Here, because the set of recent operation instances is constantly changing, we must be more precise about what knowledge is available. If operations cannot both change and view the shared state atomically, then the number of processes which can solve consensus is given by the number of consecutive changes a process can view atomically. Further, these do not need to be the most recent changes, as long as processes know how old the data they receive is.

If operations can atomically view and change the shared state, then they generally have the potential for more computational power. We show that if an operation set has a mixed operation which can see one of the two most recent changes, then it can solve consensus for two processes, where without a mixed operation, such an operation set could only solve consensus for one process. In general, though, allowing arbitrary mixed operations allows an arbitrary number of processes to solve consensus, depending on the power of the mixed operation. Also, mixed operations may be more expensive to implement than pure accessors or mutators, which would cause a trade-off between computational power and operation cost.

We summarize our results in Table 6.1. We have results for front-sensitive sets of operations and several subclasses of end-sensitive operation sets. Several of these classes have different consensus numbers if we allow mixed accessor/mutator operations or only allow pure accessors and pure mutators, so we separate those results. Note also that all upper bounds further assume a data type with a strictly sensitive set of operations.

Table 6.1: Summary of Upper and Lower Bounds on Consensus Numbers

Operation Set			Lower Bounds		Upper Bounds	
			Pure	Mixed	Pure	Mixed
Front-sensitive			∞		-	
End-Sensitive	k-end:	$k > 2$	1	?	1	3
		$k = 1$	1	2	1	?
		$k = 2$	2	?	?	3
	l-consecutive-k-end		l		l ($k = 0$)	?

In future work, we wish to fill missing entries in the above table. In addition, we wish to further explore conditions on the knowledge of the execution which operations can extract to classify more operations. More generally, the idea of exploring how information travels through the execution history of a shared object, affecting the return values of different subsequent operations in different ways, is fascinating. As currently defined, sensitivity cannot classify all possible operation sets, so an exploration of classifying and providing generic results for other shared data types is of interest.

Another direction is to consider trade-offs between the implementation costs of shared operations and their consensus numbers. It would be interesting to develop a metric which balances an operation's cost with its computational strength. Finding minima of such a metric would be an interesting result, potentially showing the optimal cost for solving consensus for any given number of processes.

7. SUMMARY AND CONCLUSIONS

In this dissertation, we have worked to extend our understanding of distributed data structures. We want to provide simple, efficient mechanisms for distributed programs to access shared data in an environment without shared hardware. Because communication time is dominant in such systems, we focus on reducing that aspect of the cost.

Our primary focus has been on data type relaxations, and moving them from a theoretical idea towards useful and understandable concepts. To do this, we have first given several equivalent definitions, each useful in a particular context, to make the properties more readily understandable and usable by humans. From there, we have given some sample implementations to show that relaxations allow higher-performance implementations. We found that worst-case communication time per operation does not improve with many relaxations, but the amortized cost can decrease significantly. Further, there is an inherent increase in performance as relaxation is increased. This suggests that relaxed data types can provide tunable performance for different applications.

Once we knew that relaxation can improve performance, we wanted to analyze what is sacrificed to allow this. Relaxing a data type adds some non-determinism, which could reduce the usefulness of the data type. We analyzed all possible parameter values for three relaxations of queues, showing that some computational power was given up. This suggests that, in addition to performance tuning, a developer can tune the power of their data types by adjusting the relaxation, and that they must be conscious of the effects of their parameter choices.

The last question about relaxed data types in this work is whether they are a good model for building high-performance shared data types. To answer this, we explored the relationship between data type relaxations and weak consistency conditions, which is the standard approach in the literature for weakening guarantees on concurent behavior. We show that several common data type relaxations can be equivalently expressed as consistency conditions. In general, our method could express any data type relaxation as a consistency condition. Conversely, we show that some known consistency conditions can be expressed as relaxations. This equivalence is partial, though,

as some weak consistency conditions use features such as knowledge of concurrency, which does not exist in a sequential specification, even if is relaxed. We can, however, use tools from the extensive literature on consistency conditions to analyze data type relaxations. We do this, showing that the relaxations we consider, which are intuitive and fairly common in the literature, are unique and different from studied consistency conditions. This establishes that data type relaxations are a useful tool, since they allow us to consider behaviors that were not intuitive to specify as consistency conditions.

To conclude this dissertation, we stepped away from relaxed data types and considered the broader question of evaluating the computational power of a data type. We gave heuristics to determine a type's consensus number. While this is generally undecidable, we hope that our heuristics may allow greater intuitive insight into what properties of data types cause them to have or lack consensus power.

Overall, we extended our understanding of distributed data types and techniques to increase their performance. We analyzed the tradeoffs and models we used, and provided some tools for analyzing data types in general. We hope that these results will lead to application benefits in a variety of distributed systems.

7.1 Future Work

7.1.1 Consistency Conditions vs. Relaxations

We have only begun exploring the relation of consistency conditions and relaxed data types. So far, we have shown that relaxations can be equivalently described as consistency conditions, and some consistency conditions can be expressed as relaxations. We have also taken advantage of tools in the consistency condition literature to compare the strength of data type relaxation definitions to consistency conditions. In future work, I would like to more fully explore the implications and uses of this partial equivalence. It may be possible to generalize the definition of data type relaxations to represent more consistency conditions than the current definition allows. In large part, the difficulty here is finding a way to represent concurrency in a sequential specification. This

begins to approach other work (e.g. [2, 42]) on the specification of problems which cannot be represented sequentially. Perhaps a broader definition of relaxation could apply to such problems, as well.

In another direction, an interesting possible method for describing, reasoning about, and implementing weak consistency conditions is to combine relaxations with consistency conditions weaker than linearizability. One benefit of this approach is that it moves part of the complexity of possible behaviors to the sequential world, which is typically easier to reason about. It remains open to explore whether this separation of definition could open the path to more efficient implementations or easier impossibility and lower bound proofs for data types. Another possible benefit is that extending relaxations to work with arbitrary consistency conditions could also allow extensions to the equivalence between the two systems. Ideally, we would be able to express a larger portion of the space of consistency conditions as data type relaxations, and vice versa. It still seems that some aspects of concurrent behavior would be impossible to express sequentially, but that leads us to the question of where the edges of the equivalence are.

7.1.2 Practical Implementations of New and Arbitrary Data Types

In this work, we have almost exclusively considered relaxations of FIFO queues. This is a good first step, as they are well-known, easy to work with, and relax intuitively. However, the concept of relaxation can apply to many more data types, potentially giving performance improvements in applications which rely on different shared-data semantics. We would like to implement relaxations of a variety of other commonly used data types to work towards a library of high-performance shared data type implementations which a developer can choose between based on their desired balance between efficiency and semantic guarantees.

In past work [3], we presented an algorithm for an arbitrary data type in a partially-synchronous environment. This model approximates a real-world system, but relies on upper bounds on message delays that may not be realistic. There are many interesting question involved in continuing this work in a truly asynchronous message-passing system, attempting to find an optimal implementation for any data type. Such an implementation should be easy to port to a real-world system,

since it assumes very little about the environment. A general implementation which can be instantiated to implement an arbitrary data type is often very difficult to achieve, and may not even be possible. It thus is probably easiest to start with implementations of various data types of interest, with the end goal of generalizing implementations as much as possible.

Another direction to take these implementations is to extend in the direction of relaxation. [3] presents a general algorithm in a partially synchronous system, but also restricts it to only deterministic operations. If this latter constraint can be removed, then the general implementation could also be applied to relaxed and other non-deterministic data types.

7.1.3 Classifying Operations

To show that a general implementation of abstract data types is optimal or near-optimal, we can start with the lower bounds from the partially-synchronous environment, since we are weakening assumptions. Those bounds are not complete, though, covering only operations with certain algebraic properties. An ongoing task, vital to optimal implementations of arbitrary data types, is to expand the classification ([32, 3], etc.) of all possible operations on data types, so that we can prove lower bounds on all operations.

Since there are infinitely many possible operations and interesting classes of operations, we can start by exploring the properties that characterize operations of particular interest in distributed data types. An obvious example characteristic is that operations must have some effect, either storing or reporting information. If we can find features that seem necessary for an operation to be of value, then we can prioritize the search for new operation classifications.

7.1.4 Applications of Relaxed Data Types

Finding applications that benefit from the increased efficiency of relaxed types but do not suffer too much from the relaxed guarantees is an area that promises many interesting questions. For example, task allocation for robot swarms has the potential for increased throughput if some tasks may be multiply assigned, while others are assigned out of order. With proper tuning, it may be possible to increase total performance, without sacrificing too much from guarantees on priority

ordering. Such problems would be ideal for collaboration with domain experts from a variety fields, who could identify tasks which do not rely too heavily on deterministic guarantees such as ordering, but which would benefit from faster distributed execution. This is the ultimate goal of research on relaxed data types: using these new types to solve new and interesting problems.

REFERENCES

[1] G. Neiger, "Set-linearizability," in *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing, Los Angeles, CA, USA, August 14-17, 1994* (J. H. Anderson, D. Peleg, and E. Borowsky, eds.), p. 396, ACM, 1994.

[2] A. Castañeda, S. Rajsbaum, and M. Raynal, "Specifying concurrent problems: Beyond linearizability and up to tasks - (extended abstract)," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings* (Y. Moses, ed.), vol. 9363 of *Lecture Notes in Computer Science*, pp. 420–435, Springer, 2015.

[3] J. Wang, E. Talmage, H. Lee, and J. L. Welch, "Improved time bounds for linearizable implementations of abstract data types," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pp. 691–701, IEEE Computer Society, 2014.

[4] T. A. Henzinger, C. M. Kirsch, H. Payer, A. Sezgin, and A. Sokolova, "Quantitative relaxation of concurrent data structures," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy, January 23 - 25, 2013* (R. Giacobazzi and R. Cousot, eds.), pp. 317–328, ACM, 2013.

[5] E. Talmage and J. L. Welch, "Improving average performance by relaxing distributed data structures," in *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings* (F. Kuhn, ed.), vol. 8784 of *Lecture Notes in Computer Science*, pp. 421–438, Springer, 2014.

[6] E. Talmage and J. L. Welch, "Relaxed data types as consistency conditions," in *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings* (P. G. Spirakis and P. Tsigas, eds.), vol. 10616 of *Lecture Notes in Computer Science*, pp. 142–156, Springer, 2017.

[7] N. Shavit and G. Taubenfeld, "The computability of relaxed data structures: Queues and stacks as examples," in *Structural Information and Communication Complexity - 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015, Post-Proceedings* (C. Scheideler, ed.), vol. 9439 of *Lecture Notes in Computer Science*, pp. 414–428, Springer, 2015.

[8] E. Talmage and J. L. Welch, "Anomalies and similarities among consensus numbers of variously-relaxed queues," in *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings* (A. E. Abbadi and B. Garbinato, eds.), vol. 10299 of *Lecture Notes in Computer Science*, pp. 191–205, Springer, 2017.

[9] C. M. Kirsch, M. Lippautz, and H. Payer, "Fast and scalable k-fifo queues," Tech. Rep. 2012-04, Department of Computer Sciences, University of Salzburg, June 2012.

[10] H. Rihani, P. Sanders, and R. Dementiev, "Brief announcement: Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015* (G. E. Blelloch and K. Agrawal, eds.), pp. 80–82, ACM, 2015.

[11] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015* (A. Cohen and D. Grove, eds.), pp. 277–278, ACM, 2015.

[12] P. Viotti and M. Vukolic, "Consistency in non-transactional distributed storage systems," *ACM Comput. Surv.*, vol. 49, no. 1, pp. 19:1–19:34, 2016.

[13] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[14] A. Aiyer, L. Alvisi, and R. A. Bazzi, "On the availability of non-strict quorum systems," in *Distributed Computing* (P. Fraigniaud, ed.), vol. 3724 of *Lecture Notes in Computer Science*, pp. 48–62, Springer Berlin Heidelberg, 2005.

[15] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[16] W. Lo and V. Hadzilacos, "All of us are smarter than any of us: Nondeterministic wait-free hierarchies are not robust," *SIAM J. Comput.*, vol. 30, no. 3, pp. 689–728, 2000.

[17] M. J. Fischer, N. A. Lynch, and M. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[18] Y. Afek, G. Korland, and E. Yanovsky, "Quasi-linearizability: Relaxed consistency for improved concurrency," in *Principles of Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings* (C. Lu, T. Masuzawa, and M. Mosbah, eds.), vol. 6490 of *Lecture Notes in Computer Science*, pp. 395–410, Springer, 2010.

[19] R. J. Lipton and J. S. Sandberg, "PRAM: A scalable shared memory," Tech. Rep. CS-TR-180-88, Princeton University, Department of Computer Science, September 1988.

[20] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Trans. Comput. Syst.*, vol. 12, no. 2, pp. 91–122, 1994.

[21] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. M. Michael, and M. T. Vechev, "Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated," in *POPL* (T. Ball and M. Sagiv, eds.), pp. 487–498, ACM, 2011.

[22] W. Chen, G. Hu, and J. Zhang, "On the power of breakable objects," *Theor. Comput. Sci.*, vol. 503, pp. 89–108, 2013.

[23] E. Talmage and J. L. Welch, "Generic proofs of consensus numbers for abstract data types," in *19th International Conference on Principles of Distributed Systems, OPODIS 2015, December 14-17, 2015, Rennes, France* (E. Anceaume, C. Cachin, and M. G. Potop-Butucaru, eds.), vol. 46 of *LIPIcs*, pp. 32:1–32:16, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[24] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings* (X. Défago, F. Petit, and V. Villain, eds.), vol. 6976 of *Lecture Notes in Computer Science*, pp. 386–400, Springer, 2011.

[25] D. Bermbach and J. Kuhlenkamp, "Consistency in distributed storage systems - an overview of models, metrics and measurement approaches," in *Networked Systems - First International Conference, NETYS 2013, Marrakech, Morocco, May 2-4, 2013, Revised Selected Papers* (V. Gramoli and R. Guerraoui, eds.), vol. 7853 of *Lecture Notes in Computer Science*, pp. 175–189, Springer, 2013.

[26] R. Friedman, R. Vitenberg, and G. V. Chockler, "On the composability of consistency conditions," *Inf. Process. Lett.*, vol. 86, no. 4, pp. 169–176, 2003.

[27] R. Vitenberg and R. Friedman, "On the locality of consistency conditions," in *Distributed Computing, 17th International Conference, DISC 2003, Sorrento, Italy, October 1-3, 2003, Proceedings* (F. E. Fich, ed.), vol. 2848 of *Lecture Notes in Computer Science*, pp. 92–105, Springer, 2003.

[28] L. Lamport, "On interprocess communication. part II: algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[29] C. Shao, J. L. Welch, E. Pierce, and H. Lee, "Multiwriter consistency conditions for shared memory registers," *SIAM J. Comput.*, vol. 40, no. 1, pp. 28–62, 2011.

[30] N. Hemed and N. Rinetzky, "Brief announcement: concurrency-aware linearizability," in *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014* (M. M. Halldórsson and S. Dolev, eds.), pp. 209–211, ACM, 2014.

[31] N. Hemed, N. Rinetzky, and V. Vafeiadis, "Modular verification of concurrency-aware linearizability," in *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo,*

Japan, October 7-9, 2015, Proceedings (Y. Moses, ed.), vol. 9363 of *Lecture Notes in Computer Science*, pp. 371–387, Springer, 2015.

[32] M. J. Kosa, "Time bounds for strong and hybrid consistency for arbitrary abstract data types," *Chicago J. Theor. Comput. Sci.*, vol. 1999, 1999.

[33] J. Wang, J. L. Welch, and H. Lee, "Time bounds for shared objects in partially synchronous systems," in *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011* (C. Gavoille and P. Fraigniaud, eds.), pp. 347–348, ACM, 2011.

[34] P. Jayanti and S. Toueg, "Some results on the impossibility, universality, and decidability of consensus," in *Distributed Algorithms, 6th International Workshop, WDAG '92, Haifa, Israel, November 2-4, 1992, Proceedings* (A. Segall and S. Zaks, eds.), vol. 647 of *Lecture Notes in Computer Science*, pp. 69–84, Springer, 1992.

[35] E. Borowsky, E. Gafni, and Y. Afek, "Consensus power makes (some) sense! (extended abstract)," in *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, (New York, NY, USA), pp. 363–372, ACM, 1994.

[36] R. A. Bazzi, G. Neiger, and G. L. Peterson, "On the use of registers in achieving wait-free consensus," *Distributed Computing*, vol. 10, no. 3, pp. 117–127, 1997.

[37] O. Rachman, "Anomalies in the wait-free hierarchy," in *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings* (G. Tel and P. M. B. Vitányi, eds.), vol. 857 of *Lecture Notes in Computer Science*, pp. 156–163, Springer, 1994.

[38] P. C. Attie, "Wait-free byzantine consensus," *Inf. Process. Lett.*, vol. 83, no. 4, pp. 221–227, 2002.

[39] E. Ruppert, "Consensus numbers of multi-objects," in *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998* (B. A. Coan and Y. Afek, eds.), pp. 211–217, ACM, 1998.

[40] E. Ruppert, "Determining consensus numbers," *SIAM J. Comput.*, vol. 30, no. 4, pp. 1156–1168, 2000.

[41] S. Chordia, S. K. Rajamani, K. Rajan, G. Ramalingam, and K. Vaswani, "Asynchronous resilient linearizability," in *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings* (Y. Afek, ed.), vol. 8205 of *Lecture Notes in Computer Science*, pp. 164–178, Springer, 2013.

[42] A. Castañeda, S. Rajsbaum, and M. Raynal, "Long-lived tasks," in *Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings* (A. E. Abbadi and B. Garbinato, eds.), vol. 10299 of *Lecture Notes in Computer Science*, pp. 439–454, Springer, 2017.