SIMULATING A SHARED QUEUE ON TOP OF EVENTUALLY SYNCHRONOUS MESSAGE-PASSING DISTRIBUTED SYSTEMS

An Undergraduate Research Scholars Thesis

by

EMMA ZIESMER

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jennifer L. Welch

May 2022

Major: Computer Science

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Emma Ziesmer, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety office.

TABLE OF CONTENTS

ABSTRACT

Simulating a Shared Queue on Top of Eventually Synchronous Message-Passing Distributed Systems

Emma Ziesmer
Department of Computer Science and Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jennifer L. Welch
Department of Computer Science and Engineering
Texas A&M University

Distributed computer systems are comprised of multiple processing nodes that communicate and coordinate actions in order to achieve a common goal. They are often used to handle tasks that are too large to be handled by a single processor or are inherently distributed. Distributed systems can be either a message-passing system, where nodes pass messages between each other to coordinate actions, or a shared-memory system, where nodes have access to a shared data object between all of the nodes. These shared objects provide a higher level of abstraction to the programmer than a message-passing system but may not be available to use. Instead, algorithms have been developed in the past that allow the nodes to keep a local copy of the shared data object and send messages between each other to coordinate modifications to the object. This way, concurrent operations on the implemented object are guaranteed to be linearizable, or give the illusion that the operations are executed sequentially and in a legal order. This thesis will examine one of these algorithms which was originally developed for a synchronous system, where there is an upper bound on message delays. We will instead examine the algorithm's behavior in a system that starts with arbitrarily long message delays, including messages that are lost and never delivered, but eventually stabilizes and has bounded message delays at a time unknown to the nodes.

We will be measuring the length of time needed for the execution to become linearizable after the system stabilizes, called the grace period. We will focus on the case where the shared object is a first-in-first-out Queue. We have simulated this algorithm using a discrete event simulator so that the timing elements can be precisely controlled.

DEDICATION

I dedicate this work to my dad, who has always pushed me to be the best that I can be.

ACKNOWLEDGMENTS

Contributors

I would like to thank my faculty advisor, Dr. Jennifer Welch, for her guidance and support throughout the course of this research. Thanks also go to my friends and family for their support and patience throughout the research process.

Funding Sources

This work did not receive funding.

NOMENCLATURE

AOP	Pure acessor operation
MOP	Pure mutator operation
XOP	Mixed operation
FIFO	First-in-first-out

1. INTRODUCTION

A distributed system is a computer system comprised of multiple processing nodes on a network that communicate and coordinate actions in order to achieve a common goal. One of the main purposes of using distributed systems is to handle projects that are too large for a single machine or are inherently distributed, such as with cloud computing. They can also be more fault tolerant than single machines because having multiple nodes avoids having a single point of failure. If one of these nodes experiences a crash failure, there is potential for the rest of the system to function as normal.

A shared-memory model of a distributed system in particular is a model in which all of the processes in the system share a single data object between them, as the name implies. This object can be accessed by multiple processes concurrently. Having a shared data object provides a higher level of abstraction than a message-passing model, where the nodes send messages to each other to coordinate their actions. This makes it easier for programmers to use data structures they are already familiar with.

However, shared memory is not always available to be used. Instead, shared objects can be implemented in a message-passing distributed system. In algorithms that achieve this, each process keeps a local copy of the shared object and messages are sent between the processes to keep the copies up to date [1, 2, 3, 4]. Because messages can take a significant amount of time to be delivered and each process needs to update their copy, operations on the simulated shared object are no longer instantaneous. Operations invoked by different processes can overlap in time, which necessitates a method of deciding the behavior of the implemented object when overlaps occur.

One popular option is to utilize linearizable shared objects. Linearizability is a consistency condition placed on implemented shared objects, in which an execution of concurrent operations on the object can be given a legal ordering as if they were sequential and instantaneous [5]. This gives an illusion to programmers that operations are in fact sequential and instantaneous, and overlapping

operations do not have to be considered when working with a shared object. The implementation hides the details of how overlapping operations are handled.

Several algorithms have been developed previously that simulate shared data objects in a message-passing system. One early algorithm focuses on implementing a shared register in an asynchronous message-passing system with crash failures [1]. In their model, they implement single-writer multi-reader registers. Nodes are subject to crash failures and links between nodes can fail in that messages can no longer be passed between the two nodes at each end of the link. A node that wants to invoke an operation will send messages to every other node containing the type of operation, the argument if any, and its timestamp. It will wait to receive acknowledgement responses from a majority of the other nodes before it executes its operation. Each operation on the register requires $O(n^2)$ messages, where n is the number of nodes in the system.

However, implementing any data structure more powerful than a read/write register has proven to be difficult, if not impossible. Each type of data structure, such as Stacks and Queues, can be given a consensus number, or the maximum number of nodes in a system that can solve the consensus problem using the given object. An object with consensus number n cannot be implemented with an object with consensus number lower than n while still being fault-tolerant [6]. It has been shown that solving the consensus problem is impossible for data structures with a consensus number greater than 1 in an asynchronous message passing system, including Stacks and Queues. This does not include read/write registers, as they have a consensus number of 1.

On the other hand, we can consider the case of a synchronous system, such as with the algorithm given in [2]. Their algorithm implements a generic data structure in a synchronous message-passing system where nodes cannot crash and there is an upper bound on message delays. When a node wants to invoke an operation, it will send a message containing the type of operation, the argument if any, and its timestamp to all of the other nodes. The nodes will then set a timer for that operation and execute it on its local copies once the timer goes off. Although this algorithm is theoretically simple to implement, it is difficult to apply to any real world system. The system used by the algorithm is assumed to have good behavior while real world systems tend to experience

faulty behavior to some degree.

A good middle ground between a fully asynchronous system and a fully synchronous system is one that starts poorly behaved but becomes well behaved after a certain point in time [4]. The system used by the algorithm in [4] starts as an asynchronous system with lost messages, unbounded message delays, and nodes that experience crash failures. After a point in time unknown to the nodes, called the system stabilization time, the system becomes synchronous with bounded message delays and no crash failures. In the algorithm, one node is elected as the leader, which can be replaced by another node if the current leader crashes. Any node that wants to invoke an operation on the shared object will send a message to the leader containing the type of operation, the argument if any, and its timestamp. The leader will take batches of operations and order them, then send the batches back to the other nodes. Every node will then execute the operations in the batch in order. Though this algorithm can be complex to implement, it is more readily applicable to real world systems than the algorithm in [2].

In this thesis, we would like to see how well the algorithm described in [2] is able to hold up in a system similar to that used in [4], where messages can take an arbitrarily long time to be sent or event be lost. We will not be considering nodes with crash failures at this time. The algorithm will be tested experimentally using a discrete event simulator so that timing elements and parameters can be precisely controlled. We will be focusing on a Queue with first-in-first-out (FIFO) behavior as the shared data object being implemented. A FIFO Queue can be thought of as a list of data, where data can only be removed or read from the front of the list and added to the back of the list.

Executions produced by the simulation will be checked for linearizability using a construction given in [2], which provides an ordering for the concurrent operations in the execution that is guaranteed to be linearizable in the synchronous system. For our purposes, this method of checking linearizability may be too conservative as it only provides a single possible ordering. If the check fails, it does not mean that the execution is definitely not linearizable, only that the specific ordering produced by the construction is not linearizable.

2. DEFINITIONS

2.1 Shared Data Object

The generic algorithm in [2] classifies operations on the shared object as pure accessors (AOP), pure mutators (MOP), and mixed operations (XOP). A pure accessor returns information about the state of the object without changing it. A pure mutator changes the state of the object in a way that a later sequence of operations can tell that the mutator occurred. Pure mutators do not return information to the user. A mixed operation is a combination of the two previous types of operations, where it both modifies the object and returns information about the state of the object.

The shared data object we will be focusing on in this thesis is a Queue with first-in-first-out (FIFO) behavior. A Queue can be thought of as a list of items, as implied by the name. When a Queue has FIFO behavior, the first item to be added to the queue is also the first item to be removed from the queue. In other words, items must be removed in the same order as they were added.

Basic operations on a FIFO Queue include:

- `enqueue()`,
- `dequeue()`, and
- `peek()`.

The `enqueue()` operation appends an item to the end of the queue. The `dequeue()` operation removes and returns the item at the front of the queue. When a user attempts to invoke `dequeue()` on an empty queue, the operation returns a special value signifying that the queue is empty. The `peek()` operation returns but does not remove the item at the front of the queue. Again, the operation returns a special value signifying that the queue is empty when a user attempts to invoke `peek()` on an empty queue.

To translate the generic algorithm to an algorithm for a FIFO Queue, `enqueue()` will be implemented as a pure mutator, `dequeue()` will be implemented as a mixed operation, and `peek()` will be implemented as a pure accessor.

2.2 System Model

The distributed system consists of n processes that communicate by passing messages between each other. Each process has a local state and takes steps triggered by one of three things: the invocation of an operation, the receipt of a message, or a timer going off. The local state includes a local clock value and a set of timers that have been set for a clock time in the future. The local clocks of each process are synchronized to within $\epsilon = (1 - 1/n)u$ of each other at all times. A step of a process causes the node to transition from its current local state to a new local state, which may cause messages to be sent, timers to be set or cancelled, or operation responses to occur.

An execution is a collection of sequences of steps, one for each node in the system. The sequence for a node must start with the initial local state of the node where no timers can be set, and timer steps that occur in the sequence must be consistent with the timers in the local state. Each step in a sequence is associated with a real time, which is increasing, and local clock times must be equal to some fixed offset from the real time.

In our modification of the model, there is initially no upper bound on message delays. Messages can take an arbitrarily long time to be delivered or even be lost. There is a lower bound of $d - u$ on the delay. Each message sent is received by at most one node, and every message received was previously sent by exactly one process. After a certain point in time unknown to the nodes, the system will stabilize and all messages sent must eventually be received. The message delay after this time is in the range $[d-u, d]$. Each message sent is received by exactly one process, and every message received was previously sent by exactly one process.

2.3 Problem Definition

Given an execution of operations on a shared object, a linearization of that execution is an ordering of the operations that can be thought of as having occurred sequentially [5]. This ordering can be found by selecting linearization points for each operation and sorting the operations in increasing order of the linearization points. A linearization point for an operation is a point in time after the operation has started and before it has completed where the operation can be thought of as occurring instantaneously at that time. The ordering of the operations in an execution must be

legal in the sense that the ordering respects the rules of a sequential version of the shared object.

Linearizability as a consistency condition for the shared object provides the illusion that operations are happening sequentially and instantaneously. This allows the programmer to reason about the shared object in a familiar way as if it was a sequential object. In reality, the operations on the shared object can overlap with each other in time—one process can invoke an operation before an operation invoked by another process has completed.

We know that the generic algorithm in [2] is linearizable in a synchronous distributed system in which there is an upper bound on message delays. In this thesis, we will be extending this algorithm by specializing it for a FIFO Queue and studying its behavior in a system that initially experiences message loss and arbitrarily long message delays but eventually becomes synchronous with bounded message delays. We would like to see how long it takes for the shared Queue to become linearizable once the system stabilizes, called the grace period.

Because we will be measuring the grace period using a simulation, the measured time will not correspond to any real-world units. Instead of focusing on any exact values produced by the simulation, we will be focusing on the general trend of the values as certain parameters change.

3. ALGORITHM

This algorithm accomplishes its goal of implementing a shared object in a message-passing distributed system primarily through the use of timers. As a high-level overview of the algorithm, whenever an operation is invoked or a message is received by a process, the process will set a timer for a certain amount of time depending on the operation invoked. Once the timer goes off, the operation will execute on the process's local copy of the queue and return a response if necessary.

Each process holds a local clock, a `toExecute` priority queue of operations sorted by timestamp, and a local copy of the shared FIFO Queue.

The algorithm includes a parameter X in the range $[0, d - \epsilon]$, which controls the trade off in the operation times of `peek()` and `enqueue()` operations. Small values of X result in slow `peek()` times and fast `enqueue()` times, while large values of X result in the opposite. X does not affect `dequeue()` times.

Operation invocations are represented as a tuple `<OP, arg, ts>`. `OP` represents the type of operation (`enqueue()`, `dequeue()`, or `peek()`), `arg` represents the argument passed by the user to the operation, if any, and `ts` represents the timestamp of the operation. The timestamp is itself represented by a tuple `<clock, i>` where `clock` is the local clock value of the process at the time of invocation, and `i` is a value that uniquely identifies the process.

When a process p invokes a `peek()` operation, it sets a timer for $d - X$ time and adds the operation to its local `toExecute` queue. Because `peek()` does not affect the state of the queue, no messages are sent. The timer value ensures that p has received all messages for operations that may occur before this `peek()`. When the timer goes off, p will generate and return an appropriate response to the user. An example of a `peek()` operation is shown in Figure 3.1.

When p invokes an `enqueue()` operation, it sets a timer for $X + \epsilon$ time, after which p will return an acknowledgement response to the user. Process p will also send messages to all other nodes containing the operation tuple, as well as simulate sending a message to itself by setting a

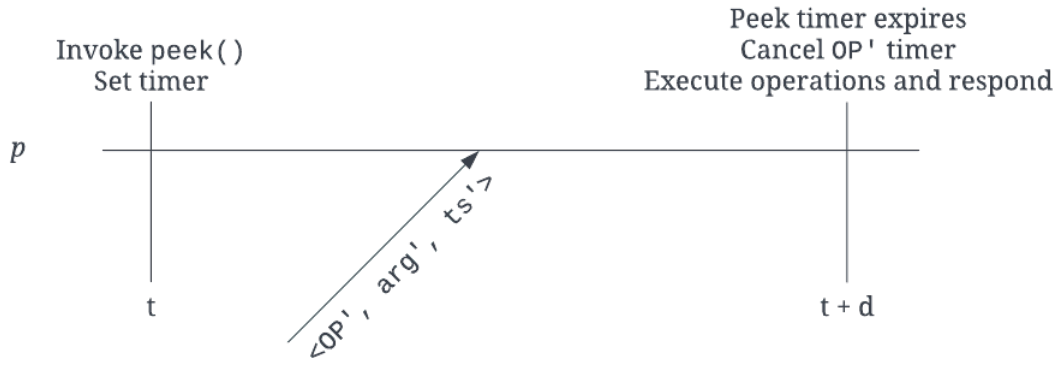

Figure 3.1: Sample execution timeline for $X = 0$ of an instance of `peek()` invoked by process p at time t. Process p receives a message containing a mutator operation *OP'* with timestamp *ts'* $< t$. Because *OP'* has a smaller timestamp than p's `peek()`, p locally executes *OP'* before the `peek()` and cancels its timer. This figure is a modified version of Figure 11 in [2] specialized for a FIFO Queue.

timer for $d - u$ time, the minimum message delay.

Similarly, when p invokes a `dequeue()` operation, p will send messages containing the operation tuple to all other nodes as well as simulate sending the message to itself. Unlike with `enqueue()`, p will not return a response to the user until after the operation is executed on its local copy of the queue.

When p receives a message or the timer for simulating a message send goes off, p will place the operation in its `toExecute` queue and set a timer for $u + \epsilon$ time. This final timer accounts for the difference in message delay and clock skew at each process. Once this timer goes off, the operation will be executed locally and a response will be generated and returned if the operation is a `dequeue()` Examples of an `enqueue()` operation and a `dequeue()` operation are shown in in Figures 3.2 and 3.3, respectively.

If a timer for an operation op not at the front of the `toExecute` queue goes off, all operations in front of op will be locally executed at that time in timestamp order and their timers will be cancelled.

The pseudocode of the algorithm specialized for a FIFO Queue is shown in Figure 3.4.

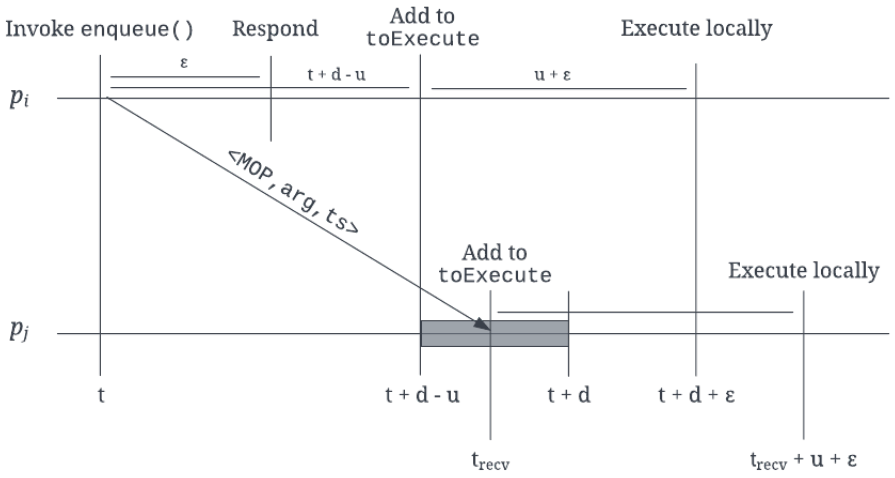

Figure 3.2: Sample execution timeline for $X = 0$ of an instance of `enqueue()` invoked by process p_i at time t. Process p_i sends a message to process p_j containing the operation. The gray area represents variance in the message delay and t_{recv} represents the time of receipt within the possible range. This figure is a modified version of Figure 12 in [2] specialized for a FIFO Queue.

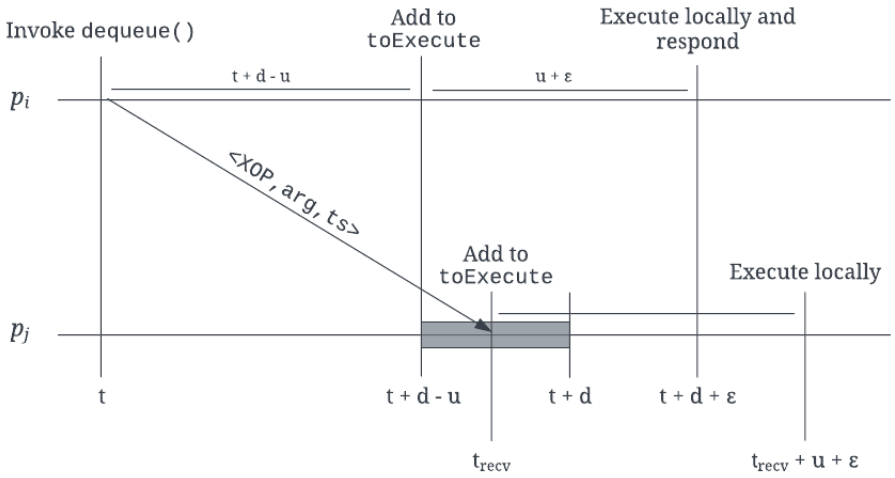

Figure 3.3: Sample execution timeline for $X = 0$ of an instance of `dequeue()` invoked by process p_i at time t. Process p_i sends a message to process p_j containing the operation. The gray area represents variance in the message delay and t_{recv} represents the time of receipt within the possible range. This figure is a modified version of Figure 13 in [2] specialized for a FIFO Queue.

```
Code for p_i:

local variables:
- clock, current time on p_i's local clock
- toExecute, priority queue of operations waiting to be locally executed;
  initially empty
- queue, FIFO queue, initially empty; supports peek, enqueue, dequeue
  with the standard semantics

HandleEvent invokePeek(-):   // pure accessor
  set timer(d, <peek,-,<clock,i>,respond)

HandleEvent ExpireTimer(<peek,-,ts>,respond):
  while ts >= timestamp of toExecute.min() do
    <OP',arg',ts'> := toExecute.extract_min()
    ret := executeLocally(OP',arg')
    cancel_timer(<OP',arg',ts'>,execute)
  endwhile
  // last iteration of while loop locally executed the peek
  generate return(ret)  // response to invoker of the peek

HandleEvent invokeEnqueue(arg):  // pure mutator
  set_timer(epsilon,<enqueue,arg,<clock,i>>,respond)
  set_timer(d-u,<enqueue,arg,<clock,i>,add)
  send(<enqueue,arg,<clock,i>>) to all other processes

HandleEvent ExpireTimer(<enqueue,arg,ts>,respond):
  generate ack  // response to invoker of enqueue

HandleEvent invokeDequeue(-):  // accessor/mutator
  set_timer(d-u,<dequeue,arg,<clock,i>,add)
  send(<dequeue,arg,<clock,i>>) to all other processes

HandleEvent ExpireTimer(<OP,arg,ts>,add) or Receive(<OP,arg,ts>):
  // OP is either enqueue or dequeue
  set_timer(u+epsilon,<OP,arg,ts>,execute)
  toExecute.add(<OP,arg,ts>)

HandleEvent ExpireTimer(<OP,arg,ts>,execute):  // OP is either enq or deq
  while ts >= timestamp of toExecute.min() do
    <OP',arg',ts'> := toExecute.extract_min()
    ret := executeLocally(OP',arg')
    cancel_timer(<OP',arg',ts'>,execute)
    if (ts' = <*,i> and OP' = dequeue) then  // this is a deq invoked by p_i
      generate return(ret)  // response to invoker of dequeue
    endif
  endwhile

function executeLocally(OP,arg):
  if OP = peek then return queue.peek()
  if OP = enqueue then return queue.enqueue(arg)  // returns "ack"
  if OP = dequeue then return queue.dequeue()
```

Figure 3.4: Specialization of generic algorithm from [2] for FIFO Queue with operations `enqueue()`, `dequeue()`, and `peek()` and $X = 0$.

4. SIMULATION

The simulation of the algorithm was implemented using Java. Rather than implementing the algorithm using distributed processes or threads, we developed a single-threaded program to simulate geographically separated nodes using a discrete event simulator. This allows us to precisely control all timing of aspects such as message delays and clock skews.

The program is split into the `Node` class, which holds a local copy of the shared object and other relevant data for each node in the system, and the `DistQueue` wrapper class for the entire system, which includes a `run()` method that acts as the discrete event simulator as well as other helper methods. There is also the `Op` helper class, which holds information on the events being simulated such as what event will occur, the timestamp, and the node that will handle the event.

4.1 The DistQueue Class

The bulk of the computation is done in the `DistQueue` class that acts as the model for our system. Its `run()` method contains the discrete event simulator. The `run()` method begins with a list of n `Node` objects, assigning each a unique integer to identify the node and a random clock skew in the range $[-(1/2)\epsilon, (1/2)\epsilon]$. It will then begin the simulation of the algorithm with a set of random operation invocations, one for each node in the system.

A discrete event simulator is used to model a system as a sequence of discrete events in time. The simulator will keep track of the system clock value and an event list [7]. For each event on the event list, the simulator updates the clock value, modifies the state of the system based on the event, and adds new events to the event list if any are generated. The clock value of the simulator is updated independent of real time, so one simulated time point can take more or less real time than another simulated time point depending on how much change occurs to the system at each time. A simulated time point can even be skipped entirely if there is no event that occurs at that time.

In our implementation, the simulator keeps a priority queue `events`, a list of events that

will occur, ordered by timestamp. Each event in the list is represented as an `Op` object. The simulator will continuously remove events from the top of the `event` queue and execute actions according to the data contained in the event object. New events are generated and placed on the `event` queue as preexisting events are handled. For example, if the next event is a node sending a message, an event for another node receiving the message will be generated and placed on the `event` queue.

For our algorithm, events can be categorized as an operation being invoked, a message being sent, a message being received or a timer expiring. The logic of how events are handled is controlled by a switch statement based on what type of event occurs and which operation was invoked on the shared queue for that event. For example, if the event is a timer expiring to execute a `peek()`, the simulator will execute the operation on the node's copy of the shared queue.

When a node needs to send a message, the message delay is randomly decided by the simulator. Before the system stabilizes, the simulator will check if the message should be lost based on a given probability. If the check fails and the message will eventually be received, the possible message delay is in the range $[d - u, 100]$. A delay of 100 was decided to be sufficiently large to simulate an arbitrarily long delay. The simulator chooses a delay by approximating a linear probability distribution. It will generate two random values within the given range and choose the minimum of the two values to be the delay. After the system has stabilized, the simulator will choose a message delay with uniform probability in the range $[d - u, d]$.

Finally, the invocation of operations is decided randomly by the simulator. Initially, the simulator will populate its `event` queue with the invocation of a random operation at a random time in the first ten time points for each node in the system with uniform probability for each operation and time. When an operation is completed by any node, the simulator will randomly select the next operation for the node to invoke between one and ten time units after the completion of the first operation, unless the current clock value is past a predetermined end time.

4.2 The Node Class

A `Node` object represents one of the geographically distributed nodes in the system. It holds an integer that uniquely identifies itself, a copy of the shared queue as a `LinkedList` object, and a `toExecute` priority queue that contains the operations to be executed on its copy of the queue ordered by timestamp.

For each of the three operations, the `Node` class contains a method that executes that operation and a method that places the operation on the `toExecute` queue. These can be called by the event simulator when handling a timer expiration or a receipt of a message.

There is also a method to cancel timers when an operation not at the front of the `toExecute` queue is executed. The node will execute all operations in front of the operation with the expired timer. It will return a list of these operations to the discrete event simulator, which will remove the timer events associated with those operations from its `event` queue.

Each `Node` object also keeps track of data that isn't necessary for the algorithm but is useful for logging purposes, such as the last mutator (`enqueue()` or `dequeue()`) that was executed on its local copy of the queue. The log produced by an execution is used to check the execution for linearizability.

4.3 Check for Linearizability

To find the grace period of an execution, we will be adapting a construction described in [2]. First, all `enqueue()` and `dequeue()` operations are placed in increasing order by timestamps. Each `peek()` operation executed by node p is inserted into the sequence directly after the last mutator p executed on its local copy before the `peek()` was executed. If there is no such mutator, the `peek()` is placed at the beginning of the sequence.

The check for linearizability starts after the system stabilization time, as the shared queue before the system stabilization time is expected to be affected by the poor behavior of the system. The checker finds the first `enqueue()` operation after the system stabilization time and discards all operations before it. It then finds the first accessor (`peek()` or `dequeue()`) that returns the same value enqueued by the first `enqueue()` operation and discards all other accessors before it. This

attempts to "clean out" any values enqueued during the poorly-behaved prefix of the execution.

This ordering of operations produced by the construction is then executed by a sequential FIFO Queue. If the value returned by a `dequeue()` or `peek()` operation on the sequential queue does not match the value returned by the corresponding operation in the construction, the ordering is considered to be not linearizable. In this case, the checker finds the second `enqueue()` operation after the system stabilization time and its first matching accessor and executes the ordering in a sequential queue. If this also fails, the checker continues to find the next `enqueue()` and its matching accessor until the linearizability check passes. The grace period of any execution is measured as the time between the system stabilization time and the timestamp of the first `enqueue()` operation in the linearizable ordering.

The downside to this check is that only one ordering of the operations is considered. It is possible that a linearizable ordering does exist for an execution that fails this check. Our method for checking linearizability may perhaps be too conservative in this way. Results may show a longer grace period for some executions than is necessary, when a different ordering may result in a shorter grace period.

5. EXPERIMENTAL RESULTS

5.1 Experimental Design

We changed and tested five parameters in our simulation:

- The proportion d/u in the range $[2, 10]$,
- the number of nodes in the system (n), in the range $[5, 100]$,
- the probability of lost messages in the range $[0, 0.95]$, and
- the time until the system stabilizes, in the range $[50, 500]$.

Specifically, the value of u was fixed at 2 and d was changed to fit the desired proportion.

Only one parameter was changed at a time. All other parameters, including X for all trials, remained at a fixed value, which were the following:

- 2 for the proportion of d/u,
- 50 for the number of nodes in the system (n),
- 0.1 for the probability of lost messages,
- 250 time units until the system stabilizes, and
- 0 for the value of X.

For each value of any parameter, we ran 500 trials using a driver program.

5.2 Results

Here we will show the results of our experiments and provide an intuition for why these results occurred. These explanations are only intuitions and more research is necessary to make any conclusive statements about the observed behavior.

First, as the proportion of d/u increases, the length of the grace period increases as well, as shown in Figure 5.1. This proportion only applies to messages sent after the system stabilizes, so any affect it has on the grace period occurs immediately after the system stabilization time. As message delays have the potential to be longer, more time is needed to "clean out" the values that were enqueued during the poorly-behaved prefix of the execution from each node's local queue.

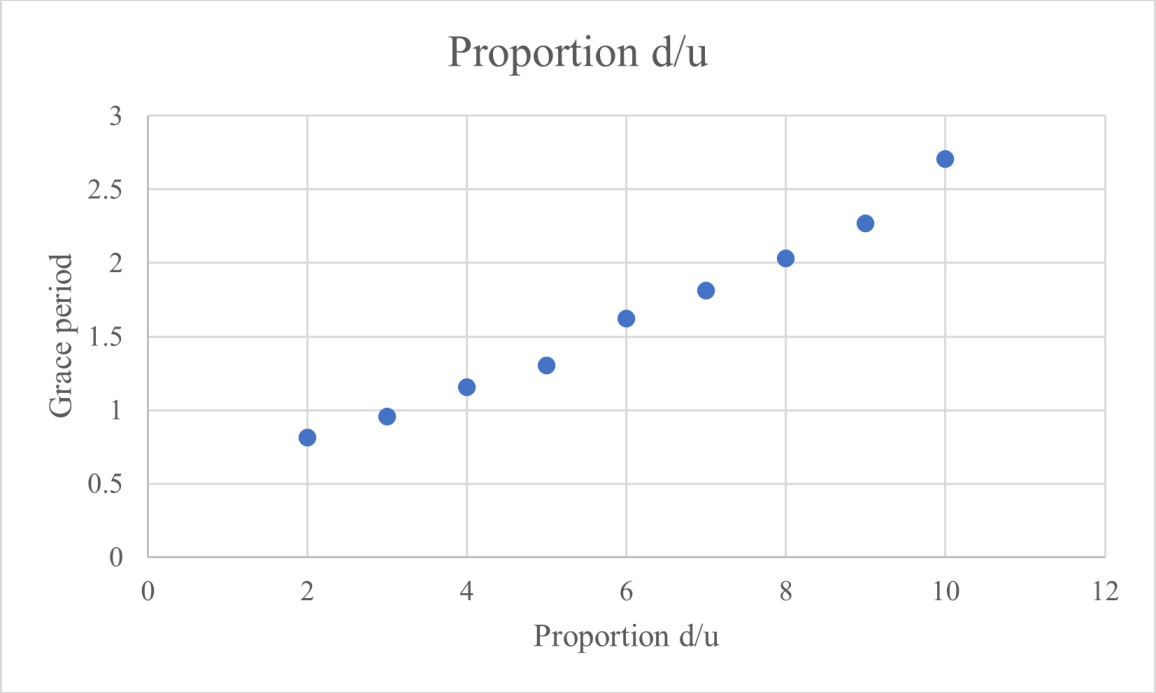

Figure 5.1: Average grace period for change in the proportion of d/u.

Interestingly, the length of the grace period decreases and flattens out as the number of nodes increases, as shown in Figure 5.2. This is likely because as there are more nodes to invoke operations, `enqueue()` operations occur more often. There are then earlier chances for the linearizability checker to begin its check, as compared to sparser `enqueue()` operations in executions with fewer nodes. It should be noted that small grace periods are possible with few nodes. For example, our set of executions for 5 nodes had an average grace period of 10.336 but a standard deviation of 11.496. The long grace period is likely primarily caused by there being few nodes to call `enqueue()` operations, and the linearizability checker is not able to check orderings that start earlier in the execution.

The grace period is generally consistent as the probability for lost messages increases, but does trend downwards slightly, as shown in figure 5.3. Similar to the proportion of d/u, the probability of lost messages is only applicable when the system is poorly behaved. Because the grace period is measured as time after the system stabilizes, it makes sense that it is mostly unaffected by the change in probability of lost messages.

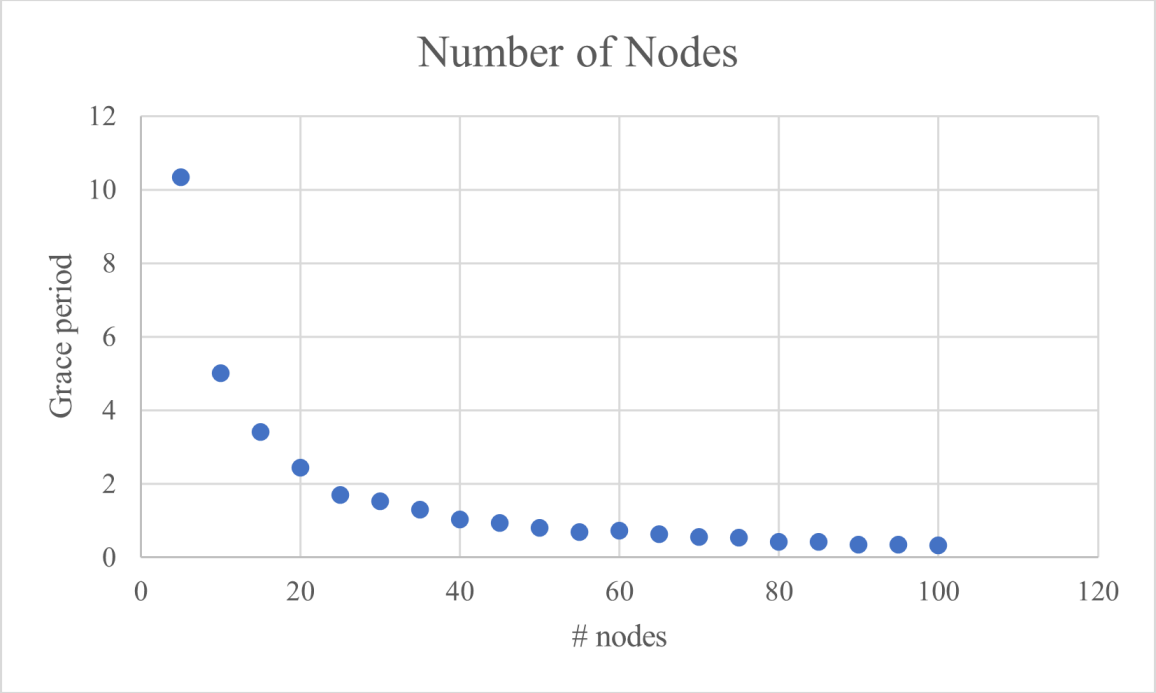

Figure 5.2: Average grace period for change in the number of nodes in system.

However, the downward trend is surprising. We would expect to see the length of the grace period increase as more messages are lost and the system take more time to recover. Perhaps as more messages are lost, there are fewer values that need to be cleaned out of any given node's local copy of the queue, and the grace period becomes smaller.

Finally, as the system takes longer to stabilize, the length of the grace period increases, as shown in Figure 5.4. This is likely because there is more time for operations to be invoked while the system experiences poor behavior, and more values need to be cleaned out of the local queues once the system stabilizes.

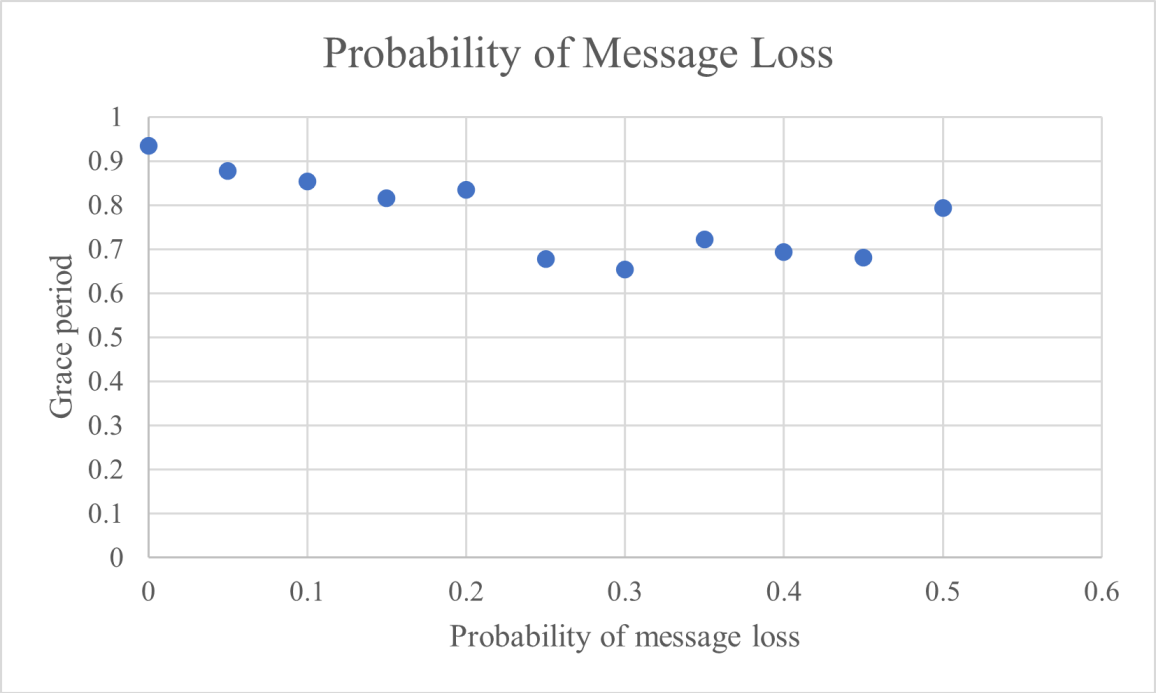

Figure 5.3: Average grace period for change in the probability of lost messages.

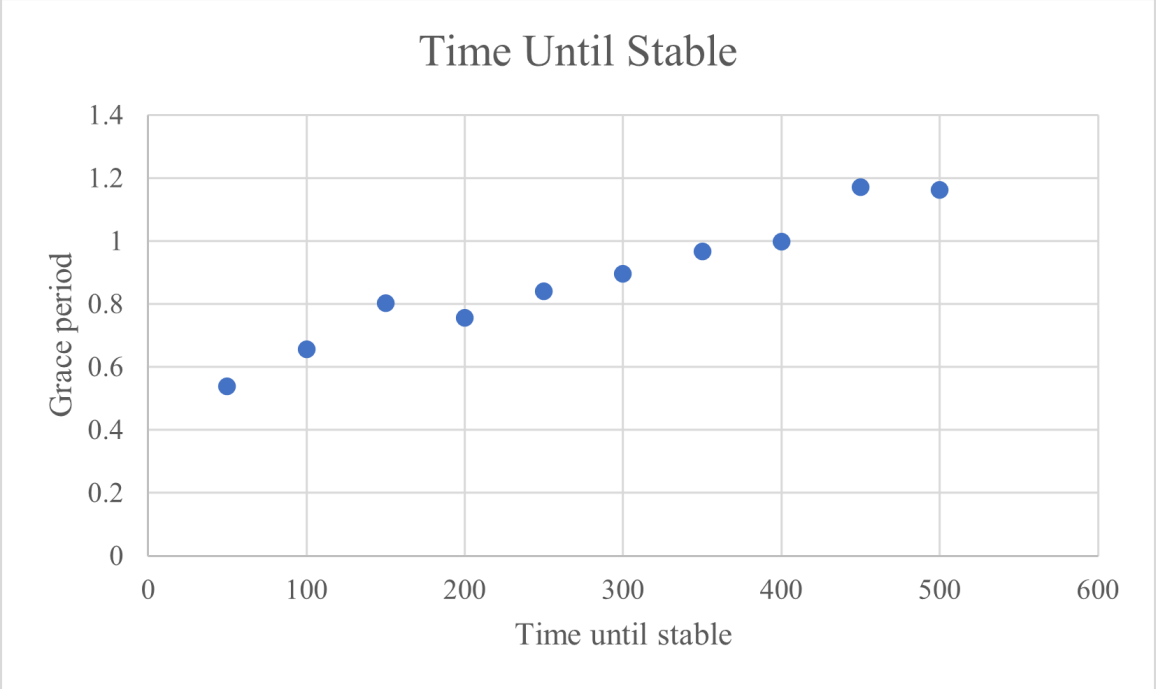

Figure 5.4: Average grace period for change in the system stabilization time.

6. CONCLUSION AND FUTURE WORK

In this thesis, we extended the algorithm given in [2] to implement a shared generic data structure in a synchronous message-passing system to a FIFO Queue in a system that initially experiences lost messages and arbitrarily long message delays but eventually becomes well-behaved with bounded message delays. We examined the algorithm's behavior by experimentally measuring the grace period of executions with different values for several parameters. The grace period for an execution was defined as the time between the system stabilization time and the timestamp of the first `enqueue()` operation in the earliest linearizable subset of operations in the execution.

We found that to minimize the grace period, the proportion of d/u and the time until the system stabilizes should be low, and the number of nodes in the system and the probability of lost messages should be high. It would be interesting to see whether a combination of appropriate values for these parameters will keep the grace period low, or whether the parameters interact in a way that will unexpectedly increase the grace period.

Further experiments on our simulation of a shared FIFO Queue could investigate the behavior for different values of X, as this was omitted from our trials, as well as other values of u when looking at the proportion of d/u. Experiments can also use systems with different behavior, such as including crash failures or different probability distributions for message delays before the system stabilizes.

Additionally, similar experiments can be performed for implementations of other shared data objects, such as a Stack or a Binary Tree. If these experiments produce different results than those produced when focusing on a Queue, there may be something to learn about the algorithm by figuring out why this is the case. Ultimately, we would like to find upper and lower bounds on the grace period for a generic shared data object through a theoretical proof. Based on this, improvements could possibly be made to the generic algorithm so that the upper bound on the grace period could be lowered.

REFERENCES

[1] H. Attiya, A. Bar-Noy, and D. Dolev, "Sharing memory robustly in message-passing systems," *J. ACM*, vol. 42, no. 1, pp. 124–142, 1995.

[2] 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.

[3] 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.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg, "An algorithm for replicated objects with efficient reads," in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016* (G. Giakkoupis, ed.), pp. 325–334, ACM, 2016.

[5] 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.

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

[7] J. Misra, "Distributed discrete-event simulation," *ACM Comput. Surv.*, vol. 18, no. 1, pp. 39–65, 1986.

[8] Y. A. Liu, S. D. Stoller, B. Lin, and M. Gorbovitski, "From clarity to efficiency for distributed algorithms," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012* (G. T. Leavens and M. B. Dwyer, eds.), pp. 395–410, ACM, 2012.

APPENDIX A: IMPLEMENTATION IN DISTALGO

The original simulation of the algorithm was done using DistAlgo. DistAlgo is a programming language built on top of Python that provides an efficient implementation of distributed algorithms [8]. Nodes are contained in separate threads and are able to pass messages between each other. Because of this provided framework to pass messages, implementing the algorithm in DistAlgo seemed like the best approach for our algorithm.

The problem with this approach arises with the timing elements of the algorithm. First, because all threads would be located on one machine, messages would be sent and delivered instantaneously. Second, DistAlgo does not provide any built-in mechanism for setting timers.

To get around this, we added a discrete event simulator to the program that would control message delays and timers, similar to our use of the simulator in the final Java program. It would not control the timing of the actions the nodes take once a timer expires or a message is received, only when those actions started.

What resulted was a combination of real time and simulated time that kept the algorithm from running correctly. The simulator would send a message to a node saying that a timer had expired or a message from another node was delivered. Because the simulator was contained in a thread separate from the other nodes, it would continue to the next simulated time point before the nodes had completed their actions at the current time point.

Instead of attempting to find potentially complex ways to solve these problems in DistAlgo, we switched our implementation to a sequential program in Java with timing that was entirely controlled by a discrete event simulator.

APPENDIX B: CODE FOR SIMULATION OF SHARED FIFO QUEUE

```
import java.util.*;
import java.time.*;
import java.io.*;

enum OpType{
    PEEK,
    PEEK_EXEC,
    ENQUEUE,
    ENQUEUE_RESPOND,
    ENQUEUE_ADD,
    ENQUEUE_SEND,
    ENQUEUE_EXEC,
    DEQUEUE,
    DEQUEUE_ADD,
    DEQUEUE_SEND,
    DEQUEUE_EXEC
}

class Op implements Comparable<Op>{
    public OpType opCall;
    public int arg;
    public int ts;
    public int calledBy;
    // For sending message
    public int sendTo;
    public int delay;
    public int timer;

    public Op(OpType opCall, int arg, int ts, int calledBy){
        this.opCall = opCall;
        this.arg = arg;
        this.ts = ts;
        this.calledBy = calledBy;
        sendTo = -1;
        delay = -1;
        timer = -1;
    }
```

```
// Continued from previous page
public Op(OpType opCall, int arg, int ts, int calledBy, int sendTo, int delay){
    this.opCall = opCall;
    this.arg = arg;
    this.ts = ts;
    this.calledBy = calledBy;
    this.sendTo = sendTo;
    this.delay = delay;
    timer = -1;
}

public Op(OpType opCall, int arg, int ts, int calledBy, int timer){
    this.opCall = opCall;
    this.arg = arg;
    this.ts = ts;
    this.calledBy = calledBy;
    this.timer = timer;
    sendTo = -1;
    delay = -1;
}

public Op(OpType opCall, int arg, int ts, int calledBy, int timer, int sendTo, int delay){
    this.opCall = opCall;
    this.arg = arg;
    this.ts = ts;
    this.calledBy = calledBy;
    this.timer = timer;
    this.sendTo = sendTo;
    this.delay = delay;
}

public int compareTo(Op o){
    int diff = ts - o.ts;
    if(diff == 0){
        diff = calledBy - o.calledBy;
    }
    return diff;
}
```

```
    // Continued from previous page
    public boolean equals(Object obj){
        Op o = (Op) obj;
        return opCall == o.opCall &&
                arg == o.arg &&
                ts == o.ts &&
                calledBy == o.calledBy &&
                sendTo == o.sendTo;
    }

    // for debugging
    public String toString(){
        return "node: " + calledBy +"\n" +
                "opType: " + opCall + "\n" +
                "arg: " + arg + "\n" +
                "ts: " + ts + "\n" +
                "sendto: " + sendTo;
    }
}

class Node{
    private PriorityQueue<Op> toExecute;
    private LinkedList<Integer> data;
    private int id;
    private int skew; // clock skew from global clock
    private FileOutputStream file;
    private String lastMut;
    private LinChecker checker;

    public Node(int id, int skew, FileOutputStream file, LinChecker checker){
        this.id = id;
        this.skew = skew;
        this.file = file;
        this.checker = checker;
        toExecute = new PriorityQueue<>();
        data = new LinkedList<>();
        lastMut = "";
    }
```

```
// Continued from previous page
public int id(){
    return id;
}

public int skew(){
    return skew;
}

public void peek(int clock) throws IOException{
    if(data.size() != 0){
        file.write(("Peek " + data.peek() + " by node " + id + " at time " + clock + "\n").getBytes());
        checker.addToLog("end 1 " + id + " " + data.peek() + " " + (clock + skew));
        // log format: start/end opType nodeID value globalClock
        checker.addToLog(lastMut);
        file.write(lastMut.getBytes());
        file.write('\n');
    }
    else{
        checker.addToLog("end 1 " + id + " -1 " + clock);
        checker.addToLog(lastMut);
        file.write(("Peek empty by node " + id + " at time " + clock + "\n").getBytes());
        file.write(lastMut.getBytes());
        file.write('\n');
    }
}

public void peekSend(int ts, int d, int x){
    toExecute.add(new Op(OpType.PEEK_EXEC, 0, ts, id, ts + d - x));
}

public void enqueue(int val, int calledBy, int clock) throws IOException{
    if(calledBy == id){
        file.write(("Enqueue " + val + " by node " + calledBy + " at time " + clock + "\n").getBytes());
        checker.addToLog("end 2 " + id + " " + val + " " + (clock + skew));
    }
    data.add(val);
}
```

```
// Continued from previous page
public void enqueueSend(int val, int ts, int caller, int delay, int timer){
    toExecute.add(new Op(OpType.ENQUEUE_EXEC, val, ts, caller, timer, id, delay));
}

public int dequeue(int calledBy, int clock) throws IOException{
    if(data.size() != 0){
        int ret = data.pollFirst();
        if(calledBy == id){
            file.write(("Dequeue " + ret + " by node " + calledBy + " at time "
            + clock + "\n").getBytes());
            checker.addToLog("end 3 " + id + " " + ret + " " + (clock + skew));
        }

        return ret;
    }
    else{
        if(calledBy == id){
            checker.addToLog("end 3 " + id  + " -1 " + clock);
            file.write(("Dequeue empty by node " + calledBy + " at time " + clock + "\n").getBytes());
        }
        return -1;
    }
}

public void dequeueSend(int ts, int caller, int delay, int timer){
    toExecute.add(new Op(OpType.DEQUEUE_EXEC, 0, ts, caller, timer, id, delay));
}

public ArrayList<Op> cancel(Op event, int ts) throws IOException{
    ArrayList<Op> cancelled = new ArrayList<>();

    Op called = find(event);
    if(called == null){
        return cancelled;
    }
```

```
        // Continued from previous page
        while(toExecute.size() != 0 && toExecute.peek().compareTo(called) <= 0){
            Op exec = toExecute.poll();
            switch(exec.opCall){
                case PEEK_EXEC:
                    peek(ts);
                    break;
                case ENQUEUE_EXEC:
                    enqueue(exec.arg, exec.calledBy, ts);
                    lastMut = "2 " +  exec.arg + " " + exec.ts + " " + exec.calledBy;
                    break;
                case DEQUEUE_EXEC:
                    int d = dequeue(exec.calledBy, ts);
                    lastMut = "3 " + d + " " + exec.ts + " " + exec.calledBy;
                    break;
            }

            if(exec.calledBy != id){
                exec.sendTo = id;
            }

            cancelled.add(exec);
        }

        cancelled.remove(cancelled.size() - 1);
        return cancelled;
    }

    private Op find(Op event){
        for(Op op: toExecute){
            if(event.opCall == op.opCall && event.arg == op.arg && event.ts >= op.ts
            && event.calledBy == op.calledBy ){
                return op;
            }
        }
        return null;
    }
}
```

```
// Continued from previous page
public class DistQueue{
    private static PriorityQueue<Op> events = new PriorityQueue<Op>();
    private int d;
    private int badDelay;
    private int u;
    private int x;
    private int eps;
    private int nextVal;
    private int stable;
    private double lossP;
    private int stop;
    private int n;
    private LinChecker checker;

    public DistQueue(int d, int u, int x, int stable, int stop, double lossP, int n){
        this.d = d;
        this.u = u;
        this.stable = stable;
        this.stop = stop;
        this.lossP = lossP;
        this.n = n;

        badDelay = 100;

        eps = (int) ((1 - (1.0 / n)) * u);
        nextVal = 0;
        checker = new LinChecker(stable);

        if(x == 1){
            x = d - eps;
        }
        else{
            x = 0;
        }
    }
```

```
// Continued from previous page
public int run() throws Exception{
    Instant now = Clock.systemDefaultZone().instant();
    String path = "data/" + now.toString().substring(0, 18).replace(':', '-') + ".txt";
    FileOutputStream file = new FileOutputStream(path);

    int clock = 0;
    Node[] nodes = new Node[n];
    for(int i = 0; i < n; i++){
        int skew = (int) (Math.random() * (eps + 1)) - (eps * 1/2);
        nodes[i] = new Node(i, skew, file, checker);
        file.write(("Node " + i + " has skew " + skew + "\n").getBytes());
        scheduleNext(i, -1);
    }
    file.write(("\n").getBytes());

    while(events.size() != 0){
        Op event = events.poll();
        clock = event.ts;

        switch(event.opCall){
            case PEEK: // Invoke peek(), set timer to execute
                file.write(("Peek started by node " + event.calledBy + " at time "
                + clock + "\n").getBytes());
                checker.addToLog("start 1 " + event.calledBy + " 0 " +
                (clock + nodes[event.calledBy].skew()));
                events.add(new Op(OpType.PEEK_EXEC, 0, clock + d - x, event.calledBy));
                nodes[event.calledBy].peekSend(clock, d, x);
                break;
            case PEEK_EXEC: // Execute peek()
                ArrayList<Op> cancelled = nodes[event.calledBy].cancel(event, clock);
                updateCancels(cancelled, event.calledBy);
                scheduleNext(event.calledBy, clock);
                break;
```

```
// Continued from previous page
case ENQUEUE: // Invoke enqueue(), set response timer, send to other nodes
    file.write(("Enqueue started by node " + event.calledBy + " at time "
    + clock + "\n").getBytes());
    checker.addToLog("start 2 " + event.calledBy + " " + event.arg +
    " " + (clock + nodes[event.calledBy].skew()));
    events.add(new Op(OpType.ENQUEUE_RESPOND, event.arg, clock + x + eps, event.calledBy));
    events.add(new Op(OpType.ENQUEUE_ADD, event.arg, clock + d - u, event.calledBy));
    for(Node node: nodes){
        if(clock >= stable || Math.random() > lossP){
        // chance message is lost, if message is not lost
            if(node.id() != event.calledBy){
                int delay = getRandomDelay(clock);
                events.add(new Op(OpType.ENQUEUE_SEND, event.arg,
                clock + delay, event.calledBy, node.id(), delay));
            }
        }
    }
    break;
case ENQUEUE_SEND: // Receive enqueue message
    Node node = nodes[event.sendTo];
    node.enqueueSend(event.arg, clock - event.delay +
    nodes[event.calledBy].skew(), event.calledBy, event.delay, clock + u + eps);
    events.add(new Op(OpType.ENQUEUE_EXEC, event.arg, clock + u + eps,
    event.calledBy, node.id(), event.delay));
    break;
case ENQUEUE_RESPOND: // Enqueue response to user
    file.write(("Enqueue ack by node " + event.calledBy + " at time " +
    clock + "\n").getBytes());
    break;
case ENQUEUE_ADD: // Adds enqueue to toExecute queue
    events.add(new Op(OpType.ENQUEUE_EXEC, event.arg, clock + u + eps,
    event.calledBy, event.calledBy, -1));
    nodes[event.calledBy].enqueueSend(event.arg, clock +
    nodes[event.calledBy].skew() - d + u, event.calledBy, 0, clock + u + eps);
    break;
```

```
// Continued from previous page
case ENQUEUE_EXEC: // executes enqueue()
    cancelled = nodes[event.sendTo].cancel(event, clock);
    updateCancels(cancelled, event.calledBy);
    if(event.sendTo == event.calledBy){
        scheduleNext(event.calledBy, clock);
    }
    break;

case DEQUEUE: // Invoke dequeue(), send to other nodes
    file.write(("Dequeue started by node " + event.calledBy + " at time " +
    clock + "\n").getBytes());
    checker.addToLog("start 3 " + event.calledBy + " 0 " + (clock +
    nodes[event.calledBy].skew()));
    events.add(new Op(OpType.DEQUEUE_ADD, 0, clock + d - u, event.calledBy));
    for(Node nodeLoop: nodes){
        if(clock >= stable || Math.random() > lossP){ // If message is not lost
            if(nodeLoop.id() != event.calledBy){
                int delay = getRandomDelay(clock);
                events.add(new Op(OpType.DEQUEUE_SEND, event.arg,
                clock + delay, event.calledBy, nodeLoop.id(), delay));
            }
        }
    }
    break;
case DEQUEUE_SEND: // Receive dequeue message
    node = nodes[event.sendTo];
    node.dequeueSend(clock - event.delay + nodes[event.calledBy].skew(), event.calledBy,
    event.delay, clock + u + eps);
    events.add(new Op(OpType.DEQUEUE_EXEC, event.arg, clock + u + eps, event.calledBy,
    node.id(), event.delay));
    break;
case DEQUEUE_ADD: // Adds dequeue() to toExecute queue
    events.add(new Op(OpType.DEQUEUE_EXEC, 0, clock + u + eps,
    event.calledBy, event.calledBy, -1));
    nodes[event.calledBy].dequeueSend(clock + nodes[event.calledBy].skew() - d + u,
    event.calledBy, 0, clock + u + eps);
    break;
```

```
            // Continued from previous page
            case DEQUEUE_EXEC: // Executes dequeue()
                cancelled = nodes[event.sendTo].cancel(event, clock);
                updateCancels(cancelled, event.calledBy);
                if(event.sendTo == event.calledBy){
                    scheduleNext(event.calledBy, clock);
                }
                break;
        }
    }

    file.close();

    return checker.check();
}

private void updateCancels(ArrayList<Op> cancelled, int nodeId){
    for(Op op: cancelled){
        Op cancel = new Op(op.opCall, op.arg, op.timer, op.calledBy, op.sendTo, 0);
        events.remove(cancel);
    }
}

private int getRandomDelay(int clock){
    if(clock < stable){
        // Approximately linear distribution
        int a = (int) (Math.random() * (badDelay - (d - u) + 1)) + (d - u);
        int b = (int) (Math.random() * (badDelay - (d - u) + 1)) + (d - u);
        int delay = Math.min(a, b);
        if(delay + clock >= stable){
            // If the message will be delivered after the stabilization time, the delay is changed
            // to be delivered at stabilization time or within the possible delay range
            delay = stable - clock;
            if(stable - clock < d - u){
                delay = (int) (Math.random() * (u + 1)) + (d - u);
            }
        }
        return delay;
    }
    return (int) (Math.random() * (u + 1)) + (d - u);
}
```

```
    // Continued from previous page
    private void scheduleNext(int node, int clock){
        if(clock > stop){
            return;
        }
        int opType = (int) (Math.random() * 3);
        int minWait = 1;
        int maxWait = 10;
        int wait = (int) (Math.random() * (maxWait - minWait + 1)) + minWait;

        if(opType == 0){
            events.add(new Op(OpType.PEEK, 0, clock + wait, node));
        }
        else if(opType == 1){
            events.add(new Op(OpType.ENQUEUE, nextVal++, clock + wait, node));
        }
        else{
            events.add(new Op(OpType.DEQUEUE, 0, clock + wait, node));
        }
    }
}
```

Figure B.1: Code for simulating shared FIFO Queue.

APPENDIX C: CODE FOR CHECKING LINEARIZABILITY

```
import java.util.*;

public class LinChecker {
    private ArrayList<String> log;
    private ArrayList<int[]> pi;
    private int stable;

    public LinChecker(int stable) {
        this.stable = stable;
        log = new ArrayList<>();
        pi = new ArrayList<>();
    }

    public void addToLog(String line) {
        log.add(line);
    }

    public int check() throws Exception {
        ArrayList<int[]> peeks = new ArrayList<>();
        // Inner array contains {start time, end time, node, operation type, argument/returned}
        // Operation type: 1 = peek, 2 = enqueue, 3 = dequeue
        Hashtable<Integer, ArrayList<Integer>> unfinished = new Hashtable<>();
        // Key = node ID
        // ArrayList contains {operation type, start time}

        // -------- LOG SCRAPER --------
        // Reads log from DistQueue.java
        int l = 0;
        for (int i = 0; i < log.size(); i++) {
            String line = log.get(i);
            String[] tokens = line.split(" ");
            // Log format: start/end opType nodeID value globalClock
```

```
// Continued from previous page
if (tokens[0].equals("start")) {
    ArrayList<Integer> arr = new ArrayList<>();
    arr.add(Integer.parseInt(tokens[1])); // operation type
    arr.add(Integer.parseInt(tokens[4])); // clock

    unfinished.put(Integer.parseInt(tokens[2]), arr);
}
else {
    ArrayList<Integer> started = unfinished.get(Integer.parseInt(tokens[2]));

    int opType = Integer.parseInt(tokens[1]);
    if (started.get(0) != opType) {
        throw new Exception("Something didn't match");
    }

    int[] arr = new int[5];
    arr[0] = started.get(1);                // start time
    arr[1] = Integer.parseInt(tokens[4]);   // end time
    arr[2] = Integer.parseInt(tokens[2]);   // node
    arr[3] = started.get(0);                // operation type
    arr[4] = Integer.parseInt(tokens[3]);   // argument

    if (arr[3] == 1) {
        int[] peekArr = new int[9];
        for (int j = 0; j < 5; j++) {
            peekArr[j] = arr[j];
        }

        line = log.get(++i);
        // Get last mutator
        // Enqueue/dequeue arg

        if (line.equals("")) {
            peekArr[5] = -1;
            peekArr[6] = -1;
            peekArr[7] = -1;
            peekArr[8] = -1;
        }
```

```
            // Continued from previous page
            else {
                String[] lastMut = line.split(" ");

                peekArr[5] = Integer.parseInt(lastMut[0]); // last mutator type
                peekArr[6] = Integer.parseInt(lastMut[1]); // last mutator argument
                peekArr[7] = Integer.parseInt(lastMut[2]); // last mutator timestamp
                peekArr[8] = Integer.parseInt(lastMut[3]); // node that called last mutator
                peeks.add(peekArr);
            }
        }
        else {
            pi.add(arr);
        }
    }
}

// -------- CONSTRUCTOR --------
// Orders operations according to construction
// Throws away operations before stable time
// Finds first enqueue value and throws away all accessors until one that returns the first enqueued value

Collections.sort(pi, new Comparator<int[]>() {
    public int compare(int[] a, int[] b) {
        if (a[0] == b[0]) {
            return a[2] - b[2];
        }
        else {
            return a[0] - b[0];
        }
    }
});

for (int[] peekOp : peeks) {
    int lastMut = peekOp[5];
    int lastArg = peekOp[6];
    int mutTS = peekOp[7];
    int mutCaller = peekOp[8];
```

```
    // Continued from previous page
    if (lastMut == -1) {
        pi.add(0, peekOp);
        continue;
    }

    for (int i = 0; i < pi.size(); i++) {
        if (pi.get(i)[3] == lastMut && pi.get(i)[4] == lastArg && pi.get(i)[0] == mutTS &&
        pi.get(i)[2] == mutCaller) {
            // If peek's last mutator matches current operation seen in pi
            pi.add(i + 1, peekOp);
            break;
        }
    }
}

for(int i = 0; i < pi.size(); i++){
    int[] op = pi.get(i);
    if(op[0] < stable){
        pi.remove(i);
        i--;
    }
}

// -------- CHECKER --------
// Checks ordering from above if it is linearized

boolean found = false;
while(pi.size() > 0 && !found) {
    LinkedList<Integer> queue = new LinkedList<>();
    nextEnq(); // Finds first enqueue, removes all accessors until one matches the first enqueue
    for (int[] op : pi) {
        found = true;
        switch (op[3]) {
            case 1: // peek
                int peeked = -1;
                if (queue.size() != 0) {
                    peeked = queue.peek();
                }
```

```
                    // Continued from previous page
                    if (peeked != op[4]) {
                        found = false;
                        break;
                    }
                    break;
                case 2: // enqueue
                    queue.add(op[4]);
                    break;
                case 3:
                    int dq = -1;
                    if (queue.size() != 0) {
                        dq = queue.poll();
                    }
                    if (dq != op[4]) {
                        found = false;
                        break;
                    }
            }
        }

        if(found) {
            return pi.get(0)[0] - stable; // start time of first enqueue when linearizable
        }
    }

    // Execution was never linearizable
    throw new Exception("Too short");
}

private void nextEnq(){
    if(pi.size() == 0){
        return;
    }

    pi.remove(0);
    int firstEnq = -1;
```

```
        // Continued from previous page
        for(int i = 0; i < pi.size(); i++){
            int[] op = pi.get(i);
            if(op[3] == 2){
                firstEnq = op[4];
                break;
            }
            else{
                pi.remove(i);
                i--;
            }
        }

        for(int i = 0; i < pi.size(); i++){
            int[] op = pi.get(i);
            if(op[3] != 2 && op[4] != firstEnq){
                pi.remove(i);
                i--;
            }
            else if(op[3] != 2 && op[4] == firstEnq){
                break;
            }
        }
    }

    // for debugging
    private void printPi() {
        for (int i = 0; i < pi.size(); i++) {
            System.out.println("Start " + pi.get(i)[0] + " End " + pi.get(i)[1] + " Node " +
            pi.get(i)[2] + " Op type " + pi.get(i)[3] + " Arg/return " + pi.get(i)[4]);
        }
    }
}
```

Figure C.1: Code for checking linearizability of an execution.

APPENDIX D: CODE FOR RUNNING EXPERIMENTS

```
import java.io.*;
import java.util.*;

public class Driver {
    // Driver code for experiments
    public static void main(String[] args) throws Exception{
        FileOutputStream file = new FileOutputStream("data/experiments.csv");

        int d = 4;
        int u = 2;
        int x = 0;
        int stable = 250;
        int stop = 500;
        double lossP = .1;
        int n = 50;

        file.write(("Grace period needed until linearizable\n").getBytes());

        // Proportion d/u
        // 2, 3, 4, ..., 10
        System.out.println("Proprtion d/u");
        file.write(("Proportion d/u\n").getBytes());
        for(int i = 2; i <= 10; i++){
            d = u * i;
            System.out.println(i);
            file.write((i + ",").getBytes());

            ArrayList<Integer> data = new ArrayList<Integer>();
            int total = 0;
            for(int j = 0; j < 500; j++){
                DistQueue q = new DistQueue(d, u, x, stable, stop, lossP, n);
                int time = q.run();
                total += time;
                data.add(time);
            }
```

```
    // Continued from previous page
    double avg = total / 500.0;
    double sd = stdDev(data, avg);
    file.write((avg + "," + sd + "\n").getBytes());
}
file.write('\n');
d = 4;

// Time until stable
// 50, 100, 150, ... 500
System.out.println("Time until stable");
file.write(("Time until stable\n").getBytes());
for(int i = 50; i <= 500; i += 50){
    stable = i;
    stop = i + 250;
    System.out.println(i);
    file.write((i + ",").getBytes());

    ArrayList<Integer> data = new ArrayList<Integer>();
    int total = 0;
    for(int j = 0; j < 500; j++){
        DistQueue q = new DistQueue(d, u, x, stable, stop, lossP, n);
        int time = q.run();
        total += time;
        data.add(time);
    }
    double avg = total / 500.0;
    double sd = stdDev(data, avg);
    file.write((avg + "," + sd + "\n").getBytes());
}
file.write('\n');
stable = 250;
stop = 500;
```

```
// Continued from previous page
// Probability of message loss
// 0, .05, .1, .15, ..., .5
System.out.println("Probability of message loss");
file.write(("Probability of message loss\n").getBytes());
for(double i = 0; i <= .5; i += .05){
    lossP = i;
    System.out.println(i);
    file.write((i + ",").getBytes());

    ArrayList<Integer> data = new ArrayList<Integer>();
    int total = 0;
    for(int j = 0; j < 500; j++){
        DistQueue q = new DistQueue(d, u, x, stable, stop, lossP, n);
        int time = q.run();
        total += time;
        data.add(time);
    }
    double avg = total / 500.0;
    double sd = stdDev(data, avg);
    file.write((avg + "," + sd + "\n").getBytes());
}
file.write('\n');
lossP = .1;

// Number of nodes
// 5, 10, 15, ..., 100
System.out.println("Number of nodes");
file.write(("Number of nodes\n").getBytes());
for(int i = 5; i <= 100; i += 5){
    n = i;
    System.out.println(i);
    file.write((i + ",").getBytes());

    ArrayList<Integer> data = new ArrayList<Integer>();
    int total = 0;
```

```
    // Continued from previous page
    for(int j = 0; j < 500; j++){
        DistQueue q = new DistQueue(d, u, x, stable, stop, lossP, n);
        int time = q.run();
        total += time;
        data.add(time);
    }
    double avg = total / 500.0;
    double sd = stdDev(data, avg);
    file.write((avg + "," + sd + "\n").getBytes());
}
file.write('\n');
n = 50;

// X
// 0, d - eps
System.out.println("X");
file.write(("X\n").getBytes());
System.out.println("0");
file.write(("0,").getBytes());
x = 0;
ArrayList<Integer> data = new ArrayList<Integer>();
int total = 0;
for(int j = 0; j < 500; j++){
    DistQueue q = new DistQueue(d, u, x, stable, stop, lossP, n);
    int time = q.run();
    total += time;
    data.add(time);
}
double avg = total / 500.0;
double sd = stdDev(data, avg);
file.write((avg + "," + sd + "\n").getBytes());

System.out.println("d - epsilon");
file.write(("d - epsilon,").getBytes());
x = 1;
total = 0;
```

```
        // Continued from previous page
        data.clear();
        for(int j = 0; j < 500; j++){
            DistQueue q = new DistQueue(d, u, x, stable, stop, lossP, n);
            int time = q.run();
            total += time;
            data.add(time);
        }
        avg = total / 500.0;
        sd = stdDev(data, avg);
        file.write((avg + "," + sd + "\n").getBytes());

        file.write('\n');
        x = 0;
    }

    private static double stdDev(ArrayList<Integer> data, double avg){
        double sd = 0;
        for(int n: data){
            sd += Math.pow(n - avg, 2);
        }
        sd /= (data.size() - 1);
        return Math.sqrt(sd);
    }
}
```

Figure D.1: Code for performing experiments.