Specification, Implementation and Application of Randomized Regular Registers

Hyunyoung Lee
hlee@cs.tamu.edu

Jennifer L. Welch
welch@cs.tamu.edu

Technical Report 00-012
Department of Computer Science
Texas A&M University

March 31, 2000

Abstract

This paper presents a definition of a randomized regular register, shows that the definition is implemented by the probabilistic quorum algorithm of Malkhi, Reiter and Wright (1997) and shows how to program with such registers using the framework of Üresin and Dubois (1990). Consequently, existing iterative algorithms for a large class of problems (including solving systems of linear equations, finding shortest paths, etc.) will converge with high probability if executed in a system in which the shared data is implemented with registers satisfying the new condition. A modified definition is presented and its expected time for convergence is calculated and compared experimentally with that for the original definition.

1 Introduction

1.1 The Problem

Randomization is a powerful tool for developing software. As summarized in [11], randomized algorithms are often simpler and more efficient than deterministic algorithms for the same problem. Simpler code has the advantages of being less error-prone and shorter. Efficiency improvements are typically in the running time, number of messages, and/or degree of fault-tolerance. Randomization can even allow the *existence* of solutions to some problems for which there are no deterministic algorithms. The drawback of randomized algorithms is that absolute correctness cannot be guaranteed in all situations; typically a randomized algorithm has some, preferably minuscule, probability of either not terminating or running for a long time, or returning an incorrect answer.

Because of these advantages, randomized algorithms are a natural approach to develop distributed software that is more scalable, both in tightly coupled environments, such as parallel computers, and in more loosely coupled environments, such as the world wide web; and that must work in resource-poor environments, such as mobile computing.

The shared memory abstraction is attractive for writing distributed programs, since it is a style of programming that is familiar from the uniprocessor case and supports good software development practice. Evidence for this is the volume of work in the area of distributed shared memory systems, which provide the illusion of shared read-write variables on top of a distributed system in which processes actually communicate by message passing.

Since randomized implementations will sometimes either fail to terminate or return a wrong answer, the specification of the shared variable provided will need to be modified in order to realize the advantages of randomization. We are interested in these questions:

- What are appropriate formal specifications for shared data structures that have some probability of error? How should the error be specified?

- What are efficient randomized implementations of the shared data structures so specified?

- How can users program safely with this kind of shared object? Are there programming rules that can be followed to mitigate the potential damage caused by the errors? In particular, what kinds of distributed applications can tolerate the weaker semantics provided by randomized data structures, and can they be characterized?

Answers to these questions will help characterize the tradeoffs between ease of programming and performance, when randomized data structures are used. For example, some deterministic algorithms for implementing shared registers use broadcast to keep the replica of shared data up-to-date (e.g., [6]), which incurs a great deal of message overhead in non-broadcast networks. Substituting the randomized register implementation studied in this paper results in fewer messages, but seems to require a special style of programming. 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].

1.2 Prior 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 interested in, 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 [20, 19, 10], discussed below.

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

Malkhi et al. [18, 17] have proposed a probabilistic quorum algorithm to implement a read-write variable on top of a message passing system. Each read is translated into messages to a subset ("quorum") of the processes to obtain the latest value, and each write is translated into messages to a quorum of the processes 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 message-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 [18]). However, to make them 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 synchronization mechanisms called combining funnels [20] and diffracting trees [19] on top of simpler shared objects. In these algorithms, the effect of randomization is on the performance; wrong answers are never returned.

Czumaj et al. [10] have implemented PRAM models on top of 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. [18] references two other PRAM simulations that use randomized data structures.

What kind of distributed applications can tolerate shared objects subject to errors? Our intuition was that iterative convergent algorithms could 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 [9]. Subsequently a number of authors refined this work (cf. Chapter 7 of [7] for an overview). Üresin and Dubois [21] give a general necessary and sufficient condition on the function for convergence. Essentially the same convergence theorem is presented in Chapter 6 of [7]. This class of functions includes solutions to many practical applications, including solving systems of linear equations, finding shortest paths, and network flow [7].

1.3 Our Contribution

To the best of our knowledge, little existing work has focused on defining the semantics of distributed data structures that sometimes give wrong answers, or on trying to characterize classes of applications that can tolerate such data structures.

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

Then we show that there exists at least one efficient implementation of our definition, namely the probabilistic quorum algorithm of [18, 17].

Thirdly, we show how registers satisfying our definition can be used to program iterative algorithms in the framework presented by Üresin and Dubois [21]. The implication is that we can use existing iterative algorithms for a large class of problems (including solving systems of linear equations, finding shortest paths, etc.) 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.

Finally, 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. Relationships between the modified and the original specifications are clarified with the help of a simulation. The convergence rates of iterative algorithms have been studied by, e.g., [21, 7]; 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.

The rest of this paper is organized as follows. Section 2 presents our system model and definition of a random regular register. Section 3 shows that the probabilistic quorum algorithm of [18, 17] implements our definition. Section 4 reviews the framework for the iterative algorithms from [21], and shows how those conditions are satisfied by our definition of random regular registers. In Section 5, we describe a variation of our definition, show its expected convergence behavior, and present the simulation results. Section 6 concludes this paper with a discussion of further research.

2 Specifying a Random Regular 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 can apply to any kind of implementation.

2.1 Read/Write Registers

A read/write **register** X shared by n processes supports two operations, read and write. Each operation has an invocation and a response. $\text{Read}_p(X)$ is the invocation by process p of a read, $\text{Write}_p(X, v)$ is the invocation by p of a write of the value v, $\text{Return}_p(X, v)$ is the response to p's read invocation which returns the value v, and $\text{Ack}_p(X)$ is the response to p'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 [16].

Defining a probabilistic version of regularity requires specifying a probability space. We do so in the next few subsections.

2.2 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 randomized 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 P of n 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.

2.3 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, only the trigger.

Let RAND be the set of all n-tuples of the form $\langle R^1, \ldots, R^n \rangle$ where each R^i is an infinite sequence of integers in $\{0, \ldots, D\}$. D indicates the range of the random numbers. R^i describes the sequence of random numbers available to process i in an execution — R^i_j is the random number available at step j. Call each element in RAND a **random tuple**.

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

- the current state in the first step of process p is p's initial state, for all $p \in P$;

- the current state in the i-th step of process p is the same as the new state in the $(i-1)$-st step of p, for all $p \in P$ and all $i > 1$;

- the trigger in σ_i equals $A(\sigma_1 \ldots \sigma_{i-1})$, for all $i \geq 1$ (the trigger is chosen by the adversary);

- the random number in σ_i equals R^p_i, where p is the process in σ_i's trigger (the random number comes from $\mathcal{R}$, not the adversary).

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

- 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 p only if a response by p follows the latest preceding invocation for p 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 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.

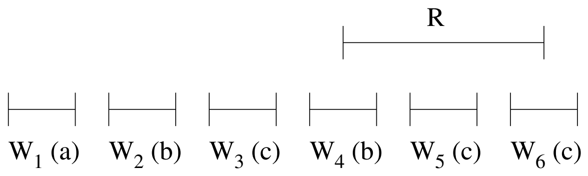

Figure 1: Diagram for definition of reads from.

2.4 Random Regular Registers

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.

A system is said to implement a **random regular 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 a write that starts before the read ends, 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.

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 $\mathcal{R}$. Consider 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. This set of executions 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.

3 Implementing a Random Regular Register

In this section, we show that the probabilistic quorum algorithm presented by Malkhi et al. [18, 17] implements a random regular register. Their algorithm works in a reliable, asynchronous message passing environment.

The following specializations are needed to the general model given in Section 2: 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 P, the set of all processes, of size k (the **quorum size**). We have simplified the read/write register implementation algorithm from [17] to assume only one writer. Each process keeps a local replica of the register to be implemented. A timestamp is associated with the replica. To perform a read, a process queries a quorum and returns the value resulting from the query with the largest timestamp. To perform a write, 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 P) [18].

Each process p uses the following local variables:

- val_p : holds the value of its replica of the (logical) shared register. Initially val_p holds the initial value of the shared register.
- ts_p : holds the timestamp associated with the value in val_p. Initially $ts_p = 0$.
- $rval_p$: holds the most recent value received from the processes in the current quorum.
- rts_p : holds the received timestamp associated with the value in $rval_p$.
- $numresp$: holds the number of responses that p has obtained so far from the current quorum.

In addition, the unique writer 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 process to perform when each event occurs is presented as Algorithm 1.

Theorem 1 *The probabilistic quorum algorithm implements a random regular 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 read from infinitely often is 0, we will show that the probability that at least one of the replicas in W's quorum survives ℓ subsequent writes goes to 0 as ℓ increases without bound.

Let k be the quorum size.

$$\begin{aligned}
&\text{Pr[at least one replica from } W\text{'s quorum survives } \ell \text{ subsequent writes]} \\
&\quad \leq \; k \cdot \text{Pr[a specific replica } r \text{ from } W\text{'s quorum survives } \ell \text{ subsequent writes]} \\
&\quad = \; k \cdot \text{Pr[} r \text{ is not in the quorum of any of the } \ell \text{ subsequent writes]} \\
&\quad = \; k \cdot \text{Pr[(} r \text{ is not in } Q_1) \bigcap (r \text{ is not in } Q_2) \bigcap \ldots \bigcap (r \text{ is not in } Q_\ell)]
\end{aligned}$$

when Read$_p(X)$ occurs at process p: // invocation for the Read

    pick a random quorum Q := $\{q_1, \ldots, q_k\}$
    rts_p := 0
    $numresp$:= 0
    send QUERY messages to $q_1, \ldots, q_k$

when a QUERY message is received by q_i from p:

    send a VALUE(val_{q_i},ts_{q_i}) message to p

when a VALUE(v,t) message is received by p from q_i:

    $numresp$++
    if $t > rts_p$ then $rval_p$:= v; rts_p := t
    if $numresp$ = k then Return$_p(X, rval_p)$ // response for the Read

---

when Write$_w(X, v)$ occurs at the writer process w: // invocation for the Write

    wts_w++
    pick a random quorum Q := $\{q_1, \ldots, q_k\}$
    $numresp$:= 0
    send UPDATE(v,wts_w) messages to $q_1, \ldots, q_k$

when an UPDATE(v,t) message is received by q_i from w:

    val_{q_i} := v
    ts_{q_i} := t
    send an ACK message to w

when an ACK message is received by w from q_i:

    $numresp$++
    if $numresp$ = k then Ack$_p(X)$ // response for the Write

Algorithm 1: Probabilistic Quorum Algorithm [18, 17] where Q_i is the quorum of the i-th subsequent write

$$= \quad k \cdot \prod_{i=1}^{\ell} \Pr[r \text{ is not in } Q_i] \qquad \text{since quorums are chosen independently}$$

$$= \quad k \cdot \left(\frac{n-k}{n}\right)^{\ell} \qquad \text{since } n-k \text{ out of the } n \text{ replicas are not in a given quorum.}$$

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

4 Iterative Programs Using Random Regular Registers

4.1 A Framework for Iterative Algorithms

Üresin and Dubois [21] 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 their framework and point out that random regular 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 regular registers model.

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 [21].

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 $S_1, \ldots, S_m$.

A **schedule** is a collection of a function *change* from N (the natural numbers) to $2^{\{1,\ldots,m\}}$, and m functions *view*$_i$, $1 \leq i \leq m$, from N to N. The schedule produces a sequence of **updates** to the vector, detailed below. The value of *change*(k) indicates which 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.

A schedule is called **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 [A1] through [A3] are equivalent to the following condition (which will be used in Section 5): 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.

A schedule is combined with a function **F** and an initial vector **i** to produce an **asynchronous iteration**, which is 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).

The function **F** is called an **asynchronously contracting operator** 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.

[C2] If **x** is in $D(K)$, then **F(x)** is in $D(K+1)$, for all K.

Theorem 2 *[21] If* **F** *is an asynchronously contracting operator on* $D(0), D(1), \ldots$*, then in every asynchronous iteration starting with initial data* $\mathbf{i} \in D(0)$ *using a pseudoperiodic schedule, the computed vector eventually converges to the fixed point of* **F**.

Their proof shows that after all the components are updated in the Kth 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.

4.2 Using Random Regular Registers

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

Code for each process $p \in P$:
```
while true do
    for i := 1 to m do x_i := read(X_i)
    y := F(x)
    for each j such that p is responsible for computing X_j do
        write(X_j, y_j) // an update
```

Algorithm 2: Asynchronous Iteration Using Random Regular Registers

An asynchronous iteration using random regular registers corresponds to an execution of the following algorithm. In this algorithm, responsibility for updating the m components of the vector **x** is partitioned among the n processes. For each i, $1 \leq i \leq m$, component i of **x**, denoted x_i, is held in a shared variable X_i, which is a random regular register. Recall that **i** is the initial vector on which the iterative algorithm is to compute. Each X_i is initialized to contain the value of component i of **i**. The code is given as Algorithm 2.

Condition [A1] is satisfied in any execution thanks to part [R2] of the definition of a random regular 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 regular register. Therefore, we have proved:

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

5 Monotone Random Regular Register

In this section, we define a variation of a random regular 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.

The goal of adding this condition was to be able to analyze the expected convergence time of an iterative algorithm in the [21] framework. Our approach for doing so required us to make an additional, more technical, requirement on the register, in terms of its probabilistic behavior.

This section gives the formal definition for the modified register, shows how to implement the definition with a simple addition to the probabilistic quorum algorithm [17, 18], and then proves that if all the registers satisfy the definition, then the expected convergence time of an iterative algorithm is bounded.

5.1 Definition

A random regular register is **monotone** if it satisfies the following for every adversary:

[R4] In every execution (of the adversary), if read R by process p_i follows read R' by process p_i then R does not read from a write that precedes the write from which R' reads.

We now state the additional technical property we require. 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.

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

The probability space for [R5] is all writes in all executions of the adversary.

5.2 Implementation

To implement condition [R4] in the probabilistic quorum algorithm, 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 processes with quorum size k implements a monotone random regular register with $p = 1 - \binom{n-k}{k}/\binom{n}{k}$.*

Proof. Condition [R4] is clearly true. The rest of the proof shows that [R5] holds.

Choose a particular write W in a particular execution and a particular process p_i. W or a later write will be read from by p_i if W is followed by a read whose quorum overlaps W's quorum. (There are other scenarios in which p_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 $(1-p)^{r-1} \cdot p$, since quorums are chosen independently. ∎

5.3 Expected Convergence Time

In this section we show an upper bound on the expected number of rounds required per pseudocycle (cf. Section 4.1), if the vector components are implemented with a monotone random regular register.

A **round** is 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.)

Theorem 5 *In every execution of Algorithm 2 using monotone random regular registers with constant* p*, the expected number of rounds per pseudocycle is at most* $\frac{1}{p}$.

Proof. Consider any adversary A and any finite execution e of A that has just completed pseudocycle i, for any $i \geq 0$. We will calculate how many rounds are needed on average for pseudocycle $i+1$ 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 $p \in P$, p must read from a write that is, or follows, the first write to X_j in pseudocycle i, before pseudocycle $i+1$ can end. Once this read occurs, by [R4] all subsequent reads by p_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 5.1.

$$\begin{aligned} EY &= \sum_{r=1}^{\infty} r \cdot \Pr[Y = r] \qquad \text{by definition of expectation} \\ &\leq \sum_{r=1}^{\infty} r \cdot (1-p)^{r-1} \cdot p \qquad \text{by [R5]} \\ &= \frac{1}{p} \qquad \text{by algebra.} \end{aligned}$$

■

Corollary 6 *Let* **F** *be an asynchronously contracting operator that converges in* M *pseudocycles. The expected number of rounds taken by any complete execution of Algorithm 2 using monotone random regular registers with constant* p *is at most* M/p.

We now compute the value of $1/p$ for the monotone probabilistic quorum algorithm with n processes and quorum size k. Proposition 3.2 in [18] 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* $\frac{1}{1-(\frac{n-k}{n})^k}$.

As a sanity check, note that if $k > n/2$, so that all quorums overlap, the expression in the corollary is less than 2. Certainly even in the worst case, there is one round per pseudocycle.

5.4 Simulation Results

We simulated a system of both non-monotone and monotone random regular registers implemented with the algorithms from Sections 3 and 5.2 and then ran some experiments with a specific asynchronously contracting operator on top. We hoped to get some insight into two issues that eluded us analytically:

1. how much of an over-estimate is the upper bound derived in Corollary 7 on the expected number of rounds per pseudocycle for the monotone case, and

2. what is the convergence behavior in the original, non-monotone, case.

We took as our example application an all-pairs-shortest-path algorithm presented in [21] and shown to be an asynchronously contracting operator. The vector **x** to be computed is two-dimensional, n by n, where n is the number of nodes in the graph. Initially each x_{ij} contains the weight of the edge from node i to node 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

$$\min_{1 \leq k \leq n} \{x_{ik} + x_{kj}\}.$$

In the asynchronous iteration, there are n processes, and process p_i is responsible for updating the i-th row vector of **x**, $1 \leq i \leq n$. The worst-case number of pseudocycles required for convergence of **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 18 nodes that is a chain, with node 1 the sink and node 18 the source. Each edge has weight 1. For this graph, $\lceil \log_2 17 \rceil = 5$ pseudocycles are required for convergence.

We simulated the execution of this APSP application on top of random regular registers, implemented with both the modified and original probabilistic quorum algorithm, over a range of quorum sizes, from 1 to 10. Once the quorum size is at least 10, all quorums overlap (since $n = 18$), so every read gets the value of the latest write, and the randomization in the quorum choice has no effect. Thus there is no point in considering additional quorum sizes beyond 10. We simulated a synchronous system (experiments with asynchronous systems are currently underway).

The upper bounds on the expected number of rounds until convergence for the various quorum sizes are given in Table 1. The first row holds the quorum sizes, the second row the upper bound on the expected number of rounds per pseudocycle, and the third row the upper bound on the total expected number of rounds, which is five times the second row.

For each quorum size, three runs were performed and the number of rounds required for convergence was noted. The average of these three values was then plotted; see Figure 2. (Our experience indicated that the values had little variation, so in this preliminary data we only have three runs; we are working on obtaining more data.)

The discrepancy between the calculated upper bound and the experimental value is quite large for very small quorums (90 vs. 6.3), but it eventually disappears as the quorum size increases. One source of the overestimate is in the proof of Theorem 5, where we did not take into account the

Table 1: Upper Bounds on Expected Number of Rounds per Pseudocycle and Total with Quorum Size k

k	1	2	3	4	5	6	7	8	9	10
per p.c.	18.00	4.636	2.454	1.492	1.159	1.043	1.019	1.002	1.000	1.000
total	90.00	23.18	12.27	7.46	5.795	5.215	5.095	5.01	5.000	5.000

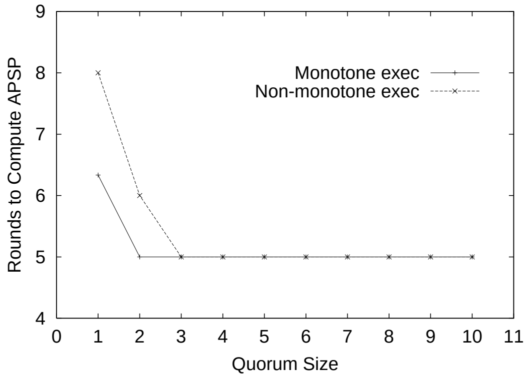

Figure 2: Simulation Result: Quorum Size vs. Round Numbers to Converge

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, at least in the synchronous case, is not much worse than that of the monotone algorithm, and is less than the computed upper bound for the monotone case.

In both cases, notice how a small quorum (say 3) is as good as a large one (large enough to be deterministic). This is in line with the intuition behind the original probabilistic quorum paper [18].

6 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 [17, 18]. Furthermore, our specifications can be used to implement a large class of iterative algorithms [21] of practical interest.

A number of challenging directions remain as future work. One of them is how to design more powerful read/write registers and other data types in our framework. [18] mentions being able to build stronger kinds of registers, such as multi-writer and atomic, out of the regular 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.

The issue of fault tolerance of 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 [9] and Nancy Amato for helpful conversations.

References

[1] S. Adve and M. Hill. Weak Ordering — A New Definition. *Proc. 17th Annual Int'l Symp. on Computer Architecture*, pp. 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] H. Attiya and J. Welch. Sequential Consistency vs. Linearizability. *ACM Trans. Computer Systems*, vol. 12, no. 2, pp. 91–122, May 1994.

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

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

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

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

[18] 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.

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

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

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

[22] 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.