# Randomized Registers and Iterative Algorithms *

Hyunyoung Lee†
Department of Computer Science
University of Denver
Denver, CO 80208, U.S.A.
hlee@cs.du.edu

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

May 21, 2004

**Abstract**

We present three different specifications of a read-write register that may occasionally return out-of-date values — namely, a (basic) **random register**, a ***P*-random register**, and a **monotone random register**. We show that these specifications are implemented by the probabilistic quorum algorithm of Malkhi, Reiter, Wool, and Wright, and we illustrate how to program with such registers in the framework of Bertsekas, using the notation of Üresin and Dubois. Consequently, existing iterative algorithms for a significant class of problems (including solving systems of linear equations, 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 specifications. Furthermore, the algorithms in this framework will inherit positive attributes concerning load and fault-tolerance from the underlying register implementation. The expected convergence time for iterative algorithms using the monotone implementation is analyzed and shown experimentally to improve on that of the original implementation. The message complexity for iterative algorithms using the monotone probabilistic quorum implementation is shown to improve on that of non-probabilistic implementations in a quantifiable situation.

**Keywords:** distributed shared memory, registers, randomization, probabilistic quorums, iterative algorithms.

## 1 Introduction

Distributed computing systems are ubiquitous today, ranging from multiprocessors to local area networks to wide-area networks such as the Internet. In order to exploit the potential

*This research was supported in part by Texas Higher Education Coordinating Board grant ARP-00512-0091-2001. A preliminary version of this paper appeared as "Applications of Probabilistic Quorums to Iterative Algorithms", in *Proc. of 21st Int. Conf. on Distributed Computing Systems (ICDCS)*, pp. 21–28, April 2001.

†Contact author.

advantages of resource sharing, information distribution, and increased reliability that are offered by distributed systems, it is crucial to provide mechanisms for writing distributed programs that are correct and efficient. A popular approach is to provide the programmer with a shared memory paradigm for inter-process communication. The shared memory abstraction is attractive for writing concurrent programs, since it is a familiar style of programming and supports good software development practice. In the simplest case, the shared memory (or, more generally, any service) in a distributed computing environment could be concentrated in a single unit. However, in general, this solution is not scalable, not fault tolerant, and not appropriate for applications that require high availability. These properties are typically improved by replicating the memory or service, i.e., by including many redundant copies, or *replicas*.

In this paper, we define a shared memory framework for distributed algorithms, in which the implementation of the shared memory can be randomized. Randomization is a powerful tool in the design of algorithms. As summarized in [20, 10], 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 primality testing, for which randomized algorithms are widely used because they are simpler and more efficient than any known deterministic algorithms. 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 our randomized shared memory definition, read operations can return out-of-date values with some probability. We define new conditions that constrain this error probability, such that an interesting class of popular algorithms will work correctly when implemented over our *random registers*. 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 and fault-tolerance [19].

The main problem in replicated systems is to maintain consistency among the replicas. One way of maintaining consistency among replicas is using quorum systems. *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. They show that this relaxation leads to significant performance improvements in the load of the busiest replica server and the fault-tolerance of the quorum system in the face of replica server crashes, at the expense of strict consistency.

As we will show, using random registers can result in improved load and message complexity, and fault-tolerance in the face of server crashes, if implemented with a probabilistic quorum system, but seems to require a special style of programming. Apparently there is a trade-off 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 basic random read-write register. The consistency condition provided by our definition is a probabilistic variation on the concept of regularity from Lamport's paper [15].

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 fault-tolerance 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 of Bertsekas [6], presented by Üresin and Dubois [27]. The implication is that we can use existing iterative algorithms for a significant class of problems (including solving systems of linear equations, 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 fault-tolerance from the underlying register implementation.

Then we present an alternative version of our random register definition with an additional condition. The motivation for the new condition is to provide a more general version of our basic random register, in a way that the probability of a read value being outdated can be tuned as needed, allowing a trade-off between the degree of consistency and the performance provided by the distributed shared memory system. The new condition allows us to provide the probability distribution on how outdated the value returned by a read is when implemented with the probabilistic quorum algorithm. The type of register defined by this condition can be used to implement the same class of iterative algorithms as the basic type can.

The third definition of random register we present is a monotone version of our original definition. We show how a reasonable, and easily implemented, modification of the original definition can be analyzed to prove expected convergence time in the iterative framework. Next, we prove that the use of monotone random registers can lead to a significant reduction in message complexity compared to strict systems in a quantifiable situation. Finally, we present simulation results that show that there is a significant benefit from the monotone definition in that iterative algorithms converge faster.

The rest of this paper is organized as follows. Section 2 describes related work. In Section 3, we present our system model and definition of a random register. Section 4 shows that the probabilistic quorum algorithm of [19, 18] implements our definition. Section 5 reviews the framework for the iterative algorithms from [27], and shows how those conditions are satisfied by our basic definition of random registers. Section 6 presents an alternative definition of our random registers with an additional condition, provides the probability distribution on the

outdatedness of values returned by reads, when implemented with the probabilistic quorum algorithm, and shows that this condition can also be used to implement the iterative algorithms in the framework of Bertsekas [6], presented in [27]. In Section 7, we describe a monotone variation of our basic definition, show its expected convergence behavior, and identify situations in which it has superior message complexity. Section 8 presents our simulation results. Section 9 concludes the paper with a discussion of further research.

## 2 Related Work

A number of consistency conditions for shared memory have been proposed over the years, including safety, regularity and atomicity [14, 15], sequential consistency [13], linearizability [11], causal consistency [3] and hybrid consistency [5]. These definitions have all been deterministic with little or no regard to the possibility of incorrect return values.

Afek et al. [2] and Jayanti et al. [12] 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, 9], discussed below.

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.

Czumaj et al. [9] have implemented PRAM models over a reconfigurable mesh, using randomization to resolve conflicting accesses that occur at the same time step quickly with high probability. Again, wrong answers are never returned. Two other PRAM simulations that use randomized data structures are referenced in [19].

*Probabilistic Quorums.* Malkhi et al. [19, 18] have proposed a probabilistic quorum algorithm to implement a read-write variable over a message passing system. Each read is translated into messages to a subset ("quorum") of the replicated servers to obtain the latest value, and each write is translated into messages to a quorum of the replicated servers to store the latest value. Each quorum is chosen randomly so that with high probability the quorums overlap sufficiently for a read to obtain the latest value written. The smaller the quorums, the more efficient the algorithm is, but the larger the probability that a read will observe an out-of-date value. Probabilistic quorums seem like a useful distributed building block, thanks to their good performance (analyzed in [19]). 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. In this paper, we specify types of registers that can be implemented with the probabilistic quorums, and identify a programming framework

to effectively use such registers.

*Iterative Algorithms.* In this paper, we apply the result in [27] to identify one class of iterative convergent algorithms that can handle infrequent out-of-date values caused by randomized register semantics. 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 [7] for an overview). Üresin and Dubois [27] give a general necessary and sufficient condition on the function for convergence. Essentially the same convergence theorem is presented in a preceding work by Bertsekas [6], and a nice exposition of it can be found in Chapter 6 of [7]. This class of functions includes solutions to many practical applications, including solving systems of linear equations, optimization problems, finding shortest paths, dynamic programming, and network flow [6, 7]. The convergence rates of iterative algorithms have been studied in [7, 28]; 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

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.

## 3.1 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; thus, the read can be invoked by all the processes while the write can be invoked only by one process.

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 version of the consistency condition known as regularity. A register is **regular** if every read returns the value written either by an overlapping write or by the most recent write that precedes the start of the read [15].

Defining a probabilistic consistency condition requires specifying a probability space. We do so in the next few subsections.

### 3.2 Processes and Their Steps

In our model, 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.

### 3.3 Adversaries and Executions

No matter what the details of the implementation system, there will be three potential sources of nondeterminism, from the viewpoint of the register implementation:

1. the sequences of random numbers available to the processes (due to the random number generators)

2. the sequences in which operation invocations are made on the processes (due to the application program that is using the shared register layer)

3. uncertainties in the communication system used by the processes (for instance, variability in message delays for a message passing implementation, or variability in the response time for a shared memory implementation)

In order to facilitate the specification of probabilistic consistency conditions (as well as the analysis of randomized algorithms), we will 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; it has an influence on 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. The random numbers are uniformly distributed over the range $D$, and each random number is chosen independently. $R^i$ 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 $\mathcal{R} = \langle R^1, \ldots, R^p \rangle$, define an **execution** $exec(A, \mathcal{R})$ (of this adversary) to be the sequence of steps $\sigma_1, \sigma_2, \ldots$ such that

- the current state in the first step of each process $i$ 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 $\sigma_j$ equals $A(\sigma_1, \ldots, \sigma_{j-1})$, for all $j \geq 1$ (the trigger is chosen by the adversary);
- the random number in $\sigma_j$ equals $R^i_j$, where $i$ is the process at which the trigger in $\sigma_j$ occurs (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:

- 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. We assume that the application never has more than one operation pending per process at a time. More formally, for each finite sequence of steps $e$, $A(e)$ is an invocation for process $i$ only if a response by $i$ follows the latest preceding invocation for $i$ in $e$.
- Any conditions imposed by the nature of the underlying interprocess communication medium are respected. (For example, a message is received only if it was previously sent.)

An execution $e$ is **complete** if it is either infinite and no application process is starved or, in the case it is finite, $A(e)$ is undefined. In the finite case, 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.

### 3.4 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 $W_1$; 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 $A$,

[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.

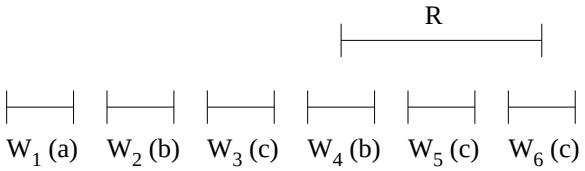


Figure 1: Diagram for definition of reads from.

[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 after $A(e)$.

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.

To be more explicit about the probability mentioned in condition [R3] of the definition, note that $e$ consists of a finite number of steps, say $m$. Thus $e = exec(A, \mathcal{R}_m)$, where $\mathcal{R}_m$ is the "prefix" of some random tuple $\mathcal{R}$ in which each component of $\mathcal{R}_m$ is the $m$-length prefix of the corresponding component of $\mathcal{R}$. Let $\mathcal{S}$ be the set of all executions of the form $exec(A, \mathcal{R}')$, where each $\mathcal{R}'$ is an (infinite) extension of $\mathcal{R}_m$, i.e., each of these executions is a possible future for $e$, for the given adversary. The subset of $\mathcal{S}$ consisting of all executions with an infinite number of writes is our probability space.

Condition [R2] is where "errors" can creep in, as compared to the more restrictive set of writes that can be read from in the original definition of regularity. However, [R3] limits these errors.

# 4 Implementing a Random Register with Probabilistic Quorums

In this section, we show that the probabilistic quorum algorithm in [19, 18] implements a random register. For simplicity, we first assume an asynchronous reliable message passing environment and no failure of *client* processes (which invoke shared memory operations). Furthermore, we consider a specific system model as follows. 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 a given size $k$ (the **quorum size**). We use the following definitions from [19]. Consider a universe

$U$ of $n$ replica servers. A **set system Q** over $U$ is a set of subsets of $U$. A **strict** quorum system **Q** over a universe $U$ is defined as a set system over $U$ such that for every $Q$, $Q' \in \mathbf{Q}$, $Q \cap Q' \neq \emptyset$. Each $Q \in \mathbf{Q}$ is called a quorum. A **probabilistic** quorum system is defined as follows:

An access strategy $w$ for a set system **Q** is a probability distribution on the elements of **Q**. A **probabilistic** quorum system is a tuple $\langle \mathbf{Q}, w, \epsilon \rangle$, where $\epsilon$ is a constant, $0 < \epsilon < 1$, such that $\Pr[Q \cap Q' \neq \emptyset] \geq 1 - \epsilon$, where $Q$ and $Q'$ are independent random quorums chosen according to the probability distribution $w$.

In this paper we assume a uniformly distributed access strategy. We also assume that the size $k$ of a quorum is a constant for the algorithm, that is, all quorums have the same size. We have simplified the read-write register algorithm from [18] to assume only one writer and the absence of failures.

The shared register $X$ is replicated over $n$ servers. This replicated server system is used by $p$ client processes. Each server keeps a local replica of the register to be implemented. A timestamp is associated with the replica. To perform a read, a client chooses a quorum, queries the quorum, and returns the value with the largest timestamp resulting from the query. To perform a write, a client chooses a quorum and causes the replicas in the quorum to be updated with the new value and its new timestamp. Each quorum is chosen randomly and independently, with uniform distribution from the set of all possible quorums (all $k$-subsets of the set of all replicas) [19].

Each replica server $r$ uses the following local variables:

- $val_r$ : holds the value of its replica of the (logical) shared register. Initially $val_r$ holds the initial value of the shared register.
- $ts_r$ : holds the timestamp associated with the value in $val_r$. Initially $ts_r = 0$.

The client on each process $i$ uses the following local variables:

- $rval_i$ : holds the most recent value received from the replica servers in the current quorum.
- $rts_i$ : holds the received timestamp associated with the value in $rval_i$.
- $numresp_i$ : holds the number of responses that $i$ has obtained so far from the current quorum.

In addition, the unique writer client process $w$ keeps a local variable $wts_w$, which holds the timestamp of the last write performed by $w$. Initially, $wts_w = 0$. The code for each replica server and each client to perform when each event occurs is presented as Algorithm 1.

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

**Proof.** Condition [R1] is true since messages are always delivered. Condition [R2] is true by the way the values are stored and exchanged.

We now show condition [R3]. Choose any adversary $A$ and any finite execution $e$ of $A$ such that $A(e)$ is a write invocation. Let $W$ be this write. To show that the probability that $W$ is

when Read$_i(X)$ occurs at process $i$: // invocation for a Read by process $i$
pick a random quorum $Q := \{q_1, \ldots, q_k\}$
$rts_i := 0$; $numresp_i := 0$
send QUERY messages to $q_1, \ldots, q_k$

when a QUERY message is received by replica server $r$ from process $i$:
send a VALUE($val_r$,$ts_r$) message to $i$

when a VALUE($v$,$t$) message is received by process $i$ from replica server $r$:
$numresp_i$++
if $t > rts_i$ then $rval_i := v$; $rts_i := t$ endif
if $numresp_i = k$ then Return$_i(X, rval_i)$ endif // response for the Read

---

when Write$_w(X, v)$ occurs at process $w$: // invocation for a Write by the writer $w$
pick a random quorum $Q := \{q_1, \ldots, q_k\}$
$wts_w$++; $numresp_w := 0$
send UPDATE($v$,$wts_w$) messages to $q_1, \ldots, q_k$

when an UPDATE($v$,$t$) message is received by replica server $r$ from process $w$:
$val_r := v$; $ts_r := t$
send an ACK message to $w$

when an ACK message is received by process $w$ from replica server $r$:
$numresp_w$++
if $numresp_w = k$ then Ack$_w(X)$ endif // response for the Write

---

**Algorithm 1:** (Simplified) probabilistic quorum algorithm [19, 18] to implement shared register $X$.

---

read from infinitely often is 0, we will show that the probability that at least one of the replicas in $W$'s quorum survives $\ell$ subsequent writes goes to 0 as $\ell$ increases without bound.

Let $k$ be the quorum size.

$$
\begin{aligned}
&\Pr[\text{at least one replica from } W\text{'s quorum survives } \ell \text{ subsequent writes}] \\
&\quad \leq \; k \cdot \Pr[\text{a specific replica } r \text{ from } W\text{'s quorum survives } \ell \text{ subsequent writes}] \\
&\quad = \; k \cdot \Pr[r \text{ is not in the quorum of any of the } \ell \text{ subsequent writes}] \\
&\quad = \; k \cdot \Pr[(r \notin Q_1) \bigcap (r \notin Q_2) \bigcap \ldots \bigcap (r \notin Q_\ell)] \\
&\qquad\quad \text{where } Q_i \text{ is the quorum of the } i\text{-th subsequent write} \\
&\quad = \; k \cdot \prod_{i=1}^{\ell} \Pr[r \notin Q_i] \qquad \text{since quorums are chosen independently} \\
&\quad = \; k \cdot \left(\frac{n-k}{n}\right)^{\ell} \qquad \text{since } n-k \text{ out of the } n \text{ replicas are not in a given quorum.}
\end{aligned}
$$

Clearly $\lim_{\ell \to \infty} k \cdot (\frac{n-k}{n})^{\ell} = 0$. ■

To explain the advantages of the probabilistic quorum implementation, we review two important properties of quorum systems: fault-tolerance and load.

The **fault-tolerance** of a quorum system is the minimum number of servers that must crash to cause at least one member of every quorum in the system to fail [22]. To achieve high fault-tolerance 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 [22], in which every set of size $\lfloor \frac{n}{2} \rfloor + 1$ is a quorum, and thus the fault-tolerance is $\lceil \frac{n}{2} \rceil$.

Malkhi and Reiter [18] proposed handling server failures for strict quorums by continuing to access servers until a quorum has responded. However, in an asynchronous system with undetectable crash failures of server processes, this approach can break the probabilistic properties of the probabilistic quorum algorithm. Therefore, we assume that we have some kind of failure detection mechanism[2].

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 have a trade-off between fault-tolerance and load such that any strict quorum system with optimal load of $\Theta\left(\frac{1}{\sqrt{n}}\right)$ has only

[2]One mechanism would be to use a failure detector to eliminate all faulty servers, and then choose a quorum at random among all quorums that do not contain a faulty server. See [29] for another approach to finding a live quorum.

$O(\sqrt{n})$ fault-tolerance. Malkhi et al. [19] showed that *using probabilistic quorums breaks this trade-off* and achieves simultaneously high fault-tolerance of $\Theta(n)$ and optimal load of $\Theta\left(\frac{1}{\sqrt{n}}\right)$.

# 5 Iterative Programs Using Random Registers

## 5.1 A Framework for Iterative Algorithms

Bertsekas [6] and Üresin and Dubois [27] presented a sufficient condition for the convergence of iterative algorithms when out-of-date data is sometimes accessed. In this section, we give a brief summary of Üresin and Dubois' exposition and point out that random registers satisfy their condition with probability 1. Thus, the same set of functions that converge in Üresin and Dubois' model will converge with probability 1 in the random registers model.

First, we give some background on the result in [6, 27]. 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. [6, 27] 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 [27].

Let $m$ be the size of the vector to be computed. For a vector $\mathbf{x}$ of size $m$, let $x_i$ denote component $i$ of $\mathbf{x}$, $1 \leq i \leq m$. 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 vectors, each of which can be considered as the result of an "update". 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 during update $k$.

Given a function $\mathbf{F}$, an initial vector $\mathbf{i}$, and *change* and *view* functions, an **asynchronous iteration** of $\mathbf{F}$ is defined to be an infinite sequence of vectors $\mathbf{x}(0)$, $\mathbf{x}(1)$, $\mathbf{x}(2), \ldots$ such that

- $\mathbf{x}(0) = \mathbf{i}$,
- 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 $change(k)$.

An asynchronous iteration is said to be **pseudoperiodic** if the *change* and *view* functions 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.

Üresin and Dubois show that a pseudoperiodic asynchronous iteration satisfies the following condition (which will be used in Sections 6 and 7): there exists an increasing infinite sequence of integers $\varphi(0) = 0, \varphi(1), \varphi(2), \ldots$, where the vectors resulting from 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.

In addition, we assume that the pseudocycles are the minimal sequences of updates with properties [B1] and [B2]. A pseudocycle comprises at least one update to each vector component using information that is not too out of date. Thus, when a pseudocycle completes, there has been some progress made toward the solution.

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 $D_1(K), \ldots, D_m(K)$.

[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 $\mathbf{x}$ is in $D(K)$, then $\mathbf{F(x)}$ is in $D(K+1)$, for all $K$.

**Theorem 2** *[27] If* $\mathbf{F}$ *is an ACO on* $D(0), D(1), \ldots$*, then every pseudoperiodic asynchronous iteration of* $\mathbf{F}$ *starting with* $\mathbf{i} \in D(0)$ *converges to the fixed point of* $\mathbf{F}$*.*

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.

### 5.2 Using Random Registers

Now we show that if each vector component in the framework just described is implemented with a random register, according to our definition from Section 3, then Theorem 2 is true with probability 1.

Code for each process $i$:

```
while true do
    for j := 1 to m do x_j := read(X_j) // obtain a view of each component
    y := F(x_1, ..., x_m) // compute updated vector locally
    for each j such that i is responsible for updating X_j do
        write(X_j, y_j) // update j-th component
```

**Algorithm 2:** Asynchronous iteration using random registers.

An asynchronous iteration using random registers corresponds to an execution of the following algorithm. In this algorithm, responsibility for updating the $m$ components of the vector $\mathbf{x}$ is partitioned among the $p$ processes. For each $j$, $1 \leq j \leq m$, component $j$ of $\mathbf{x}$, denoted $x_j$, is held in a shared variable $X_j$, which is a random register. Let $\mathbf{i}$ be the input vector on which the iterative algorithm is to compute. Each $X_j$ is initialized to contain the value of component $j$ of $\mathbf{i}$. The code is given as Algorithm 2.

**Theorem 3** *If* $\mathbf{F}$ *is an ACO on* $D(0), D(1), \ldots$*, then in every complete execution of Algorithm 2 using random registers initialized to a vector in* $D(0)$*, the computed vector eventually converges to the fixed point of* $\mathbf{F}$ *with probability 1.*

**Proof.** We show that the asynchronous iteration extracted from an execution is pseudoperiodic, i.e., 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 that is represented as Algorithm 2. This is satisfied in any complete execution produced by an adversary, since the adversary must be consistent with the application and the application (Algorithm 2) 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 $P$-Random Register

In this section, we propose an alternative definition of a random register that specifies a probability distribution on how outdated the read values are. The motivation for the new condition is to generalize the definition of our basic random register so that the probability of an outdated read can be tuned as needed. This can provide a trade-off between the degree of consistency and the performance of the distributed shared memory: Given the probability function $P$ (explained

below), one can implement the random register using different methods. By setting the probability of outdatedness to zero, the traditional (deterministic) register semantics can be defined and a strong consistency can be provided; by setting the probability of outdatedness to a larger value, a weaker consistency (with more nondeterministic behavior) can be implemented.

We show that the new definition is satisfied when implemented with the probabilistic quorum algorithm in [19, 18]. Furthermore, we compute the probability distribution of the outdatedness yielded by the implementation. We then show that the new definition can also be used to implement the iterative algorithms in the framework of Section 5.1.

### 6.1 Definition

Based on the definition of regularity given by Lamport [15] as described in Section 3, we define **allowable** values of a register as follows: Given a read, the values written by the overlapping writes and the most recent preceding write are called **allowable** values for that read.

Given an execution $e$, a read operation in $e$ that returns $v$ is defined to be **0-outdated** if $v$ is an allowable value. For $\ell > 0$, the read is **$\ell$-outdated** if the returned value $v$ is not allowable but is the value of the $\ell$-th write preceding[3] the beginning of the read (and is not the value of any later write that precedes the read). A read operation is said to be at least (resp., at most) $\ell$-outdated if it is $j$-outdated for some $j \geq \ell$ (resp., $j \leq \ell$).

Let $P$ be a non-increasing function from the natural numbers to $[0, 1]$. A system is said to implement a **$P$-random register** when [R3] in Section 3.4 is replaced by the following probabilistic outdatedness condition:

[R3′] For every finite execution $e$ (of the adversary) such that $A(e)$ is a read invocation, the probability that this read returns an at least $\ell$-outdated value is at most $P(\ell)$.

### 6.2 Implementation

Here we show that Algorithm 1, the probabilistic quorum algorithm, implements a $P$-random register. As before, $n$ is the number of replicas and $k$ is the quorum size.

**Theorem 4** *The probabilistic quorum algorithm implements a $P$-random register, where $P(\ell) \leq k \cdot (\frac{n-k}{n})^{\ell}$.*

**Proof.** In Theorem 1, we showed that the probabilistic quorum algorithm implements a random register. To prove that the condition [R3] is satisfied, we showed:

$$\Pr[\text{at least one replica from } W\text{'s quorum survives } \ell \text{ subsequent writes}] \leq k \cdot \left(\frac{n-k}{n}\right)^{\ell} ,$$

[3] This is well defined since there is only one writer.

where $n$ is the number of replicas and $k$ is the quorum size. Thus,

$$\Pr[\text{a read returns a value that is at least } \ell\text{-outdated}] \leq k \cdot \left(\frac{n-k}{n}\right)^{\ell}.$$

■

### 6.3 Probability Distribution $p(\ell)$

In the previous section, we showed an upper bound on the probability of reading an at least $\ell$-outdated value. In this section we derive the precise value of the probability $p(\ell)$, that a read returns an exactly $\ell$-outdated value. Note that $p(\ell) = P(\ell) - P(\ell+1)$. We only consider cases when a read does not overlap a write.

Let $e$ be a fixed finite execution (of some adversary) with no writes in progress. For any extension $e'$ of $e$, and for every $t$, define the random variable $S_t$ as follows: Let $W_1, \ldots, W_t$ be the next $t$ writes in $e'$ after $e$ ends. $S_t$ is defined to be the number of replicas that do not occur in any of the quorums used by $W_1$ through $W_t$. We refer to such replicas as *surviving* replicas.

When we refer to the probability of $S_t$ having a certain value, the probability space is all extensions $e'$ of $e$ with that adversary. We will consider the worst-case starting point $e$ and worst-case adversary.

As observed in the proof of Theorem 8, the probability of returning an exactly $\ell$-outdated value is the difference between the probabilities of returning an at least $\ell$-outdated value and an at least $(\ell+1)$-outdated value. The probability of returning an at least $\ell$-outdated value is shown in Lemma 7 to be a function of the probability that the number of replicas surviving $\ell$ writes has a certain value. Lemma 6 provides a recursive formula for the latter probability; its proof relies in Lemma 5, which states the probability of a certain number of replicas surviving $t$ writes when the number of replicas surviving $t-1$ writes is given.

**Lemma 5** *Given that $c_{t-1}$ writes survive $t-1$ writes, the probability that $c_t$ writes survive $t$ writes is equal to the following:*

$$\Pr[S_t = c_t | S_{t-1} = c_{t-1}] = \frac{\binom{c_{t-1}}{c_{t-1}-c_t}\binom{n-c_{t-1}}{k-(c_{t-1}-c_t)}}{\binom{n}{k}}.$$

**Proof.** There will be $c_t$ replica values surviving after $t$ writes if and only if during the $t$-th write, $c_t$ replicas out of $c_{t-1}$ replicas are still not chosen for the quorum, when a write quorum of size $k$ is chosen out of $n$ replicas. Thus, for the $t$-th write quorum of size $k$, $c_{t-1} - c_t$ replicas are chosen out of $c_{t-1}$ replicas, and $k - (c_{t-1} - c_t)$ replicas are chosen out of $n - c_{t-1}$ replicas. There are $\binom{c_{t-1}}{c_{t-1}-c_t}$ ways of choosing the former replicas and $\binom{n-c_{t-1}}{k-(c_{t-1}-c_t)}$ ways of choosing the latter replicas. ■

**Lemma 6** $\Pr[S_t]$ *satisfies the following recursive equations.*

$$\textit{For } t = 1 : \Pr[S_1 = c_1] = \begin{cases} 1 & \textit{if} \quad c_1 = n - k \\ 0 & \textit{otherwise} \end{cases}$$

$$\textit{For } t > 1 : \Pr[S_t = c_t] = \sum_{c_{t-1}=0}^{n} \Pr[S_t = c_t | S_{t-1} = c_{t-1}] \cdot \Pr[S_{t-1} = c_{t-1}].$$

**Proof.** The basis follows because exactly $k$ replicas are written by $W_1$, leaving $n-k$ surviving replicas. The inductive statement is the result of applying the following formula for total probability [25]: $\Pr[B] = \sum_{i=1}^{n} \Pr[B|A_i] \Pr[A_i]$, where the $A_i$'s form a complete set of disjoint events. ■

Now consider any read $R$ in an extension $e'$ of $e$ that occurs between the $\ell$-th and $(\ell+1)$-st write of $e'$ (after $e$ ends).

**Lemma 7** *For any $\ell \geq 1$,*

$$\Pr[R \textit{ returns an at least } \ell\textit{-outdated value}] = \sum_{c_\ell=0}^{n} \frac{\binom{c_\ell}{k}}{\binom{n}{k}} \Pr[S_\ell = c_\ell].$$

**Proof.** Applying the formula for total probability [25], we get

$$\Pr[R \text{ returns an at least } \ell\text{-outdated value}]$$

$$= \sum_{c_\ell=0}^{n} \Pr[R \text{ returns an at least } \ell\text{-outdated value} | S_\ell = c_\ell] \cdot \Pr[S_\ell = c_\ell].$$

If there are $c_\ell$ surviving values, then $R$ will return a value that is at least $\ell$-outdated if and only if every element in its quorum is chosen from those surviving values. Thus:

$$\Pr[R \text{ returns an at least } \ell\text{-outdated value} | S_\ell = c_\ell] = \frac{\binom{c_\ell}{k}}{\binom{n}{k}}.$$

■

**Theorem 8** *The probabilistic quorum algorithm implements a $P$-random register, where*

$$p(\ell) = \sum_{c_\ell=0}^{n} \frac{\binom{c_\ell}{k}}{\binom{n}{k}} \Pr[S_\ell = c_\ell] - \sum_{c_{\ell+1}=0}^{n} \frac{\binom{c_{\ell+1}}{k}}{\binom{n}{k}} \Pr[S_{\ell+1} = c_{\ell+1}].$$

**Proof.** The probability that $R$ returns an $\ell$-outdated value is

$$\begin{aligned} &= \Pr[R \text{ returns an at least } \ell\text{-outdated value}] - \Pr[R \text{ returns an at least } (\ell+1)\text{-outdated value}] \\ &= \sum_{c_\ell=0}^{n} \frac{\binom{c_\ell}{k}}{\binom{n}{k}} \Pr[S_\ell = c_\ell] - \sum_{c_{\ell+1}=0}^{n} \frac{\binom{c_{\ell+1}}{k}}{\binom{n}{k}} \Pr[S_{\ell+1} = c_{\ell+1}] \end{aligned}$$

The last step follows from two applications of Lemma 7. ■

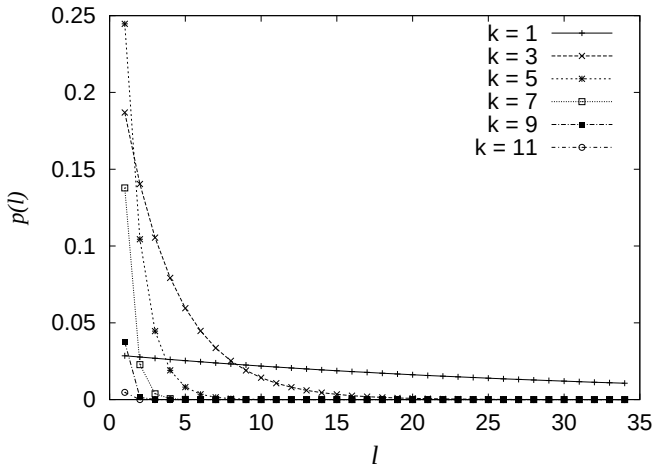


Figure 2: Probability distribution $p(\ell)$: outdatedness level $\ell$ vs. probability of being $\ell$-outdated.

We do not have a closed form to compute $p(\ell)$. However, the preceding analysis gives an algorithm to compute $p(\ell)$, the implementation of which can be found and downloaded from [16]. We ran the program in [16] with several sets of parameters. The plot in Figure 2 shows the result of the executions of the program with 34 replicas. Up to 34 outdatedness was computed for each of the quorum sizes from 1 to 18. We plotted only six quorum sizes of 1, 3, 5, 7, 9, and 11. After the quorum size of 11, the probabilities are too small to distinguish the differences. As expected, the probabilities for quorum sizes larger than 17 are all zeros. The computed results of other quorum sizes can also be found at [16].

Notice that with the quorum size of 1, the probabilities are almost constant (only decreasing a little when $\ell$ gets larger). This can be explained as follows: Consider quorums of size one. Take a round robin strategy, in which we choose each replica in turn. Then after enough (at least $\ell$) writes, we will read one-outdated, two-outdated, ..., values with almost the same probability. This behavior quickly changes with larger quorum sizes.

### 6.4 Performance of Iterative Algorithms with $P$-Random Registers

We show that, in the synchronous case, if an ACO is implemented using a $P$-random register for each vector component, the algorithm will converge with high probability, for suitable $P$.

We define a **round** to be a minimal length (contiguous) subsequence of an execution in which each process performs *at least one* execution of the while loop in Algorithm 2. 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 while loop by each process.

For a given $\ell \geq 1$, we partition an execution into "$\ell$-cycles", where each **$\ell$-cycle** consists of $\ell$ consecutive rounds. We will start numbering the $\ell$-cycles at 0 (analogously to pseudocycles). Note that in an $\ell$-cycle, each vector component is updated at least $\ell$ times.

Recall that $M$ is the number of pseudocycles required for the ACO to converge. We will now show that we can choose $\ell$ to be large enough so that the probability that each of the first $M$ $\ell$-cycles is a pseudocycle is arbitrarily large.

**Lemma 9** *Consider any read $R$ in $\ell$-cycle $K$, $K \geq 2$. The probability that $R$ reads from a write that precedes $\ell$-cycle $K-1$ is at most $P(\ell+1)$.*

**Proof.** Let $X_i$ be the variable read by $R$. Note that at least $\ell$ writes to $X_i$ occur between the start of $\ell$-cycle $K-1$ and $R$, since $R$ is in $\ell$-cycle $K$ and each $\ell$-cycle contains at least $\ell$ writes to each variable. Thus, if $R$ reads from a write that precedes $\ell$-cycle $K-1$, it returns a value that is at least $(\ell+1)$-outdated. By the definition of a $P$-random register and the fact that $P$ is non-increasing, the probability of this occurring is at most $P(\ell+1)$. ■

Let $C$ be the event that each of the $\ell$-cycles 0 through $M-1$ (i.e., each of the first $M$ $\ell$-cycles) is a pseudocycle.

**Lemma 10** $\Pr[C] \geq 1 - M \cdot n \cdot m \cdot \ell \cdot P(\ell)$.

**Proof.**

$$\begin{aligned}
\Pr[C] &= \Pr[\text{every read returns a value written in current or previous } \ell\text{-cycle}] \\
&= 1 - \Pr[\text{at least one read returns a value written before previous } \ell\text{-cycle}] \\
&\geq 1 - \sum_{\forall \text{ read } i} \Pr[\text{read } i \text{ returns a value written before previous } \ell\text{-cycle}] \\
&\geq 1 - \sum_{\forall \text{ read } i} P(\ell+1) \text{ by Lemma 9} \\
&\geq 1 - \sum_{\forall \text{ read } i} P(\ell) \text{ since P is non-increasing} \\
&= 1 - M \cdot n \cdot m \cdot \ell \cdot P(\ell)
\end{aligned}$$

since there are $M \cdot n \cdot m \cdot \ell$ reads in the first $M$ $\ell$-cycles. ■

**Corollary 11** *If $\lim_{\ell \to \infty} \ell \cdot P(\ell) = 0$, then for every $\delta > 0$, there exists an $\ell$ such that $\Pr[C] \geq 1 - \delta$.*

The corollary follows since we can make $M \cdot n \cdot m \cdot \ell \cdot P(\ell)$ as small as we like if $\ell \cdot P(\ell)$ goes to zero as $\ell$ increases. In other words, after $\ell \cdot M$ rounds, the probability that the ACO has converged becomes arbitrarily close to 1, for sufficiently large $\ell$.

Note that in the probabilistic quorum implementation, $P(\ell) \leq k \cdot \left(\frac{n-k}{n}\right)^{\ell}$, where $1 \leq k < n$. Thus, $P(\ell)$ is an exponentially decreasing function of $\ell$, and $\lim_{\ell \to \infty} \ell \cdot P(\ell) = 0$.

# 7 Monotone Random Register

In this section, we define a monotone variation of a random register that satisfies two properties in addition to [R1] – [R3] from Section 3.4.

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 a 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.

We also place an additional, more technical, requirement on the register, in terms of its probabilistic behavior. This condition allows us to analyze the expected convergence time of an iterative algorithm in the [27] framework.

## 7.1 Definition

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

[R4] In every execution (of the adversary), 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. Assume the application program has an infinite number of reads. Let $Y$ be a random variable whose value is the number of reads (by a process), which start after a write $W$ ends, 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 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 $r \geq 1$, $\Pr[Y > r] \leq (1-q)^r$.

The probability space for [R5] is all writes in all complete executions of the adversary. Once a write operation has completed, the probability of returning a value from an earlier write is independent and constant across subsequent read operations.

## 7.2 Implementation

Here we describe a *monotone probabilistic quorum algorithm*: Each client 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 12** *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.** Condition [R4] is clearly true. The rest of the proof shows that [R5] holds.

In a given execution, consider a write $W$ and a process $i$. The write $W$ or a later write will be read from by $i$ if $W$ is followed by a read whose quorum overlaps $W$'s or a later write's quorum (let the first such write's quorum be denoted by $Q$).

The probability of a read $R$'s quorum not overlapping $Q$ 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 $Q$.

The probability that $Y > r$ is bounded above by the probability that $r$ reads have quorums that do not overlap $Q$. The latter probability is $(1-q)^r$, since quorums are chosen independently. ■

### 7.3 Expected Convergence Time for an ACO

In this section we derive the expected number of rounds required per pseudocycle (cf. Section 5.1) in the execution of an ACO, if the vector components are implemented with monotone random registers.

**Theorem 13** *In every execution of Algorithm 2 by $p$ processes using monotone random registers with parameter $q$ and a vector of length $m$, the expected number of rounds per pseudocycle is at most*

$$\sum_{u=1}^{pm} \binom{pm}{u} (-1)^{u+1} \frac{1}{1-(1-q)^u}.$$

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

Condition [B1] in the definition of a pseudocycle implies that at least one round is needed. Condition [B2] implies that for all vector components $X_j, 1 \leq j \leq m$, and all processes $i, 1 \leq i \leq p$, the process $i$ must read from the first write to $X_j$ in pseudocycle $h$ or from a subsequent write to this vector component, before pseudocycle $h+1$ can complete. Denote by $Y_{i,j}$ the total number of read operations of vector component $X_j$ performed by process $i$ starting from the beginning of pseudocycle $h+1$ until this successful read occurs. Once this read occurs, all subsequent reads by process $i$ of the vector component $X_j$ will be at least as recent, due to the monotonicity condition [R4]. As in [R5] each random variable $Y_{i,j}$ is geometrically distributed with success probability $q$.

Recall that a round completes as soon as every process performs at least one execution of the while loop in Algorithm 2. This implies that in each round there exists at least one process which has performed exactly one read of each vector component. Thus, in the worst case, each process performs one read of each vector component per round. It follows that the number of rounds can be upper-bounded by the random variable $X = \max(Y_{1,1}, Y_{1,2}, \ldots, Y_{p,m})$. Essentially the same problem of computing the expectation of the maximum of a number of

geometrically distributed random variables, was studied in [26], resulting in the expression given in the statement of the theorem. For completeness, we give a more elementary derivation here. We have $\mathrm{E}[X] = \sum_{r=0}^{\infty} \Pr[X > r]$ since $X$ is a non-negative integer-valued random variable. In other words,

$$\mathrm{E}[X] = \sum_{r=0}^{\infty} \left(1 - \Pr[Y_{1,1} \leq r, \ldots, Y_{p,m} \leq r]\right) = \sum_{r=0}^{\infty} \left(1 - \Pr[Y \leq r]^{pm}\right).$$

The last equality holds because the random variables $Y_{i,j}$ are independent and identically distributed. Recall that $Y$ denotes a geometric random variable with success probability $q$, that is, $\Pr[Y = r] = (1-q)^{r-1}q$; see [R5]. Notice that $\Pr[Y > r] = (1-q)^r$. Hence,

$$\mathrm{E}[X] = \sum_{r=0}^{\infty} \Big(1 - \Big(1 - \Pr[Y > r]\Big)^{pm}\Big) = \sum_{r=0}^{\infty} \Big(1 - \Big(1 - (1-q)^r\Big)^{pm}\Big).$$

For convenience, we will use the abbreviation $z = 1 - q$. The binomial formula shows that

$$(1 - z^r)^{pm} = \sum_{u=0}^{pm} \binom{pm}{u} (-z^r)^u = \sum_{u=0}^{pm} \binom{pm}{u} (-1)^u z^{ru}.$$

Therefore,

$$1 - (1 - z^r)^{pm} = \sum_{u=1}^{pm} \binom{pm}{u} (-1)^{u+1} z^{ru}.$$

It follows that the expectation $\mathrm{E}[X]$ can be computed by

$$\begin{aligned} \mathrm{E}[X] &= \sum_{r=0}^{\infty} \Big(1 - \Big(1 - z^r\Big)^{pm}\Big) = \sum_{r=0}^{\infty} \sum_{u=1}^{pm} \binom{pm}{u} (-1)^{u+1} z^{ru} \\ &= \sum_{u=1}^{pm} \binom{pm}{u} (-1)^{u+1} \sum_{r=0}^{\infty} z^{ru} = \sum_{u=1}^{pm} \binom{pm}{u} (-1)^{u+1} \frac{1}{1 - z^u}, \end{aligned}$$

where the last equality holds because of the geometric series representation. ∎

An asymptotic expression of the expected number of rounds is given in the following Corollary.

**Corollary 14** *If the product $pm$ is large, then the expected number of rounds per pseudocycle of Algorithm 2 is given by*

$$\log_Q(pm) + O(1),$$

*where the base $Q$ of the logarithm is given by $Q = 1/(1-q)$.*

**Proof.** Combining Theorem 13 with the main proposition of [26] yields our claim. ∎

When the product $pm$ is small (at most 200), [26] shows that this expected value is at most 21.

### 7.4 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 based on the monotone probabilistic quorum algorithm, the other on strict quorum systems, in which all quorums overlap. Although the number of rounds required for convergence is greater in the probabilistic case, we show that there are some situations in which the message complexity is smaller. To ease the comparison, we consider synchronous systems, in which each process performs exactly one iteration of the loop in Algorithm 2 per round.

Let $k$ denote the size of the quorums. Let $M_{prob}(k)$ and $M_{str}(k)$ respectively denote the expected number of messages sent per pseudocycle with the monotone *probabilistic* quorum implementation and with the *strict* quorum implementation. Inspecting the code shows that the total number of messages sent per round is $2pmk + 2mk$, since each of the $p$ processes reads each of the $m$ vector components once, each of the $m$ vector components is written once, and each operation takes $2k$ messages. It follows that

$$M_{prob}(k) = c_n 2km(p+1), \tag{1}$$

where $c_n$ denotes the expected number of rounds per pseudocycle; and

$$M_{str}(k) = 2km(p+1) \tag{2}$$

because a strict quorum system uses one round per pseudocycle.

We compare the expected message complexity of the two strategies in the case of quorum systems with high fault-tolerance of $\Omega(n)$, which was defined in Section 4. Recall that in the probabilistic case a quorum of size $k = \alpha\sqrt{n}$, for some constant $\alpha > 1$, ensures high probability of intersection between read and write quorums, and yields a fault-tolerance of $\Omega(n)$, cf. [19]. Inserting in equation (1) yields

$$M_{prob} = 2c_n m(p+1)\alpha\sqrt{n} = \Theta\left(c_n mp\sqrt{n}\right) \tag{3}$$

where the expected number of rounds $c_n$ is given by $c_n = \log_Q(pm) + O(1)$ by Corollary 14.

For the strict case, $\Omega(n)$ fault-tolerance is only achieved when every set of size $\lfloor \frac{n}{2} \rfloor + 1$ is a quorum. Setting $k = \lfloor \frac{n}{2} \rfloor + 1$ in equation 2 gives

$$M_{str} = 2m(p+1)\left(\left\lfloor \frac{n}{2} \right\rfloor + 1\right) = \Theta\left(mpn\right). \tag{4}$$

Comparing the message complexity, we find that asymptotically $M_{prob} < M_{str}$ under some mild restrictions on $pm$ and $n$. Let $D$ be a constant such that the expression $\log_Q(pm) + O(1)$ in Corollary 14 is at most $\log_Q(pm) + D$.

**Theorem 15** *If $pm$ is large enough for Corollary 14 to hold and if $n$ is larger than $(2\alpha(\log_Q(pm)+D))^2$, then $M_{prob} < M_{str}$.*

**Proof.**

$$\begin{aligned} M_{prob} &= 2c_n m(p+1)\alpha\sqrt{n} && \text{by (3)} \\ &\leq 2(\log_Q(pm)) + D)m(p+1)\alpha\sqrt{n} && \text{by Cor. 14} \\ &= 2\alpha(\log_Q(pm) + D)\sqrt{n} \cdot m(p+1) \\ &< \sqrt{n} \cdot \sqrt{n} \cdot m(p+1) && \text{by the assumed lower bound on } n \\ &< M_{str} && \text{by (4).} \end{aligned}$$

■

# 8 Simulation Results for Expected Convergence Time

We have simulated systems of non-monotone (basic) and monotone random registers implemented using the algorithms from Sections 4 and 7.2 with a specific ACO. The simulation results shed some light on the following issues:

1. what is the convergence behavior in the original, non-monotone, case, and
2. what is the difference in the convergence behavior between the synchronous and asynchronous cases.

We took as our example application an all-pairs-shortest-path (APSP) algorithm presented in [27] and shown there to be an ACO. The vector $\mathbf{x}$ 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 $\mathbf{F}$ applied to $\mathbf{x}$ 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 length of the longest simple path in 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 chose this chain graph as our test input, because it has the largest $d$ among all connected graphs with the given number of vertices. This results in a larger number of pseudocycles and, thus, increased significance of our measurements. We limited the graph size in order to keep the running time of our simulator reasonable.

We simulated the execution of this APSP application over random registers, implemented with both the monotone 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,

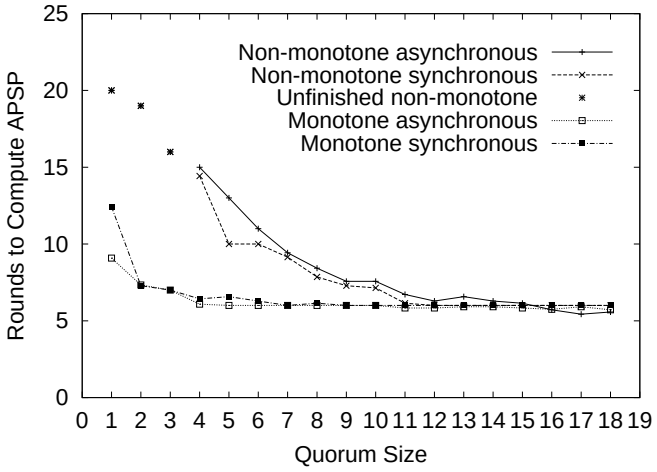


Figure 3: Simulation results: quorum size vs. rounds to converge.

so every read gets the value of the latest write, and the randomization in the quorum choice has no effect. We simulated both synchronous and asynchronous systems. The message delays in the synchronous system are all the same, whereas those in the asynchronous system are exponentially distributed.

We measured the number of rounds until every process computes the APSP of given input graph. Recall that a round finishes when every process completes at least one iteration of the while loop in Algorithm 2. Thus in the synchronous execution, a round consists of every process completing exactly one iteration of the while loop, whereas in the asynchronous execution, processes can complete various numbers of iterations of the while loop until one round is finished. At the end of each iteration of the while 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. [7, 28] for discussions of the issues involved in detecting termination for iterative algorithms.)

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 3[4].

The synchronous and asynchronous executions do not reveal much difference in the results. We conjecture that this is because the structure of a round causes the differences in the message delays, which are exponentially distributed, to average out. Each iteration of the while loop in Algorithm 2 involves 1190 round trip delays in series ($34^2 = 1156$ for the reads and 34 for the writes), where each round trip delay is the maximum of $k$ parallel round trips ($k$ is the

[4]The largest error bars we observed were 4 (which were in a few extreme cases) and in most cases ranged from 1 to 3; since this is fairly small, we decided not to clutter the plot with error bars.

quorum size). The phenomenon that asynchronous executions sometimes terminate faster than synchronous executions is explained by the different order of information propagation, i.e., it is possible that more information is available to the processes in asynchronous executions than synchronous executions after the same number of rounds has been finished.

The data indicates that the performance of the original algorithm is certainly worse than that of the monotone algorithm. In particular, for quorum sizes 1 to 3, the non-monotone simulation runs do not seem to converge in a reasonable amount of time. The open squares in Figure 3 indicate the number of rounds that elapsed in simulation runs that did not finish in a reasonable amount of time; thus they are *lower bounds* on the actual values for both synchronous and asynchronous executions.

The reason why runs with the quorum size $\leq 3$ did not complete is as follows. The Birthday Paradox tells us that random subsets of size $k$ in a universe of $n$ elements intersect with high probability if and only if $k = O(\sqrt{n})$. Therefore, for $n = 34$, the quorum size should be $k \approx 5$ or 6 to guarantee a good probability of intersection. Smaller $k$'s cause information to propagate very slowly between iterations, hence the slow convergence of the algorithm.

# 9 Discussion

We have proposed three specifications of randomized registers that can return out-of-date values, namely three probabilistic versions of a regular register. The first two specifications are non-monotone and the third one is monotone. We showed that all three 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 [27] of practical interest. We evaluated the performance of the algorithms experimentally as well as analytically, computing the expected convergence time 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. Malkhi et al. [19] mention building stronger kinds of registers, 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.

It would be interesting to investigate how to define, implement, and use other randomized data types. For example, randomized queues have been studied in [17].

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 the anonymous referees for many helpful comments, especially for bringing [6] to our attention and for pointing out an error in the earlier proof of Theorem 13. We thank Kathy Yelick for drawing our attention to reference [8], Nancy Amato, Idit Keidar, and Marcus Peinado for helpful conversations, Lyn Pierce for valuable comments on an earlier draft, and Andreas Klappenecker for insightful suggestions for the proof of Theorem 13.

## References

[1] S. Adve and M. Hill. Weak Ordering — A New Definition. In *Proceedings of the 17th Annual International Symposium 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. In *Proceedings of the 5th International 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, Feb. 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. Distributed Asynchronous Computation of Fixed Points. *Mathematical Programming,* Vol. 27, pages 107–120, 1983.

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

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

[16] H. Lee. Programs to compute the probability distribution of outdatedness for the probabilistic quorum implementation.
Available from `http://www.cs.du.edu/~hlee/Research/Programs/`.

[17] H. Lee and J. L. Welch. Randomized Shared Queues Applied to Distributed Optimization Algorithms. In *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC 2001)*, pages 587–598, Dec. 2001.

[18] D. Malkhi and M. Reiter. Byzantine Quorum Systems. *Distributed Computing*, Vol. 11, No. 4, pages 203–213, 1998.

[19] D. Malkhi, M. K. Reiter, A. Wool, and R. N. Wright. Probabilistic Quorum Systems. *Information and Computation*, Vol. 170, pages 184–206, 2001.

[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. *SIAM J. Computing*, Vol. 27, No. 2, pages 423–447, April 1998.

[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. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 61–70, 1998.

[25] A. N. Shiryayev. *Probability,* Springer-Verlag, 1984.

[26] W. Szpankowski and V. Rego. Yet Another Application of a Binomial Recurrence: Order Statistics. *Computing*, Vol. 43, pages 401–410, 1990.

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

[28] 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.

[29] H. Yu. Overcoming the Majority Barrier in Large-scale Systems. In *Proceedings of the 17th International Conference on Distributed Computing,* pages 352–366, 2003.