Brief Announcement: Randomized Shared Queues[*]

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

ABSTRACT

This paper presents a specification of a randomized shared queue that can lose some elements or return them out of order (not in FIFO), shows that the specification can be implemented over the probabilistic quorum algorithm of [4, 3], 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. SPECIFICATION OF RANDOM QUEUE

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 [2] 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.

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, $\mathrm{Enq}(Q, v)$ and $\mathrm{Deq}(Q, v)$. $\mathrm{Enq}_i(Q, v)$ is the invocation by process i to enqueue the value v, $\mathrm{Ack}_i(Q)$ is the response to i's enqueue invocation, $\mathrm{Deq}_i(Q, v)$ is the invocation by i of a dequeue operation, and $\mathrm{Ret}_i(Q, v)$ 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.

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.

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

2. IMPLEMENTATION OF RANDOM QUEUE

We now describe an implementation of a p-random queue. The next section 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. [4]. There are r replicated memory servers.

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 incremented by one 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 that 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,

[*]A full version of this paper, which includes all proofs, is available as Technical Report: TR01-004, Department of Computer Science, Texas A&M University, March, 2001.

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 extend the single-enqueuer, single-dequeuer queue for an n-enqueuer, single-dequeuer queue by having n copies of single-enqueuer queue, i.e., n single-enqueuer queues $(Q_1, \ldots, Q_n)$, one per enqueuer. 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 can be replaced by any other queue selection criterion that 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.

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 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*

$$\mathbf{Pr}(Y_i = 1) = p = \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right).$$

THEOREM 1. *The algorithm in Section 2 implements a random queue.*

4. APPLICATION OF RANDOM QUEUE: GO WITH THE WINNERS

In this section we show how to incorporate random queues to implement a generic randomized optimization algorithm 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.

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 in the full paper. 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.

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.

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.

$$\mathbf{E}S_i = B\frac{a(i)}{s_i}, \quad 0 \leq i \leq d, \qquad \textit{and}$$

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

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* $\ell \geq 0$.

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.

$$\mathbf{E}S'_i = B'\frac{p^{i-1}a(i)}{s'_i}, \quad 0 \leq i \leq d, \qquad \textit{and}$$

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

In order to allow a direct comparison between the bounds of Lemmas 2 and 3, it is necessary to relate the constants $(s_i)_{1 \leq i < \ell}$ and $(s'_i)_{1 \leq i < \ell}$. 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. 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] D. Bertsekas and J. Tsitsiklis. *Parallel and Distributed Computation,* Prentice-Hall Inc., Englewood Cliffs, NJ, 1989.

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

[4] 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.