# Applications of Probabilistic Quorums to Iterative Algorithms

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*This paper presents a definition of a read-write register that sometimes returns out-of-date values, shows that the definition is implemented by the probabilistic quorum algorithm of [19], and shows how to program with such registers using the framework of Üresin and Dubois [25]. Consequently, existing iterative algorithms for an interesting class of problems (including finding shortest paths, constraint satisfaction, and transitive closure) will converge with high probability if executed in a system in which the shared data is implemented with registers satisfying the new definition. Furthermore, the algorithms in this framework will inherit positive attributes concerning load and availability from the underlying register implementation. A monotone version of the new register definition is specified and implemented; it can provide improved expected convergence time and message complexity for iterative algorithms.*

## 1. Introduction

Randomization is a powerful tool in the design of algorithms. As summarized in [20, 11], randomized algorithms are often simpler and more efficient than deterministic algorithms for the same problem. Simpler algorithms have the advantages of being easier to analyze and implement. A well known example is the factoring problem, for which simple randomized polynomial-time algorithms are widely used, while no corresponding deterministic polynomial time algorithm is known. Randomized algorithms have a failure probability, which can typically be made arbitrarily small and which manifests itself either in the form of incorrect results (Monte Carlo algorithms) or in the form of unbounded running time (Las Vegas algorithms).

In this paper, we will define a shared memory framework for distributed algorithms, in which the implementation of the shared memory can be randomized. In particular, read operations can return out-of-date values. We define new conditions, which constrain this error probability, such that an interesting class of popular algorithms will work correctly when implemented over our *random registers* (defined below). At the same time, our conditions are sufficiently weak to allow certain kinds of probabilistic replicated systems to implement random registers. These replicated systems have very attractive properties, such as high scalability, availability and fault tolerance [19]. The main problem in replicated systems is to maintain consistency among the replicas. Quorum systems try to maintain consistency by defining collections of subsets of replicas (*quorums*) and having each operation select and access one quorum from the collection. Traditional, or *strict*, quorum systems require all quorums in the collection to intersect pairwise. Malkhi et al. [19] introduce the notion of a *probabilistic* quorum system, in which pairs of quorums only need to intersect with high probability. Malkhi et al. show that this relaxation leads to significant performance improvements in the load of the busiest replica server and the availability of the quorum system in the face of replica server crashes. We show that our definition of random registers captures similar properties, by accommodating the probabilistic quorum system as one possible implementation.

As we will show, using random registers can result in improved load, availability in the face of server crashes, and message complexity, but seems to require a special style of programming. Apparently there is a tradeoff between ease of programming and performance, when randomized data structures are used. These results are somewhat analogous to the situation with "weak", or "hybrid", consistency conditions, which can be implemented quite efficiently but require the application programs to be data-race-free [1, 4].

To the best of our knowledge, little existing work has focused on defining the semantics of distributed data structures that sometimes return out-of-date values, or on trying to characterize classes of applications that can tolerate such data structures.

In this paper, we propose a formal definition of a random read/write register. The consistency condition provided by our definition is a probabilistic variation on the concept of regularity from Lamport's paper [16].

We show that our definition of a random register can be implemented by the probabilistic quorum algorithm of

[19, 18], which has the advantages mentioned above, that the load on the busiest replica server is limited and the availability in the face of server crashes is high.

Next we show how registers satisfying our definition can be used to program iterative algorithms in the framework presented by Üresin and Dubois [25]. The implication is that we can use existing iterative algorithms for a significant class of problems (including finding shortest paths, constraint satisfaction, and transitive closure) in a system in which the shared data is implemented with registers satisfying our condition, and be assured that the algorithms will converge with high probability. Furthermore, algorithms in the framework will inherit any positive attributes concerning load and availability from the underlying register implementation.

Then we show how a reasonable, and easily implemented, modification of our original definition can be analyzed to prove expected convergence time in the iterative framework. Simulation results show that there is a significant benefit from the modified definition in that iterative algorithms converge faster.

Finally, we prove that the use of random registers can lead to a significant reduction in message complexity compared to strict systems in at least one important situation.

## 2. Related Work

A number of consistency conditions for shared memory have been proposed over the years, including safety, regularity and atomicity [15, 16], sequential consistency [14], linearizability [12], causal consistency [3] and hybrid consistency [5]. These definitions have all been deterministic with little or no regard to possible errors.

Afek et al. [2] and Jayanti et al. [13] have studied a shared memory model in which a fixed set of the shared objects might return incorrect values, while the others never do. This model differs from the one we are proposing, where *every* object has some (small) probability of returning an incorrect value.

If the type of error caused by a randomized implementation is that there is some (small) probability of not terminating instead of producing a wrong answer, the difficulty in specifying the shared object is lessened, since any values returned will satisfy the deterministic specification. Examples of this situation include [24, 23, 10], discussed below.

Randomized implementations have been proposed for several shared data structures in various architectures, as we now discuss.

Malkhi et al. [19, 18] have proposed a probabilistic quorum algorithm to implement a read-write variable over a message passing system. Probabilistic quorums seem like a useful distributed building block, thanks to their good performance (analyzed in [19] and reviewed in Section 6.4). However, to make probabilistic quorums usable by programmers, a more complete semantics of the register which they implement must be given, together with techniques for programming effectively with them.

Shavit and Zemach have implemented novel randomized synchronization mechanisms called combining funnels [24] and diffracting trees [23] over simpler shared objects. In these algorithms, the effect of randomization is on the performance; wrong answers are never returned.

PRAM simulations using randomized data structures are shown in [10] and referenced in [19].

In this paper, we show that one class of iterative convergent algorithms can handle infrequent out-of-date values. The first analysis of the convergence of iterative functions when the input data can be out of date was by Chazan and Miranker [8]. Subsequently a number of authors refined this work (cf. Chapter 7 of [6] for an overview). Üresin and Dubois [25] give a general necessary and sufficient condition on the function for convergence. Essentially the same convergence theorem is presented in Chapter 6 of [6]. This class of functions includes solutions to many practical applications, including solving systems of linear equations, finding shortest paths, and network flow [6]. The convergence rates of iterative algorithms have been studied by, e.g., [6, 26]; the emphasis in these papers is on comparing the rate with out-of-date data to the rate with current data, under various scheduling and timing assumptions.

## 3. Specifying a Random Register (RR)

We are interested in randomized distributed algorithms that implement a shared read/write register. Our first task is to specify the behavior of such a register. Although the particular implementation to be discussed in this paper is a message-passing one, we would like the *specification* to be implementation-independent, so that it could apply to any kind of implementation.

**A Read/Write Register** A read/write **register** $X$ shared by several processes supports two operations, read and write. Each **operation** has an **invocation** and a **response**. $\text{Read}_i(X)$ is the invocation by process $i$ of a read, $\text{Write}_i(X, v)$ is the invocation by $i$ of a write of the value $v$, $\text{Return}_i(X, v)$ is the response to $i$'s read invocation which returns the value $v$, and $\text{Ack}_i(X)$ is the response to $i$'s write invocation. We will focus on *multi-reader, single-writer* registers.

A register allows sequences of invocations and responses that satisfy certain conditions, including the following: (1) the first item in the sequence is an invocation, (2) each invocation has a matching response, and (3) no process has more than one pending operation at a time.

In addition, the values returned by the read operations must satisfy some kind of **consistency condition**. Below we will present a randomized consistency condition.

**Processes and Their Steps** A **process** is a (possibly infinite) state machine which has access to a random number

generator. A process models the software at each node that implements the random register layer; it communicates with the shared memory application program above it and with some interprocess communication system below it. The process has a distinguished state called the **initial state**.

We assume a system consisting of a collection of $p$ processes.

There is some set of **triggers** that can take place in the system. Triggers consist of operation invocations as well as system-dependent events (for example, the receipt of a message in a message-passing system). 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 as well as some system-dependent events (for example, message sends in a message-passing system). A step is completely described by the current state, the trigger, the random number, the new state, and the set of outputs.

**Adversaries and Executions** To capture the nondeterminism due to the uncertainties in communication delays and pattern of operation invocations, we formally define an **adversary** to be 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 $p$-tuples of the form $\langle R^1, \ldots, R^p \rangle$ 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 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 $\mathcal{R} = \langle R^1, \ldots, R^p \rangle$, an **execution** $exec(A, \mathcal{R})$ can be defined to be a sequence of steps in the standard way (the full paper has a detailed description).

The adversary must respect the applications pattern of operation invocations and the characteristics of the communication medium.

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 variables and no further action is required by the interprocess communication layer.

**A Random Register** Given an execution $e$, a read operation $R$ in $e$ is said to **read from** write operation $W$ in $e$ if (1) $W$ begins before $R$ ends, (2) the value returned by $R$ is the same as that written by $W$, and (3) $W$ is the latest write satisfying the previous two conditions. Consider the example in Figure 1. If $R$ returns $a$, then it is defined to read from

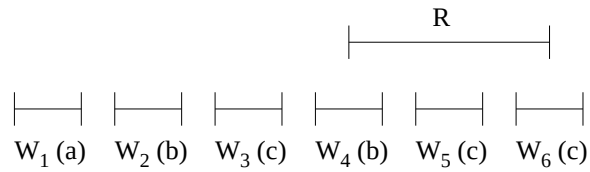


**Figure 1. Diagram for definition of reads from.**

$W_1$; if it returns $b$, then it is defined to read from $W_4$; and if it returns $c$, then it is defined to read from $W_6$.[1]

A system is said to implement a **random register** if, for every adversary $A$,

[R1] every operation invocation in every complete execution (of the adversary) has a matching response,

[R2] every read in every complete execution (of the adversary) reads from some write, and

[R3] for every finite execution $e$ (of the adversary) such that $A(e)$ is a write invocation, the probability that this write is read from infinitely often is 0, if an infinite number of writes are performed in the extension.

Notice that this is a kind of "worst-case" probabilistic definition as the probabilistic condition in [R3] must hold for *every* adversary and *every* write.

## 4. Implementing a Random Register

In this section, we show that the probabilistic quorum algorithm presented by Malkhi et al. [19, 18] implements a random register. Their algorithm works in a reliable, asynchronous message passing environment.

The following specializations are needed to the general model given in Section 3: Triggers include receiving a message from a process. Outputs include sending a message to a process. Constraints on the adversary include: every message sent is eventually received, and every message received was previously sent but not yet delivered.

The algorithm uses the notion of a **quorum**, which is a subset of the set of all replicas, of size $k$ (the **quorum size**). We have simplified the read/write register algorithm from [18] to assume only one writer and absence of failures. The shared register is replicated over $n$ servers. This replicated server system is used by $p$ processes through the shared register subsystem associated with each process. Each server keeps a local replica of the register to be implemented. A timestamp is associated with the replica. To perform a read, the shared register subsystem queries a quorum and returns the value with the largest timestamp resulting from the query. To perform a write, the shared register

[1] This definition might not capture the "real" write that is read from in a particular implementation, which might occur earlier. However, this definition is sufficient for proving that eventually each write stops being read from, which is what is required in this paper.

subsystem for the writer causes the replicas in a quorum to be updated with the new value and its new timestamp. Each quorum is chosen randomly with uniform distribution from the set of all possible quorums (all $k$-subsets of the set of all replicas) [19].

**Theorem 1** *The probabilistic quorum algorithm implements a random register.*

**Proof.** (Sketch) The interesting condition is [R3]. Choose a write $W$. We show in the full paper that the probability that at least one of the replicas in $W$'s quorum survives $\ell$ subsequent writes is at most $k \cdot (\frac{n-k}{n})^{\ell}$, where $k$ is the quorum size and $n$ is the number of servers. Since this expression goes to 0 as $\ell$ increases without bound, the probability that $W$ is read from infinitely often is 0. ■

To explain the advantages of the probabilistic quorum implementation, we review two important properties of quorum systems: availability and load. The **availability** of a quorum system is the minimum number of servers that must fail to cause at least one member of every quorum in the system to fail [22]. To achieve high availability of $\Omega(n)$, the smallest quorum size of the strict quorum system must be $\Theta(n)$. This property is satisfied by the *majority* quorum system, in which every quorum has size $\lfloor \frac{n}{2} \rfloor + 1$ [22].

The **load** of a quorum system was defined in [21] to be the minimal access probability of the busiest server, minimizing over all strategies for choosing the quorums. In [21] it was proved that the load of a strict quorum system with $n$ servers is at least $\max(\frac{1}{k}, \frac{k}{n})$, where $k$ is the size of the smallest quorum. Malkhi et al. [19] showed this result also holds asymptotically for probabilistic quorum systems. Thus the optimal (smallest) load for both probabilistic and strict systems is achieved when the smallest quorum has size $\Theta(\sqrt{n})$.

Naor and Wool [21] showed that strict quorum systems trade off availability and load such that any strict system with optimal load of $\Theta\left(\frac{1}{\sqrt{n}}\right)$ has only $O\left(\sqrt{n}\right)$ availability. Malkhi et al. [19] showed that *using probabilistic quorums breaks this trade-off* and achieves simultaneously high availability of $\Theta(n)$ and optimal load of $\Theta\left(\frac{1}{\sqrt{n}}\right)$.

## 5. Iterative Programs Using RRs

**A Framework for Iterative Algorithms** First, we give some background on Üresin and Dubois' result. The class of algorithms considered are those in which a function is applied repeatedly to a vector to produce another vector. In typical applications, each vector component may be computed by a separate process, based on that process' current best estimate of the values of all the vector components — estimates which might be out of date. Üresin and Dubois show that if the function satisfies certain properties and if the outdatedness of the vector entry estimates is not too extreme, then this iterative procedure will eventually converge to the fixed point of the function.

We use the following notation derived from [25].

Let $m$ be the size of the vector to be computed. If $\mathbf{x}$ denotes an $m$-vector, then $x_i$ denotes component $i$ of $\mathbf{x}$. We consider a function $\mathbf{F}$ from $S$ to $S$, where $S$ is the Cartesian product of $m$ sets $S_1, \ldots, S_m$.

Let *change* be a function from $N$ (the natural numbers) to $2^{\{1,\ldots,m\}}$, and let *view*$_i$, $1 \leq i \leq m$, be a function from $N$ to $N$. These functions will be used to produce a sequence of updated vectors, as detailed below. The value of *change*$(k)$ indicates which vector components are updated during update $k$; the value of *view*$_i(k)$ indicates which version of component $i$ is used in the update occurring during update $k$. We require the *change* and *view* functions to satisfy these conditions:

[A1] *view*$_i(k) < k$, for all $i$ and $k$, implying that the view of a component must always come from the past

[A2] each $i \in \{1, \ldots, m\}$ occurs in *change*$(k)$ for infinitely many values of $k$, implying that each component is updated infinitely often

[A3] for each $i \in \{1, \ldots, m\}$, *view*$_i(k)$ takes on a particular value for only finitely many values of $k$. This condition restricts the asynchrony by stating that a particular computed value for a component is used subsequently only finitely often.

Given a function $\mathbf{F}$, an initial vector $\mathbf{i}$, and *change* and *view* functions, define an **update sequence** of $\mathbf{F}$ to be an infinite sequence of vectors $\mathbf{x}(0)$, $\mathbf{x}(1)$, $\mathbf{x}(2), \ldots$ such that $\mathbf{x}(0) = \mathbf{i}$; and for each $k \geq 1$ and all $i$, $1 \leq i \leq m$, $x_i(k)$ equals $x_i(k-1)$ if $i$ is not in *change*$(k)$, and equals $F_i(x_1(view_1(k)), \ldots, x_m(view_m(k)))$ if $i$ is in *change*$(k)$.

Üresin and Dubois show that [A1] through [A3] are equivalent to the following condition (which will be used in Section 6): there exists an increasing infinite sequence of integers $\varphi(0) = 0, \varphi(1), \varphi(2), \ldots$, where updates $\varphi(K)$ through $\varphi(K+1) - 1$ comprise **pseudocycle** $K$, such that [B1] each component of the vector is updated at least once in each pseudocycle, and [B2] during each update in pseudocycle $K \geq 1$, the view of each component $i$ is a value that was updated in pseudocycle $K - 1$ or later.

Roughly speaking, a pseudocycle comprises at least one update to each vector component using information that is not too out of date.

The function $\mathbf{F}$ is called an **asynchronously contracting operator** (ACO) if there is a sequence of sets $D(0)$, $D(1)$, $D(2)$, ..., where $D(0) \subseteq S$, satisfying the following conditions: [C1] For each $K$, $D(K)$ is the Cartesian product of $n$ sets $D_1(K), \ldots, D_n(K)$. [C2] There exists some integer $M$ such that $D(K+1)$ is a proper subset of $D(K)$ for all $K < M$, and $D(K)$ contains a particular single vector for all $K \geq M$. This single vector is the fixed point of the

function. [C3] If **x** is in $D(K)$, then **F(x)** is in $D(K+1)$, for all $K$.

**Theorem 2** *[25] If* **F** *is an ACO on* $D(0), D(1), \ldots$*, then every update sequence of* **F** *starting with* $\mathbf{i} \in D(0)$ *converges to the fixed point of* **F**.

Their proof shows that after all the components are updated in the $K$th pseudocycle the computed vector subsequently is always contained in $D(K)$, and thus the vector converges to the fixed point in at most $M$ pseudocycles.

**Using Random Registers** An asynchronous iteration using random registers corresponds to an execution of the following algorithm (**Alg. 1**): Responsibility for updating the $m$ components of the vector **x** is partitioned among the $p$ processes. For each $j$, $1 \leq j \leq m$, component $j$ of **x**, denoted $x_j$, is held in a shared variable $X_j$, which is a random register. Each $X_j$ is initialized to contain the value of component $j$ of **i**, where **i** is the initial vector on which the iterative algorithm is to compute. Each processor repeatedly reads every $X_j$, applies the function $F$ to the data, and updates the $X_j$'s for which it is responsible. This algorithm satisfies:

**Theorem 3** *If* **F** *is an ACO on* $D(0), D(1), \ldots$*, then in every complete execution of Alg. 1 using random registers initialized to a vector in* $D(0)$*, the computed vector eventually converges to the fixed point of* **F** *with probability 1.*

**Proof.** We show that the update sequence extracted from an execution satisfies [A1], [A2] and [A3] with probability 1. Then Theorem 2 will hold with probability 1.

Condition [A1] is satisfied in any execution thanks to part [R2] of the definition of a random register, since the value returned by a read is always a value that was previously written. Condition [A2], which says that each vector component is updated infinitely often, is really a requirement on the application. This is satisfied in any complete execution produced by an adversary, since the adversary must be consistent with the application and the application has the necessary infinite loop. Finally, condition [A3] is satisfied with probability 1, since it is equivalent to part [R3] of the definition of a random register. ■

## 6. Monotone Random Register

In this section, we define a variation of a random register that satisfies two additional properties.

One property is that the values returned by the register are *monotone*, meaning that if a read reads from a certain write, then no subsequent read by the same process reads from an earlier write. This requirement should yield performance improvement by avoiding updates which might be wasted on reading more outdated values even though a more recent value has already been read in a previous update.

### 6.1. Definition

A random register is **monotone** if it satisfies the following two additional conditions for every adversary. The first condition is that the returned values are monotone:

[R4] In every execution, if read $R$ by process $i$ follows read $R'$ by process $i$ then $R$ does not read from a write that precedes the write from which $R'$ reads.

The second additional condition is needed in order to bound the convergence time when computing an ACO using monotone random registers. Let $Y$ be a random variable whose value is the number of reads by a process after a write $W$ until $W$ or a later write is read from by that process. The intuition is that $q$ is the probability of "success" for a read; the probability that $r$ reads are required is (at most) the probability that $r-1$ reads fail and then the $r$-th read succeeds.

[R5] There exists $q$, $0 < q \leq 1$, such that for all $r \geq 1$, $\Pr(Y = r) \leq (1-q)^{r-1} \cdot q$.

### 6.2. Implementation

Here we sketch a *monotone probabilistic quorum algorithm*: The shared register subsystem for each process keeps track of the largest timestamp, as well as the associated value, that it has returned so far during any read. If the queries to a read quorum all return smaller timestamps, then the saved value is returned, otherwise the original algorithm is followed.

**Theorem 4** *The monotone probabilistic quorum algorithm for* $n$ *replicas with quorum size* $k$ *implements a monotone random register with* $q = 1 - \binom{n-k}{k} / \binom{n}{k}$.

**Proof.** The interesting condition to show is [R5]. Choose a particular write $W$ in a particular execution and a particular process $i$. $W$ or a later write will be read from by $i$ if $W$ is followed by a read whose quorum overlaps $W$'s quorum. (There are other scenarios in which $i$ can obtain a value later than $W$, but we do not consider them in this analysis.)

The probability of a read $R$'s quorum not overlapping $W$'s is $\binom{n-k}{k} / \binom{n}{k}$, since there are $\binom{n}{k}$ possible choices for $R$'s quorum and there are $\binom{n-k}{k}$ choices for quorums that do not overlap $W$'s. The probability that $Y = r$ is at most the probability that $r-1$ reads have quorums that do not overlap $W$'s and then the $r$-th read's quorum does overlap $W$'s. The latter probability is $(1-q)^{r-1} \cdot q$, since quorums are chosen independently. ■

### 6.3. Expected Convergence Time for an ACO

In this section we show an upper bound on the expected number of rounds required per pseudocycle (cf. Section 5) in the execution of an ACO, if the vector components are implemented with a monotone random register.

A **round** is a minimal length (contiguous) subsequence of an execution in which each process reads all the registers,

applies the function, and updates its registers *at least* once. (If the system is synchronous, meaning that message delays and process step times are constant, then each round consists of *exactly* one execution of the loop by each process.)

**Theorem 5** *In every execution of Alg. 1 using monotone random registers with parameter $q$, the expected number of rounds per pseudocycle is at most $\frac{1}{q}$.*

**Proof.** Consider any adversary $A$ and any finite execution $e$ of $A$ that has just completed pseudocycle $h$, for any $h \geq 0$. We will calculate how many rounds are needed on average for pseudocycle $h+1$ to complete. (Pseudocycle 0 needs just one round since there are no values earlier than the initial values.)

Condition [B1] in the definition of pseudocycle implies that at least one round is needed.

Condition [B2] implies that for all $X_j$ and all processes $i$, $i$ must read from a write that is, or follows, the first write to $X_j$ in pseudocycle $h$, before pseudocycle $h+1$ can end. Once this read occurs, by [R4] all subsequent reads by process $i$ of $X_j$ will be at least as recent.

The required number of rounds is at most the random variable $Y$, as defined for [R5] in Section 6.1. [R5] and laws of probability show that $EY \leq \frac{1}{q}$. ■

**Corollary 6** *Let* **F** *be an ACO that converges in $M$ pseudocycles. The expected number of rounds taken by any complete execution of Alg. 1 using monotone random registers with parameter $q$ is at most $M/q$.*

We now provide an upper bound on the value of $1/q$ for the monotone probabilistic quorum algorithm with $n$ replicas and quorum size $k$. Proposition 3.2 in [19] implies that $\binom{n-k}{k}/\binom{n}{k} \leq (\frac{n-k}{n})^k$. Thus we have:

**Corollary 7** *For the monotone probabilistic quorum algorithm, the expected number of rounds per pseudocycle is at most $\frac{1}{1-(\frac{n-k}{n})^k}$.*

### 6.4. Expected Message Complexity for an ACO

In this section, we compare the expected message complexity per pseudocycle when executing an ACO for two implementation strategies of the vector components. One implementation strategy is the monotone probabilistic quorum algorithm. The other strategy consists of strict quorum systems, in which all quorums overlap. We show that although the number of rounds required for convergence is greater for the probabilistic case, there are some important situations in which the message complexity is smaller. The relationship between $n$, the number of servers used in the random register implementation, and $p$, the number of processes used by the ACO application, is crucial for this comparison. To ease the comparison, we consider synchronous systems, in which each process performs exactly one iteration of the loop in Alg. 1 per round.

Let $M_{prob}(k)$ be the expected number of messages sent per pseudocycle with the monotone probabilistic quorum implementation, and $M_{str}(k)$ be that with a strict quorum implementation, where the parameter $k$ indicates the size of the quorums. Inspecting Alg. 1 shows that the total number of messages sent per round is $2pmk + 2mk$. Then [Eqn. 1] $M_{prob}(k) = 2c_n m(p+1)k$ where $c_n$ is the expected number of rounds per pseudocycle. And [Eqn. 2] $M_{str}(k) = 2m(p+1)k$ since a strict quorum system uses one round per pseudocycle.

We will compare the expected message complexity of the two strategies in two extreme situations: quorum systems with high availability, and those with optimal load. (See Section 4 for definitions.)

We first consider quorum systems with high availability of $\Omega(n)$. For the probabilistic case, we set $k = \Theta(\sqrt{n})$, which ensures high probability of intersection between read and write quorums and also gives $\Omega(n)$ availability [19]. Plugging into Eqn. 1 gives [Eqn. 3] $M_{prob} = \Theta\left(2c_n m(p+1)\sqrt{n}\right) = \Theta\left(mp\sqrt{n}\right)$ since $1 < c_n < 2$ for all $n$ when the quorum size is $\sqrt{n}$ (cf. Corollary 7).

For the strict case, $\Omega(n)$ availability is only achieved when every quorum has size $\lfloor \frac{n}{2} \rfloor + 1$. Setting $k = \lfloor \frac{n}{2} \rfloor + 1$ in Eqn. 2 gives, $M_{str} = 2m(p+1)\left(\lfloor \frac{n}{2} \rfloor + 1\right) = \Theta\left(mpn\right)$ which is asymptotically larger than $M_{prob}$ for any $p$.

Now we consider quorum systems that have optimal load. For the probabilistic case, again we set $k = \Theta(\sqrt{n})$, which also gives optimal load. Then $M_{prob}$ is the same as Eqn. 3. There exist strict quorum systems in which a priori sets of servers form the quorums (e.g., finite projective planes [17], a grid construction [9], etc.). Some of these systems have $k = \Theta(\sqrt{n})$, and $M_{str} = \Theta(mp\sqrt{n})$, which yields the same message complexity as the probabilistic case. However, it trades off with lower availability.

## 7. Simulation

We have simulated systems of non-monotone and monotone random registers implemented using the algorithms from Sections 4 and 6.2 with a specific ACO. The simulation results shed some light on how much Corollary 7 overestimates the expected number of rounds per pseudocycle in the monotone case, the convergence behavior in the non-monotone case, and the difference between the synchronous and asynchronous cases.

Our example application is an all-pairs-shortest-path (APSP) algorithm presented in [25] and shown there to be an ACO. The vector **x** to be computed is two-dimensional, $n$ by $n$, where $n$ is the number of vertices in the graph. Initially each $x_{ij}$ contains the weight of the edge from vertex $i$ to vertex $j$ (if it exists), is 0 if $i = j$, and is infinity otherwise. The function **F** applied to **x** computes a new vector whose $(i, j)$ entry is $\min_{1 \leq k \leq n}\{x_{ik} + x_{kj}\}$. There are

$p = n$ processes, and process $i$ is responsible for updating the $i$-th row vector of $\mathbf{x}$, $1 \leq i \leq n$. The worst-case number of pseudocycles required for convergence of $\mathbf{F}$ is $\lceil \log_2 d \rceil$, where $d$ is the diameter of the input graph.

The sample input for our experiments is a directed graph on 34 vertices that is a chain, with vertex 1 the sink and vertex 34 the source. Each edge has weight 1. For this graph, $\lceil \log_2 33 \rceil = 6$ pseudocycles are required for convergence.

We simulated the execution of this APSP application over random registers, implemented with both the modified and original probabilistic quorum algorithm using 34 replicas, over a range of quorum sizes, from 1 to 18. Once the quorum size is at least 18, all quorums overlap, so every read gets the value of the latest write, and the randomization in the quorum choice has no effect. Message delays in synchronous executions are all the same, whereas those in asynchronous executions are exponentially distributed.

We measured the number of rounds until every process computes the APSP of given input graph. A round finishes when every process completes at least one iteration of Alg. 1 in which it reads the registers, applies the function, and writes its registers. Thus in the synchronous execution, a round consists of every process completing exactly one iteration of the loop, whereas in the asynchronous execution, processes can complete various numbers of iterations of the loop until one round is finished. At the end of each iteration of the loop, the simulation compares each process's local copy of the row for which that process is responsible, against the precomputed correct answer for that row. The simulation completes when each comparison is equal. (Cf. [6, 26] for discussions of the issues involved in detecting termination for iterative algorithms.)

The upper bounds on the expected number of rounds until convergence in the monotone case for the various quorum sizes were calculated using the formula from Corollary 7 and plotted in Figure 2. For each of the four combinations of monotone/non-monotone and synchronous/asynchronous, seven runs of the simulation were performed per quorum size and the number of rounds required for convergence was recorded for each. The average of these seven values was then plotted in Figure 2.

The synchronous and asynchronous executions do not reveal much difference in the results. This is because the structure of a round causes the differences in the message delays, which are exponentially distributed, to average out. Asynchronous executions sometimes terminated faster than synchronous ones if information propagation happened to be favorable.

The discrepancy between the calculated upper bound and the experimental value for the monotone case is quite large for very small quorums (e.g., 204 vs. 12.43 for synchronous and 9.08 for asynchronous executions when $k = 1$), but it decreases as the quorum size increases. One source of the

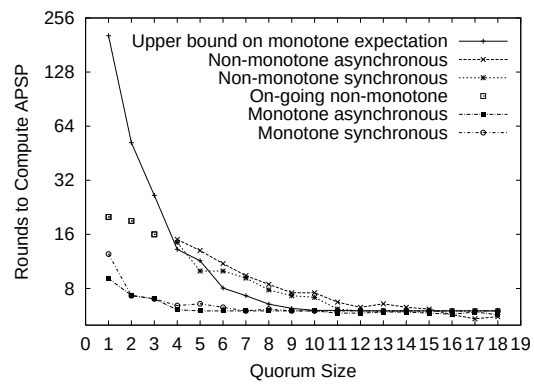


**Figure 2. Quorum Size vs. Rounds**

overestimate is in the proof of Theorem 5, where we did not take into account the fact that a read could obtain a value more recent than a given write without having to overlap any of that write's replicas.

The data indicates that the performance of the original algorithm is worse than that of the monotone algorithm. (The open squares in Figure 2 are *lower bounds* on the actual values — the simulations did not complete in a reasonable amount of time.) For quorum sizes above 3, the original algorithm's performance is even worse than the computed upper bound for the monotone case.

With monotone executions, notice how a small quorum (say 4) is as good as a large one (large enough to be strict). This is in line with the intuition behind the original probabilistic quorum paper [19].

## 8. Discussion

We have suggested two specifications of randomized data structures that can return wrong answers, namely two probabilistic versions of a regular register, non-monotone and monotone. We showed that both specifications can be implemented with the probabilistic quorum algorithm of [18, 19]. Furthermore, our specifications can be used to implement a significant class of iterative algorithms [25] of practical interest. We evaluated the performance of the algorithms experimentally as well as analytically, computing the convergence rate and the message complexity.

A number of challenging directions remain as future work. The definition of random register given here was inspired by the probabilistic quorum algorithm and was helpful in identifying a class of applications that would work with that implementation. It would be interesting to know whether our definition is of more general interest, that is, whether there are other implementations of it, or whether a different randomized definition is more useful.

Another direction is how to design more powerful read/write registers and other data types in our framework. Malkhi et al. [19] mention building stronger kinds of reg-

isters, such as multi-writer and atomic, out of the registers implemented with their quorum algorithms, by applying known register implementation algorithms. However, it is not clear how *random* registers can be used as building blocks in stronger register implementations.

This paper has addressed the fault-tolerance of replica *servers* for applications running on top of quorum implementations for shared data. In contrast, the issue of fault tolerance of *clients* for asynchronously contracting operators is another challenge, and is ongoing work. We consider the approximate agreement problem to be a good application for such a new model.

**Acknowledgments:** We thank Kathy Yelick for drawing our attention to reference [8], Nancy Amato and Marcus Peinado for helpful conversations, and Lyn Pierce for valuable comments on an earlier draft.

## References

[1] S. Adve and M. Hill. Weak Ordering — A New Definition. *Proc. 17th Annual Int'l Symp. on Computer Architecture*, pages 2–14, May 1990.

[2] Y. Afek, D. Greenberg, M. Merritt, and G. Taubenfeld. Computing with Faulty Shared Objects. *J. ACM*, Vol. 42, No. 6, pages 1231–1274, Nov. 1995.

[3] M. Ahamad, J. E. Burns, P. W. Hutto, and G. Neiger. Causal Memory. *Proc. 5th Int'l Workshop on Distributed Algorithms*, pages 9–30, Oct. 1991.

[4] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Nonsequential Execution: Definitions and Programming Strategies. *SIAM J. Computing*, Vol. 27, No. 1, pages 65–89, February 1998.

[5] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 679–690, 1992.

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

[7] J. E. Burns and G. L. Peterson. Constructing Multireader Atomic Values From Non-atomic Values. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 222–231, Aug. 1987.

[8] D. Chazan and W. Miranker. Chaotic Relaxation. *Linear Algebra and Its Applications*, Vol. 2, pages 199-222, 1969.

[9] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The Grid Protocol: A High Performance Scheme For Maintaining Replicated Data. In *Proc. 6th IEEE Conf. Data Engineering*, pages 438–445, 1990.

[10] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Simulating Shared Memory in Real Time: On the Computation Power of Reconfigurable Architectures. *Information and Computation*, Vol. 137, pages 103–120, 1997.

[11] R. Gupta, S. Smolka, and S. Bhaskar. On Randomization in Sequential and Distributed Algorithms. *ACM Computing Surveys*, Vol. 26, No. 1, pages 7–86, Mar. 1994.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, Vol. 12, No. 3, pages 463–492, July 1990.

[13] P. Jayanti, T. Chandra, and S. Toueg. Fault-tolerant wait-free shared objects. *J. ACM*, Vol. 45, No. 3, pages 451–500, May 1998.

[14] L. Lamport. How to Make a Multiprocessor that Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, Vol. C–28, No. 9, pages 690–691, Sep. 1979.

[15] L. Lamport. On Interprocess Communication, Part I: Basic Formalism. *Distributed Computing*, Vol. 1, No. 2, pages 77–85, 1986.

[16] L. Lamport. On Interprocess Communication, Part II: Algorithms. *Distributed Computing*, Vol. 1, No. 2, pages 86–101, 1986.

[17] M. Maekawa. A $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 2, pages 145–159, May 1985.

[18] D. Malkhi and M. Reiter. Byzantine Quorum Systems. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 569–578, May 1997.

[19] D. Malkhi, M. Reiter, and R. Wright. Probabilistic Quorum Systems. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 267–273, Aug. 1997.

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

[21] M. Naor and A. Wool. The Load, Capacity and Availability of Quorum Systems. In *Proceedings of the 35th IEEE Symposium on Foundations of Computer Science*, pages 214–225, 1994.

[22] D. Peleg and A. Wool. The availability of quorum systems. *Information and Computation*, 123(2), pages 210–233, 1995.

[23] N. Shavit and A. Zemach. Diffracting Trees. *ACM Trans. on Computer Systems*, Vol. 14, No. 4, pages 385–428, Nov. 1996.

[24] N. Shavit and A. Zemach. Combining Funnels. *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 61–70, 1998.

[25] A. Üresin and M. Dubois. Parallel Asynchronous Algorithms for Discrete Data. *J. ACM*, Vol. 37, No. 3, pages 558-606, July 1990.

[26] A. Üresin and M. Dubois. Effects of Asynchronism on the Convergence Rate of Iterative Algorithms. *J. Parallel and Distributed Computing*, Vol. 34, pages 66–81, 1996.