Randomized Shared Queues

Hyunyoung Lee   Jennifer L. Welch
Department of Computer Science, Texas A&M University
College Station, TX 77843-3112, U.S.A.
{hlee, welch}@cs.tamu.edu

Abstract

This paper presents a specification of a randomized shared queue that can loose some elements or return them out of order (not in FIFO), shows that the specification can be implemented over the probabilistic quorum algorithm of [8], and analyzes the behavior of this implementation. Distributed algorithms that can tolerate some lost and out-of-order messages are candidates for replacing the message queues with random queues. The modified algorithms will inherit positive attributes concerning load and availability from the underlying queue implementation. The behavior of an application – a class of combinatorial optimization algorithms - when it is implemented using random queues is analyzed.

1 Introduction

Quorum systems have been receiving significant attention because they provide consistency and availability of replicated data and reduce the communication bottleneck of some distributed algorithms (cf. [8] for references). The probabilistic quorum model [8] relaxes the intersection property of the strict quorum system, such that pairs of quorums only need to intersect with high probability. In [6], *random registers* are defined as memory cells in which certain types of random errors can occur. It is shown in [6] that random registers can be used as an abstraction of probabilistic quorum systems. In particular, the typical access operations (read, write) are shown to have lower message complexity for random registers implemented with the probabilistic quorum algorithm of Malkhi et al. [8, 7] when compared to conventional shared memory implemented over strict quorum systems. At the same time, random registers inherit the known properties of the probabilistic quorum system, such as providing high availability and optimal load simultaneously [8].

In this paper, we extend the results of [6], which considers only the basic memory access operations (read, write), to one of the fundamental abstract data structures: the queue. We define a *random queue* – which shares the semantics of its access operations with traditional queues, but which can suffer from certain random errors. In particular, random queues can loose elements with a certain well-defined probability. The interdependency of the queue elements introduces problems that do not arise for random registers. We show that random queues can be implemented using the probabilistic quorum algorithm and, thus inherit its properties with respect to load and availability.

Queues are a fundamental concept in many areas of computer science. A common application in distributed computing are message queues in communication networks. Many distributed algorithms use high-level communication operations, such as scattering or all-to-all broadcasts (cf. Chapter 1 of [3] for an overview). These algorithms can typically tolerate inaccuracies in the order in which the queue returns its elements, as the order of the elements in the message queue is typically impacted by the unpredictability of the communications network. Furthermore, we consider randomized algorithms, in which the queue elements contain data that can be incorrect or otherwise inappropriate with some probability. Algorithms of this type can typically tolerate the random disappearance of elements in the queue (with some small probability). We believe that this constitutes a large class of algorithms, which can take advantage of random queues and their benefits of optimal load and high availability. As an example of applications from this class, we analyze the behavior of a class of optimization algorithms [1], when used with random queues.

In [12, 13], Yelick et al. propose several irregular data structures and a *relaxed* consistency model for those data structures. For example, a *task queue* is an unordered collection of objects in which the priorities are locally, but not globally, observed. Such task queues can be used in the load balancing of the tasks of irregular applications. For the task queues of [12, 13], the randomization affects only the priorities. The number of enqueued tasks is always preserved. In [4], Chakrabarti et al. propose using distributed priority queues for the load balancing of parallel processors with dynamic scheduling algorithms. Again, the distributed queues affect the performance gain in a realistic execution environment compared to that with a centralized queue. However, they do not specify any random behavior of queue operations.

2 Definitions

In this section, we define the system model. (The presentation of this material is similar to that in [6].)

2.1 Shared Objects

The data type of a shared object is defined by a set of operations and set of allowable sequences of those operations.

An **operation** consists of an **invocation** and a matching **response**. The invocation indicates the specific object and contains any inputs, while the response also indicates the relevant object and contains any outputs.

In distributed implementations of a shared object, in which several processes can perform operations on the object, operations cannot generally happen instantaneously. Thus an invocation will initiate a series of actions, which at a later time will culminate in the response. Since an operation takes place over an interval of time, there can be overlapping concurrent operations taking place on a shared object, by different processes. Throughout this paper, we assume that each process has at most one operation pending at a time.

2.2 Processes and Their Steps

A **process** is a (possibly infinite) state machine which has access to a random number generator. The process has a distinguished state called the **initial state**.

We assume a system consisting of a collection of n **client** processes and r **server** processes. A client process runs on a processor that also runs an application process which is part of a distributed application that is written assuming shared data objects. The client process communicates with the shared memory application process above it and with the message passing system below it. A server process stores replicated data and interacts with client processes through the message passing system. We will restrict attention to algorithms (such as ours) in which only client processes use randomization; trivial extensions to the model would allow servers also to be randomized.

There is some set of **triggers** that can take place in the system. Triggers consist of operation invocations and message receptions. The occurrence of a trigger at a process causes the process to take a **step**. During the step, the process applies its transition function to its current state, the particular trigger, and a random number to generate a new state and some **outputs**. The outputs can include (at most) one operation response and a set of messages to be sent. A step is completely described by the current state, the trigger, the random number, the new state, and the set of outputs.

2.3 Adversaries and Executions

There are three potential sources of nondeterminism in the system from the viewpoint of the shared object implementation:

1. the sequences of random numbers available to the client processes (due to the random number generators),
2. the sequences in which operation invocations are made on the client processes (due to the application program that is using the shared object layer), and
3. variability in the message delays.

We abstract the last two sources of nondeterminism into a construct called an "adversary." Formally, an **adversary** is a partial function from the set of all sequences of steps to the set of triggers. That is, given a sequence of steps that have occurred so far, the adversary determines what trigger will happen next. Note that the adversary *cannot* influence what random number is received in the next step, only the trigger. Let RAND be the set of all n-tuples of the form $\langle R^1, \ldots, R^n \rangle$ where each R^i is an infinite sequence of integers in $\{0, \ldots, D\}$. D indicates the range of the random numbers. R^i describes the sequence of random numbers available to client process i in an execution — R^i_j is the random number available at step j. Call each element in RAND a **random tuple**.

Given an adversary A and a random tuple $\mathcal{R} = \langle R^1, \ldots, R^n \rangle$, define an **execution** $exec(A, \mathcal{R})$ to be the sequence of steps $\sigma_1 \sigma_2 \ldots$ such that

- the current state in the first step of each process (client and server) i is i's initial state;
- the current state in the j-th step of process i is the same as the new state in the $(j-1)$-st step of i, for all processes i and all $j > 1$;
- the trigger in σ_j equals $A(\sigma_1 \ldots \sigma_{j-1})$, for all $j \geq 1$ (the trigger is chosen by the adversary);
- the random number in σ_j equals R^i_j, where i is the process in σ_j's trigger (the random number comes from $\mathcal{R}$, not the adversary).

If the adversary can generate arbitrary triggers, then it will be very difficult, if not impossible, to achieve anything sensible. Thus we put the following restrictions on the adversary:

- (Application related) The sequence of operation invocations at each process is consistent with the application layer above. That is, the operation invocations reflect the shared memory accesses that the application wants to make.
- (Message passing related) Every message received was previously sent and every message sent is eventually delivered exactly once. That is, the message passing system is asynchronous and reliable, with the exact delays under the control of the adversary.

An execution e is **complete** if it is either infinite or, in the case it is finite, $A(e)$ is undefined. This means that there is nothing further to do — the application is through making calls on the shared objects and no messages are in transit.

3 A Random Queue

In this section, we specify a randomized shared queue and propose an implementation for it. We then analyze the behavior of the implementation.

3.1 Specification of Random Queue

We define a **random queue** to be a randomized version of a shared queue, of which some properties are relaxed such that the number of enqueued data items is not preserved and the items can be dequeued out of order (not in FIFO).

A **queue** Q shared by several processes supports two operations, Enq(Q, v) and Deq(Q, v). Enq$_i(Q, v)$ is the invocation by process i to enqueue the value v, Ack$_i(Q)$ is the response to i's enqueue invocation, Deq$_i(Q, v)$ is the invocation by i of a dequeue operation, and Ret$_i(Q, v)$ is the response to i's dequeue invocation which returns the value v. A possible return value is also $\perp$, indicating an empty queue. The set of values from which v is drawn is unconstrained.

We will focus on *multi-enqueuer, single-dequeuer* queues; thus, the enqueue can be invoked by all the processes while the dequeue can be invoked only by one process.

We assume for notational simplicity that every enqueued value is uniquely identified.

Given a real number p that is between 0 and 1, a system is said to implement a p-**random queue** if the following conditions hold for every adversary A.

- In every complete execution (of the adversary),
  - (Liveness) every operation invocation has a following matching response;
  - (Integrity) every operation response has a preceding matching invocation;
  - (No Duplicates) for each value x, Deq(Q, x) occurs at most once;
  - (Per Process Ordering) for all i, if Enq$_i(Q, x_1)$ ends before Enq$_i(Q, x_2)$ begins, then x_2 is not dequeued before x_1 is dequeued.
- (Probabilistic No Loss) for every enqueued value x, $\mathbf{Pr}[x \text{ is dequeued}] \geq p$.

That is, each enqueued element is either never dequeued (which occurs with probability at most $1 - p$) or is dequeued once (which occurs with probability at least p). For a given adversary, the probability space is all extensions (of that adversary) of any finite execution of the adversary that ends with the invocation to enqueue x.

3.2 Implementation of Random Queue

We now describe an implementation of a p-random queue. The next subsection computes the value of p, assuming that the application program using the shared queue satisfies certain properties.

The queue algorithm is based on the probabilistic quorum algorithm of Malkhi et al. [8]. There are r replicated memory servers.

The construction of Algorithm 1 proceeds in two steps. We begin by describing a random queue for the special case of a single enqueuer. The case of $n \geq 1$ enqueuers is implemented over a collection of n single enqueuer queues.

The enqueue operation (Enq) mirrors the probabilistic quorum write operation: The local timestamp is increased and attached to the element that is to be enqueued. The resulting pair is sent to the replicas in the chosen quorum, a randomly chosen group of k servers.

The key notion in the dequeue operation (SingleDeq) is a timestamp limit (T). At any given time, all timestamps, which are smaller than the current value T are considered to be outdated. T is included in the dequeue messages to the replica servers and allows them to discard all outdated values. Beyond this, SingleDeq mirrors the probabilistic quorum read operation: The client selects a random quorum, sends dequeue messages to all replica servers in the quorum and selects the response with the smallest timestamp t_d. It updates the timestamp limit to $T := t_d + 1$ and returns the element that corresponds to t_d.

Each replica server implements a conventional queue with access operations enqueue and dequeue. In addition, the dequeue operation receives the current timestamp limit as input and discards all outdated values (e.g., by means of repeated dequeue operations). The purpose of this is to ensure that there are exactly k replica servers that will return the element v_T with timestamp T in response to a dequeue request. Thus, the probability of finding this element (in the current dequeue operation) is exactly the probability that two quorums intersect. This property is of critical importance in the analysis in the following section. It does not hold if outdated values are allowed to remain in the replica queues, as those values could be returned instead of v_T by some of the replica servers containing v_T.

For the case of $n \geq 1$ enqueuers, we allocate n single enqueuer queues $Q_1, \ldots, Q_n$. The i-th enqueuer ($1 \leq i \leq n$) enqueues to Q_i. The single dequeuer dequeues from all n queues by making calls to the function Deq(), which selects one of the queues and tries to dequeue from it.

Deq() checks the next queue in sequence. The round-robin sequence used in Algorithm 1 can be replaced by any other queue selection criterion, which queries all queues with approximately the same frequency. The selection criterion will impact the order in which elements from the different queues are returned. However, it does not impact the probability of any given element being dequeued (eventually), as the queues do not affect each other, and the attempt to dequeue from an empty queue does not change its state.

See Appendix A for pseudocode (presented as Algorithm 1).

3.3 Analysis of Random Queue Implementation

For this analysis, we assume that the application program invoking the operations on the shared random queue satisfies a certain property. Every complete execution of every adversary consists of a sequence of segments. Each **segment** is a sequence of enqueues followed by a sequence of dequeues, which has at least as many dequeues as enqueues. Fix a segment. Let m_e, resp., m_d, be the total number of enqueue, resp., dequeue, operations in this segment. Let $m = m_e + m_d$. Let Y_i be the indicator random variable for the event that the i-th element is returned by a dequeue operation ($1 \leq i \leq m_e$). In the following lemma, the probability space is given by the enqueue and dequeue quorums which are selected by the queue access operations. More precisely, let $\mathcal{P}_k(r)$ denote the collection of all subsets of size k of the set $\{1, \ldots, r\}$. Since there are m enqueue and dequeue operations, we let $\Omega = \mathcal{P}_k(r)^m$ be the universe. The probability space for the following lemma is given by the finite universe Ω and the uniform distribution on Ω.

Lemma 1 *The random variables Y_i ($1 \leq i \leq m_e$) are mutually independent and identically distributed with* $\mathbf{Pr}(Y_i = 1) = p = \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right)$.

Proof: Since the queues $Q_1, \ldots, Q_n$ do not interfere with each other, they can be considered in isolation. That is, it is sufficient to prove the lemma for any given single enqueuer queue Q_i. Consider any single enqueuer queue Q_z and let m_z denote the number of enqueued elements. In order to prove mutual independence, we have to show

$$\mathbf{Pr}(\bigwedge_{i=1}^{m_z} Y_i = a_i) = \prod_{i=1}^{m_z} \mathbf{Pr}(Y_i = a_i) \tag{1}$$

for all possible assignments of $\{0, 1\}$-values to the constants a_i, for which the probability on the left-hand side is greater than zero. Thus, the following conditional probabilities are well-defined. For $h = 1$: trivially, $\mathbf{Pr}(\bigwedge_{i=1}^{1} Y_i = a_i) = \prod_{i=1}^{1} \mathbf{Pr}(Y_i = a_i)$.
For all $1 < h \leq m_z$:

$$\mathbf{Pr}(\bigwedge_{i=1}^{h} Y_i = a_i) = \mathbf{Pr}(Y_h = a_h | \bigwedge_{i=1}^{h-1} Y_i = a_i)\, \mathbf{Pr}(\bigwedge_{i=1}^{h-1} Y_i = a_i) \tag{2}$$

Let $j = \max\{i < h : a_i = 1\}$*. Clearly, the event $Y_h = 1$ does not depend on any event $Y_i = a_i$ for $i < j$. Thus

$$\mathbf{Pr}(Y_h = 1 | \bigwedge_{i=1}^{h-1} Y_i = a_i) = \mathbf{Pr}(Y_h = 1 | Y_j = 1 \wedge \bigwedge_{i=j+1}^{h-1} Y_i = 0)$$

The condition corresponds to the following case: The last dequeue operation has returned the j-th element. The dequeue operation immediately following the dequeue operation that dequeued j-th element misses elements $j+1$ to $h-1$. That is, the dequeue quorum R of the dequeue operation does not intersect the enqueue quorum S_i of any element $i \in \{j+1, \ldots, h-1\}$. Thus

$$\begin{aligned} \mathbf{Pr}(Y_h = 1 | Y_j = 1 \wedge \bigwedge_{i=j+1}^{h-1} Y_i = 0) &= \mathbf{Pr}(R \cap S_h \neq \emptyset | \bigwedge_{i=j+1}^{h-1} R \cap S_i = \emptyset) \\ &= \mathbf{Pr}(R \cap S_h \neq \emptyset) \quad \text{since quorums are chosen independently} \\ &= \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right) = p \end{aligned}$$

In summary, for all $1 < h \leq m_z$ and assignments of $\{0, 1\}$ to a_i,

$$\mathbf{Pr}(Y_h = 1 | \bigwedge_{i=1}^{h-1} Y_i = a_i) = p$$

By the formula of total probabilities, $\mathbf{Pr}(Y_h = 1) = p$. Thus, returning to (2):

$$\mathbf{Pr}(\bigwedge_{i=1}^{h} Y_i = a_i) = \mathbf{Pr}(Y_h = a_h)\ \mathbf{Pr}(\bigwedge_{i=1}^{h-1} Y_i = a_i)$$

Mutual independence (1) follows from this by induction. □

Theorem 1 *Algorithm 1 implements a random queue.*

Proof: The Integrity and Liveness conditions are satisfied since the adversary cannot create or destroy messages. The No Duplicates and Per Process Ordering conditions are satisfied by the definition of the algorithm. The Probabilistic No Loss condition follows from Lemma 1, which states that each enqueued value is dequeued with probability $p = \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right)$. □

4 Application of Random Queue: Go With the Winners

In this section we show how to incorporate random queues to implement a class of randomized optimization algorithms called Go with the Winners (GWTW), which was proposed by Aldous and Vazirani [1]. We analyze how the weaker consistency provided by random queues affects the success probability of the GWTW algorithm. Our goal is to show that the success probability is not significantly reduced.

*To handle the case when $a_i = 0$ for all $i < h$, define $Y_0 = a_0 = 1$.

4.1 The Framework of GWTW

GWTW is a generic randomized optimization algorithm. A combinatorial optimization problem is given by a state space S (typically exponentially large) and an *objective function* f, which assigns a 'quality' value to each state. The task is to find a state $s \in S$, which maximizes (or minimizes) $f(s)$. It is often sufficient to find approximate solutions. For example, in the case of the clique problem, S can be the set of all cliques in a given graph and $f(s)$ can be the size of clique s.

In order to apply GWTW to an optimization problem, the state space has to be organized in the form of a tree or a DAG, such that the following conditions are met: (a) The single root is known. (b) Given a node s, it is easy to determine if s is a leaf node. (c) Given a node s, it is easy to find all child nodes of s. The parent-child relationship is entirely problem-dependent, given that $f(child)$ is better than $f(parent)$. For example, when applied to the clique problem on a graph G, there will be one node for each clique. The empty clique is the root. The child nodes of a clique s of size k are all the cliques of size $k+1$ that contain s. Thus, the nodes at depth i are exactly the i-cliques. The resulting structure is a DAG. We could have defined a tree by considering ordered sequences of vertices.

Greedy algorithms, when formulated in the tree model, typically start at the root node and walk down the tree until they reach a leaf. The GWTW algorithm follows the same strategy, but tries to avoid leaf nodes with poor values of f, by doing several runs of the algorithm simultaneously, in order to bound the running time and boost the success probability (success means a node is found with a sufficiently good value of f). We call each of these runs a *particle* – which carries with it its current location in the tree and moves down the tree until it reaches a leaf node. The algorithm works in synchronous stages. During the k-th stage, the particles move from depth k to depth $k+1$. Each particle in a non-leaf node is moved to a randomly chosen child node. Particles in leaf nodes are removed. To compensate for the removed particles, an appropriate number of copies of each of the remaining particles is added.

The main theme to achieve a certain constant probability of success is to try to keep the total number of particles at each stage close to the constant B.

The framework of the GWTW algorithms is as follows: *At stage* 0*, start with* B *particles at the root. Repeat the following procedure until all the particles are at leaves: At stage* i*, remove the particles at leaf nodes, and for each particle at a non-leaf node* v*, add at* v *a random number of particles, this random number having some specified distribution. Then, move each particle from its current position to a child chosen at random.*

We consider a distributed version of the GWTW framework, presented as Algorithm 2 (See Appendix B for the pseudocode), which is a modification from the parallel algorithm of [10]. Consider an execution of Algorithm 2 on n processes. At the beginning of the algorithm (stage 0), B particles are evenly distributed among the n processes. Since, at the end of each stage, some particles may be removed and some particles may be added, the processes need to communicate with each other to perform load balancing of the particles (global exchange). We use shared-memory communication among the processes. In particular, we use shared queues to distribute the particles among processes. Between enqueues and dequeues in Algorithm 2, we need some mechanism to recognize the total number of enqueued particles in a queue. It can be implemented by sending one-to-one messages among the processes or by having the maximum possible number of dequeues per stage. (Finding more efficient, yet probabilistically safe, ways to end a stage is work in progress.)

When using random queues, the errors will affect GWTW, since some particles disappear with some probability. However, we show that this does not affect the performance of the algorithms significantly. In particular, we estimate how the disappearance of particles caused by the random queue affects the success probability of GWTW.

4.2 Analysis of GWTW with Random Queues

We now show that Algorithm 2 when implemented with random queues will work as well as the original algorithms in [1].

We use the notation of [1] for the original GWTW algorithm (in which no particles are lost by random queues): Let X_v be a random variable denoting the number of particles at a given vertex v. Let S_i be the number of particles at the start of stage i. At stage 0, we start with B particles. Then $S_0 = B$ and $S_i = \sum_{v \in V_\ell} X_v$, for $i > 0$, where V_ℓ is the set of all vertices at depth ℓ. Let $p(v)$ be the chance the particle visits vertex v. Then $a(j) = \sum_{v \in V_j} p(v)$ is the chance the particle reaches depth j at least. $p(w|v)$ is defined to be the chance the particle visits vertex w conditioning on it visits vertex v. The values $s_i, 1 \leq i < \ell$ are constants which govern the particle reproduction rate of GWTWs. The parameter κ is defined to express the "imbalance" of the tree as follows: For $i < j$, $\kappa_{ij} = \frac{a(i)}{a^2(j)} \sum_{v \in V_i} p(v) a^2(j|v)$, and $\kappa = \max_{0 \leq i < j \leq d'} \kappa_{ij}$.

Aldous and Vazirani [1] prove

Lemma 2

$$\mathbf{E} S_i = B \frac{a(i)}{s_i}, \quad 0 \leq i \leq d, \qquad \textit{and} \qquad \mathbf{var} S_i \leq \kappa B \frac{a^2(i)}{{s_i}^2} \sum_{j=0}^{i} \frac{s_j}{a(j)}, \quad 0 \leq i \leq d.$$

We will use this lemma to prove similar bounds for the distributed version of the algorithm, in which errors in the queues can affect particles. For this purpose, we formulate the effect of the random queues in the GWTW framework.

More precisely, given any original GWTW tree T, we define a modified tree T', which accounts for the effect of the random queues. Given a GWTW tree T, let T' be defined as follows: For every vertex in T, there is a vertex in T'. For every edge in T, there is a corresponding edge in T'. In addition to the basic tree structure of T, each non-leaf node v of T has an additional child w in T'. This child w is a leaf node. The purpose of the additional leaf nodes is to account for the probability with which particles can disappear in the random queues in Algorithm 2.

Given any node w in T' (which is not the root) and its parent v, let $p'(w|v)$ denote the probability of moving to w conditional on being in v. For the additional leaf nodes w in T', we set $p'(w|v) = 1 - p$, where $1 - p$ is the probability that a given particle is lost in the queue. For all other pairs (w, v), let $p'(w|v) = p \cdot p(w|v)$. Then $a'(i)$, $a'(i|v)$, S'_i, s'_i, X'_v, and κ' can be defined similarly for T'.

Given a vertex v of T, let $\bar{p}(v)$ denote the probability that Algorithm 2, when run with a single particle and without reproduction, reaches vertex v. The term "without reproduction" means that the distribution mentioned in the first "for" loop of the algorithm is such that the number of added particles is always zero. The main property of the construction of T' is:

Fact 1 *For any vertex v of the original tree T, $p'(v) = \bar{p}(v)$. Furthermore,*

$$\mathbf{Pr}(\textit{Algorithm 2 reaches depth } \ell) = p \cdot \ \mathbf{Pr}(\textit{GWTW on } T' \textit{ reaches depth } \ell)$$

for any $\ell \geq 0$.

Proof: We prove the first statement by induction on the depth of v. At depth $d = 0$ (base case), v is the root and $p'(v) = \bar{p}(v) = 1$. For the inductive step, let $v \in V_{\ell+1}$ for $\ell \in \mathbb{N}$. Let $u \in V_\ell$ be the immediate ancestor of v. Now,

$$p'(v) = p'(v|u)p'(u) = p \cdot p(v|u)p'(u) = p \cdot p(v|u)\bar{p}(u) = \bar{p}(v|u)\bar{p}(u) = \bar{p}(v).$$

For the second statement, it is sufficient to note that

$$\mathbf{Pr}(\text{Algorithm 2 reaches depth } \ell) = \sum_{v \in V_\ell} \bar{p}(v) = \sum_{v \in V_\ell} p'(v) = p \cdot \sum_{v \in V'_\ell} p'(v).$$

□

We can now analyze the success probability of Algorithm 2 (a combination of GWTW and random queues) by means of analyzing the success probability of baseline GWTW on a slightly modified tree. This allows us to use the results of [1] in our analysis. In particular,

Lemma 3

$$\mathbf{E}S'_i = B'\frac{p^{i-1}a(i)}{s'_i},\ \ 0 \le i \le d, \qquad \textit{and} \qquad \mathbf{var}S'_i \le \frac{1}{p}\kappa B'\frac{p^{i-1}a^2(i)}{{s'_i}^2}\sum_{j=0}^{i}\frac{s'_j}{p^{j-1}a(j)},\ \ 0 \le i \le d$$

Proof: We apply Lemma 2 to the GWTW process on T' and show that $\kappa' = \kappa/p$ and $a'(i) = p^{i-1}a(i)$ for all i. Note that for any $i \le \ell$ and $v \in V_i$, $p'(v) = p(v)p^i$. Thus, for any $1 < i \le \ell$

$$\begin{aligned} a'(i) &= \sum_{w\in V'_i} p'(w) = \sum_{w\in V'_i}\sum_{v\in V_{i-1}} p'(w|v)p'(v) \\ &= \sum_{v\in V_{i-1}} p'(v)\sum_{w\in V_i} p(w|v) = p^{i-1}\sum_{v\in V_{i-1}} p(v)\sum_{w\in V_i} p(w|v) = p^{i-1}a(i) \end{aligned}$$

For any $0 \le i < j \le \ell$,

$$\begin{aligned} \kappa'_{ij} &= \frac{a'(i)}{{a'}^2(j)}\sum_{v\in V'_i} p'(v){a'}^2(j|v) \\ &= \frac{p^{i-1}a(i)}{p^{2j-2}a^2(j)}\sum_{v\in V_i} p(v)p^i a^2(j|v)p^{2(j-i-1)} \\ &= p^{-1}\frac{a(i)}{a^2(j)}\sum_{v\in V_i} p(v)a^2(j|v) = \kappa_{ij}/p \end{aligned}$$

□

In order to allow a direct comparison between the bounds of Lemmas 2 and 3, it is necessary to relate the constants $(s_i)_{1\le i<\ell}$ and $(s'_i)_{1\le i<\ell}$. These constants govern the particle reproduction rate of GWTW and can either be set externally or determined by a sampling procedure described in [1]. If we set $s'_i = p^{i-1}s_i$ then the expectations of Lemmas 2 and 3 are equal and the variance bounds are within a factor of p of each other. The variance bound is used in [1] in connection with Chebyshev's inequality to provide a lower bound on the success probability of GWTW. It follows that the negative effect of random queues on the GWTW variance bounds can be compensated for by increasing the number B of particles at the root by a factor of $1/p$.

5 Conclusion

We have proposed a specification of a randomized shared queue data structure (random queue) that can exhibit certain errors — namely the loss of enqueued values — with some small probability. The random queue preserves the order in which individual processes enqueue, but makes no attempt to provide ordering across enqueuers. We showed that this kind of random queue can be implemented with the probabilistic quorum algorithm of [7, 8]. We identified, as potential applications of random queues, distributed algorithms that have a particular pattern of interprocess communication using message passing and that can tolerate a small number of message loss and inaccuracies in the order in which messages arrive. In these algorithms, the message passing communication is to be replaced by enqueuing and dequeuing information on random queues. We believe that this constitutes a large class of algorithms, which can take advantage of random queues and their benefits of optimal load and high availability. As an example of applications from this class, we analyzed the behavior of a class of combinatorial optimization algorithms (Go With the Winners [1]).

Another possible class of applications for a random queue is randomized Byzantine agreement algorithms in which the set of faulty processes can change from round to round (e.g. Rabin's algorithm [11]). Random errors in the queue can be attributed to faulty processes. Issues to be resolved include how to adapt the message passing algorithms to the situation when too few messages are received; also whether probabilistic quorum algorithms in [8] that tolerate Byzantine failures can be exploited here.

Actually, the applications we identified do not even require the per-process ordering — a shared multiset would work just as well. An open question is whether there is a randomized implementation of a multiset, with no ordering guarantees, that is more efficient in some measure than the algorithm presented in this paper. A complementary question is to identify distributed applications that would need ordering properties on a shared queue. Clearly one can imagine a variety of weakened queue definitions and a variety of implementations. Specifying and analyzing these are challenges for future work.

References

[1] D. Aldous and U. Vazirani. "Go With the Winners" Algorithms. In *Proc. of 35th IEEE Symp. on Foundations of Computer Science*, pp. 492–501, 1994.

[2] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* McGraw-Hill Publishing Company, 1998.

[3] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1989.

[4] S. Chakrabarti, A. Ranade, and K. Yelick. Randomized Load Balancing for Tree-structured Computation. *Proc. IEEE Scalable High Performance Computing Conf.*, pp. 666–673, 1994.

[5] B. Chor and C. Dwork. Randomization in Byzantine Agreement. In S. Micali, editor, *Randomness and Computing (Advances in Computing Research, vol. 5)*, pp. 443–497, JAI Press, Greenwich, CT, 1989.

[6] H. Lee and J. L. Welch. Applications of Probabilistic Quorums to Iterative Algorithms. *Proc. of 21st Int. Conf. on Distributed Computing Systems (ICDCS-21),* April 2001 (to appear).

[7] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Proc. of the 29th ACM Symp. on Theory of Computing*, pp. 569–578, May 1997.

[8] D. Malkhi, M. Reiter, and R. Wright. Probabilistic Quorum Systems. In *Proc. of the 16th Annual ACM Symp. on Principles of Distributed Computing*, pp. 267–273, Aug. 1997.

[9] R. Motwani and P. Raghavan. *Randomized Algorithms*, Cambridge University Press, 1995.

[10] M. Peinado and T. Lengauer. Parallel 'Go with the Winners' Algorithms in the LogP Model. In *Proc. of IPPS 97*, 1997.

[11] M. O. Rabin. Randomized Byzantine Generals. In *Proc. of the 24th Annual Symp. on Foundations of Computer Science*, pp. 403–409, 1983.

[12] K. Yelick et al., “Parallel Data Structures for Symbolic Computation,” Workshop on Parallel Symbolic Languages and Systems, Oct. 1995. Also available from http://www.cs.berkeley.edu/projects/parallel/castle/multipol/papers.html.

[13] K. Yelick et al., “Data Structures for Irregular Applications,” DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems, June 1993. Also available from http://www.cs.berkeley.edu/projects/parallel/castle/multipol/papers.html.

Appendix A: Implementation of p-random queue

Algorithm for client process – for single enqueuer and single dequeuer

Initially local variable $t = 0$ // enqueue timestamp
$T = 1$ // expected dequeue timestamp
when Enq(Q, v) occurs:
$t := t + 1$
send $\langle \text{enq}, v, t \rangle$ message to a randomly chosen quorum of size k and wait for acks
Ack(Q) // response to application

when SingleDeq(Q, v) occurs:
send $\langle \text{deq}, T \rangle$ to a randomly chosen quorum of size k and wait for replies
choose value v with smallest timestamp t_d
($\perp$ is considered to have largest timestamp)
if v is not $\perp$ then $T := t_d + 1$
Ret(Q, v) // response to application

Algorithm for server process i, $1 \leq i \leq r$:

Initially local variable $Qcopy$, a queue, is empty

when $\langle \text{enq}, v, T \rangle$ is received from client j:
enqueue (v, T) to $Qcopy$
send $\langle \text{ack} \rangle$ to client j

when $\langle \text{deq}, T \rangle$ is received from client j:
remove (dequeue) every element of $Qcopy$ whose timestamp smaller than T
if $Qcopy$ is empty let $w = \perp$
otherwise let w be the result of dequeue on $Qcopy$
send $\langle w \rangle$ to client j

Algorithm for a dequeuer extension for $m \geq 1$ enqueuers

Initially local variable $i = 0$, shared queue $Q = (Q_1, \ldots, Q_n)$ // an array of n single enqueuer queues

when Deq(Q, v) occurs
$i := (i \mod n) + 1$
Deq(Q_i, v)
Ret (Q, v)

Algorithm 1: Implementation of p-random queue Q

Appendix B: Distributed version of GWTW framework

Shared variables are random queues Q_i, $1 \leq i \leq n$, each dequeued by process i and initially empty

Code for process i, $1 \leq i \leq n$:

Local variable: integer s, initially 0

Initially $\frac{B}{n}$ particles are at the root

```
while true do
    s++
    for each particle at a non-leaf node v     // clone the particles
        add at v a random number of particles, with some specified distribution
    endfor
    remove the particles at leaf nodes
    for each particle j     // move j to some process x's queue
        pick a random number x ∈ {1, ..., n}
        Enq(Q_x, j)
    endfor
    while not all particles are dequeued     // read from own queue
        Deq(Q_i, j)
    endwhile
    move each particle from its current position to a child chosen at random
endwhile
```

Algorithm 2: Distributed version of GWTW framework