A RANDOMIZED MEMORY MODEL AND ITS APPLICATIONS IN DISTRIBUTED COMPUTING

A Dissertation

by

HYUNYOUNG LEE

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2001

Major Subject: Computer Science

A RANDOMIZED MEMORY MODEL AND ITS APPLICATIONS IN DISTRIBUTED COMPUTING

A Dissertation

by

HYUNYOUNG LEE

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Approved as to style and content by:

December 2001

Major Subject: Computer Science

ABSTRACT

A Randomized Memory Model and Its Applications

in Distributed Computing. (December 2001)

Hyunyoung Lee, B.S., Ewha University;

M.S., Ewha University;

M.A., Boston University

Chair of Advisory Committee: Dr. Jennifer L. Welch

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

In this dissertation, we propose a shared memory framework for distributed algorithms, in which the implementation of the shared memory can be randomized. In particular, read operations of read/write registers can return out-of-date values, and dequeue operations of queues can return out-of-order values with some small probability of lost values.

We define new conditions, which constrain this error probability, such that interesting classes of popular algorithms will work correctly when implemented over such *randomized data structures.* At the same time, our conditions are sufficiently weak to allow certain kinds of probabilistic replicated systems to implement such memory.

It is shown by Malkhi et al. that these replicated systems have very attractive properties, such as high scalability, availability and fault tolerance .

As we will show, using this random memory model can result in improved load, availability in the face of server crashes, and message complexity, but seems to require a special style of programming.

We consider two interesting classes of algorithms as applications for our randomized data structures: a class of iterative algorithms in the framework of Üresin and Dubois as an application for random registers and a class of randomized optimization algorithms by Aldous and Vazirani for random queues. Furthermore, we explore an application of our randomized data structures to mobile computing where a mobile computing entity can rely on some partial or out-of-date data to reconfigure its computing environment in response to physical movement.

To my parents

ACKNOWLEDGMENTS

First, and above all, I would like to convey my deep thankfulness and appreciation to my advisor Dr. Jennifer L. Welch. Without her insightful ideas, this dissertation could not have happened. She is a coauthor of Chapters 2 and 3. Besides her precious and ample advice, directions, supports, and encouragement as the Ph.D advisor, she was always accessible, reliable and dependable. She will always be a role-model for me to follow through my future career and life overall.

The other members of my dissertation committee, Dr. Jianer Chen, Dr. Vivek Sarin, and Dr. Narasimha Reddy, gave me perspectives of looking into similar problems from various directions.

The idea of Chapter 4 was enhanced by Dr. Nitin Vaidya's helpful comments and suggestions. I wish to thank him for being on my dissertation committee earlier.

I thank Dr. Stephen Crouse for serving as the Graduate Counsel Representative on my committee.

It has been a pleasure studying and working in the Department of Computer Science at Texas A&M University. I would like to use this opportunity to thank all the people in the department including faculty, staff, and colleagues who have contributed to this nice, friendly, and supportive environment.

Finally, my deepest feelings and thankfulness go to my family, especially my devoted parents who were always there when I needed them. I could not have done it without the great love and support I received from my parents.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

A. Overview

Randomization is a powerful tool in the design of algorithms. As summarized in [31, 16], randomized algorithms are often simpler and more efficient than deterministic algorithms for the same problem. Simpler algorithms have the advantages of being easier to analyze and implement. A well known example is the factoring problem, for which simple randomized polynomial-time algorithms are widely used, while no corresponding deterministic polynomial time algorithm is known. Randomized algorithms have a failure probability, which can typically be made arbitrarily small and which manifests itself either in the form of incorrect results (Monte Carlo algorithms) or in the form of unbounded running time (Las Vegas algorithms).

In this dissertation, we define a shared memory framework for distributed algorithms, in which the implementation of the shared memory can be randomized. In particular, read operations of read/write registers can return out-of-date values, and dequeue operations of queues can return out-of-order values with some small probability of lost values. We define new conditions, which constrain this error probability, such that interesting classes of popular algorithms will work correctly when implemented over our *randomized data structures*. At the same time, our conditions are sufficiently weak to allow certain kinds of probabilistic replicated systems to implement such memory. These replicated systems have very attractive properties, such as high scalability, availability and fault tolerance [30].

Our focus is to show that randomization may provide a more efficient way to

The journal model is *IEEE Transactions on Computers*.

implement distributed shared memory, not to show that using randomized distributed shared memory is more efficient than message passing. Evidence of advantages of using the shared memory abstraction instead of message passing is the volume of work in the area of distributed shared memory systems (cf. Chap 9 of [7] for an overview).

The main problem in replicated systems is to maintain consistency among the replicas. Quorum systems try to maintain consistency by defining collections of subsets of replicas (*quorums*) and having each operation select and access one quorum from the collection. Traditional, or *strict*, quorum systems require all quorums in the collection to intersect pairwise. Malkhi et al. [30] introduce the notion of a *probabilistic* quorum system, in which pairs of quorums only need to intersect with high probability. Malkhi et al. show that this relaxation leads to significant performance improvements in the load of the busiest replica server and the availability of the quorum system in the face of replica server crashes. We show that our definition of randomized memory model captures similar properties, by accommodating the probabilistic quorum system as one possible implementation.

As we will show, using this random memory model can result in improved load, availability in the face of server crashes, and message complexity, but seems to require a special style of programming. Apparently there is a tradeoff between ease of programming and performance, when randomized data structures are used. These results are somewhat analogous to the situation with "weak", or "hybrid", consistency conditions, which can be implemented quite efficiently but require the application programs to be data-race-free [1, 5].

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

We consider two interesting classes of algorithms as applications for our randomized data structures: a class of iterative convergent algorithms in the framework of Üresin and Dubois [38] as an application for random registers and a class of randomized optimization algorithms by Aldous and Vazirani [4] for random queues.

Furthermore, we explore an application of our randomized data structures to mobile computing where a mobile computing entity can rely on some partial or out-of-date data to reconfigure its computing environment in response to physical movement.

B. Contents

The dissertation is organized as follows:

Chapter II presents our work on a *random register*. We define a random read-write register that sometimes returns out-of-date values, show that the definition is implemented by the probabilistic quorum algorithm of Malkhi et al. [30, 29], show how to program with such registers using the framework of Üresin and Dubois [38], and discuss the consequences in terms of convergence time and message complexity. The material in this chapter appears in the *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)* [25]. An earlier version appears in the *Proceedings of 19th ACM Symposium on Principles of Distributed Computing (PODC)* [24].

Chapter III discusses our work on a *random queue*. We present a specification of a randomized shared queue that can lose some elements or return them out of order, show that the specification can be implemented with the probabilistic quorum algorithm of [30, 29], and analyze the behavior of this implementation. Distributed algorithms that incorporate the producer-consumer style of interprocess communication are candidate applications for using random shared queues in lieu of the message queues. The behavior of an application – a class of combinatorial optimization algorithms – when it is implemented using random queues is analyzed. The material in this chapter appears in the *Proceedings of the 12th International Symposium on Algorithms and Computation (ISAAC)* [27]. An earlier version appears in the *Proceedings of 20th ACM Symposium on Principles of Distributed Computing (PODC)* [26].

Chapter IV applies the randomized data structures to mobile computing. We abstract the information dissemination problem for mobility management in mobile ad hoc networks (MANETs) as distributed shared variables. We apply a variation of the probabilistic quorum algorithm by Malkhi et al. to implement the shared information database, and compare the probabilistic quorum based implementation with the strict quorum based implementations of [20] by way of simulations. Chapter IV explores experimentally a variation of the random register of Chapter II for MANETs. The similarities are:

- The interface is same, that is, the interface of query (resp., update) is the same as that of read (resp., write). Query and update are used for historical reasons to be consistent with prior work.
- Values returned were previously written.

The differences are that the liveness properties, termination and the probability of reading stale values, depend on mobility patterns. An open question is how to specify and characterize mobility patterns that allow precise statements of liveness. Chapter IV shows simulation results that could support such formal development.

Chapter V concludes the dissertation with a discussion of further research.

CHAPTER II

APPLICATIONS OF PROBABILISTIC QUORUMS TO ITERATIVE ALGORITHMS

A. Introduction

In this chapter, we propose a formal definition of a random read-write register. The consistency condition provided by our definition is a probabilistic variation on the concept of regularity from Lamport's paper [23].

We show that our definition of a random register can be implemented by the probabilistic quorum algorithm of [30, 29], which has several advantageous properties such that the load on the busiest replica server is limited and the availability in the face of server crashes is high.

Next we show how registers satisfying our definition can be used to program iterative algorithms in the framework presented by Üresin and Dubois [38]. 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 availability from the underlying register implementation.

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

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

Section B describes related work. In Section C, we present our system model and definition of a random register. Section D shows that the probabilistic quorum algorithm of [30, 29] implements our definition. Section E reviews the framework for the iterative algorithms from [38], and shows how those conditions are satisfied by our definition of random registers. In Section F, we describe a variation of our definition, show its expected convergence behavior, and identify situations in which it has superior message complexity. Section G presents our simulation results.

B. Related Work

A number of consistency conditions for shared memory have been proposed over the years, including safety, regularity and atomicity [22, 23], sequential consistency [21], linearizability [18], causal consistency [3] and hybrid consistency [6]. These definitions have all been deterministic with little or no regard to possible errors.

Afek et al. [2] and Jayanti et al. [19] 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 [37, 36, 15], discussed below.

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

Malkhi et al. [30, 29] 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 [30] and reviewed in Section 4). However, to make probabilistic quorums usable by programmers, a more complete semantics of the register which they implement must be given, together with techniques for programming effectively with them.

Shavit and Zemach have implemented novel randomized synchronization mechanisms called combining funnels [37] and diffracting trees [36] over simpler shared objects. In these algorithms, the effect of randomization is on the performance; wrong answers are never returned.

Czumaj et al. [15] 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. Malkhi et al. [30] reference two other PRAM simulations that use randomized data structures.

In this chapter, we show that one class of iterative convergent algorithms can handle infrequent out-of-date values. The first analysis of the convergence of iterative functions when the input data can be out of date was by Chazan and Miranker [12]. Subsequently a number of authors refined this work (cf. Chapter 7 of [8] for an overview). Üresin and Dubois [38] give a general necessary and sufficient condition on the function for convergence. Essentially the same convergence theorem is presented in Chapter 6 of [8]. This class of functions includes solutions to many practical applications, including solving systems of linear equations, finding shortest paths, and network flow [8]. The convergence rates of iterative algorithms have been studied in [8, 39]; 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.

C. 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 chapter is a message-passing one, we would like the *specification* to be implementation-independent, so that it could apply to any kind of implementation.

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 [23].

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

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

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

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

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

- 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. 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.

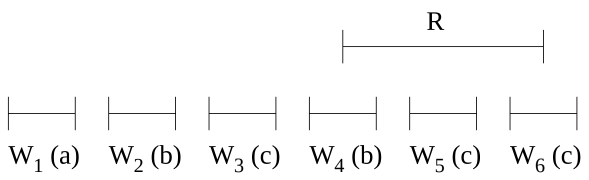


Fig. 1. Diagram for Definition of Reads From

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,

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

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

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

Notice that this is a kind of "worst-case" probabilistic definition as the probabilistic

[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 chapter.

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.

D. Implementing a Random Register with Probabilistic Quorums

In this section, we show that the probabilistic quorum algorithm presented by Malkhi et al. [30, 29] implements a random register. For simplicity, we first assume an asynchronous reliable message passing environment and no process failure.

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

The algorithm uses the notion of a **quorum**, which is a subset of the set of all replicas, of size k (the **quorum size**). We have simplified the read-write register algorithm from [29] to assume only one writer and absence of failures. The shared register X is replicated over n servers. This replicated server system is used by p processes through the shared register subsystem associated with each process. Each server keeps a local replica of the register to be implemented. A timestamp is associated with the replica. To perform a read, the shared register subsystem queries a quorum and returns the value with the largest timestamp resulting from the query. To perform a write, the shared register subsystem for the writer causes the replicas in a quorum to be updated with the new value and its new timestamp. Each quorum is chosen randomly with uniform distribution from the set of all possible quorums (all k-subsets of the set of all replicas) [30].

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 shared register subsystem 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 shared register subsystem for 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 replica server and each shared register subsystem to perform when each event occurs is presented as Algorithm 1.

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

```
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
```

Fig. 2. Algorithm 1: (Simplified) Probabilistic Quorum Algorithm [30, 29] to Implement Shared Register X

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.

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

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: availability and load.

Availability is a measure of fault tolerance. The **availability** 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 [34]. To achieve high availability of $\Omega(n)$, the smallest quorum size of the strict quorum system must be $\Theta(n)$. This property is satisfied by the *majority* quorum system, in which every quorum has size $\lfloor \frac{n}{2} \rfloor + 1$ [34].

Malkhi et al. [29] 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, 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 [32] to be the minimal access probability of the busiest server, minimizing over all strategies for choosing the quorums. In [32] 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. [30] 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 [32] showed that strict quorum systems have a trade-off between availability and load such that any strict quorum system with optimal load of $\Theta\left(\frac{1}{\sqrt{n}}\right)$ has only $O\left(\sqrt{n}\right)$ availability. Malkhi et al. [30] showed that *using probabilistic quorums breaks this trade-off* and achieves simultaneously high availability of $\Theta(n)$ and optimal load of $\Theta\left(\frac{1}{\sqrt{n}}\right)$.

[2]One mechanism would be to use failure detector to eliminate all faulty servers, and choose quorum at random among all quorums that do not include a faulty server.

E. Iterative Programs Using Random Registers

1. A Framework for Iterative Algorithms

Üresin and Dubois [38] 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 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 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 [38].

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

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

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

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

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

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

- $\mathbf{x}(0) = \mathbf{i}$,
- for each $k \geq 1$ and all i, $1 \leq i \leq m$, $x_i(k)$ equals $x_i(k-1)$ if i is not in $change(k)$, and equals $F_i(x_1(view_1(k)), \ldots, x_m(view_m(k)))$ if i is in $change(k)$.

Üresin and Dubois show that [A1] through [A3] are equivalent to the following condition (which will be used in Section F): there exists an increasing infinite sequence of integers $\varphi(0) = 0, \varphi(1), \varphi(2), \ldots$, where updates $\varphi(K)$ through $\varphi(K+1)-1$ comprise **pseudocycle** K, such that

[B1] each component of the vector is updated at least once in each pseudocycle, and

[B2] during each update in pseudocycle $K \geq 1$, the view of each component i is a value that was updated in pseudocycle $K-1$ or later.

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

The function $\mathbf{F}$ is called an **asynchronously contracting operator** (ACO) if there is a sequence of sets $D(0)$, $D(1)$, $D(2)$, ..., where $D(0) \subseteq S$, satisfying the following conditions:

[C1] For each K, $D(K)$ is the Cartesian product of n sets $D_1(K), \ldots, D_m(K)$.

[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 II.2 *[38] If $\mathbf{F}$ is an ACO on $D(0), D(1), \ldots$, then every update sequence 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 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.

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 C, then Theorem II.2 is true with probability 1.

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. Recall that $\mathbf{i}$ is the initial vector on which the iterative algorithm is to com-

```
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
        $\mathbf{y} := \mathbf{F}(x_1, \ldots, x_m)$ // 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
```

Fig. 3. Algorithm 2: Asynchronous Iteration Using Random Registers

pute. Each X_j is initialized to contain the value of component j of $\mathbf{i}$. The code is given as Algorithm 2.

Theorem II.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 update sequence extracted from an execution satisfies [A1], [A2] and [A3] with probability 1. Then Theorem II.2 will hold with probability 1.

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

F. Monotone Random Register

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

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

The goal of adding this condition was to be able to analyze the expected convergence time of an iterative algorithm in the [38] framework. Our approach for doing so required us to make an additional, more technical, requirement on the register, in terms of its probabilistic behavior.

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 after a write W until W or a later write is read from by that process. The intuition is that q is the probability of "success" for a read; the probability that r reads are required is (at most) the probability that $r-1$ reads fail and then the r-th read succeeds.

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

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

2. Implementation

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

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

Proof. 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 i. W or a later write will be read from by i if W is followed by a read whose quorum overlaps W's quorum. (There are other scenarios in which i can obtain a value later than W, but we do not consider them in this analysis.)

The probability of a read R's quorum not overlapping W's is $\binom{n-k}{k}/\binom{n}{k}$, since there are $\binom{n}{k}$ possible choices for R's quorum and there are $\binom{n-k}{k}$ choices for quorums that do not overlap W's.

The probability that $Y = r$ is at most the probability that $r - 1$ reads have quorums that do not overlap W's and then the r-th read's quorum does overlap W's. The latter probability is $(1 - q)^{r-1} \cdot q$, since quorums are chosen independently. □

3. Expected Convergence Time for an ACO

In this section we show an upper bound on the expected number of rounds required per pseudocycle (cf. Section 1) in the execution of an ACO, if the vector components are implemented with a monotone random register.

A **round** is a minimal length (contiguous) subsequence of an execution in which each process 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 II.5 *In every execution of Algorithm 2 using monotone random registers with parameter q, the expected number of rounds per pseudocycle is at most $\frac{1}{q}$.*

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

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

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

will be at least as recent.

The required number of rounds is at most the random variable Y, as defined for [R5] in Section 1.

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

$\square$

Corollary II.6 *Let* **F** *be an ACO that converges in M pseudocycles. The expected number of rounds taken by any complete execution of Algorithm 2 using monotone random registers with parameter q is at most M/q.*

We now provide an upper bound on the value of $1/q$ for the monotone probabilistic quorum algorithm with n replicas and quorum size k. Proposition 3.2 in [30] implies that $\binom{n-k}{k}/\binom{n}{k} \leq (\frac{n-k}{n})^k$. Thus we have:

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

4. Expected Message Complexity for an ACO

In this section, we compare the expected message complexity per pseudocycle when executing an ACO for two implementation strategies of the vector components. One implementation strategy is the monotone probabilistic quorum algorithm. The other strategy consists of strict quorum systems, in which all quorums overlap. We show that although the number of rounds required for convergence is greater for the probabilistic case, there are some important situations in which the message complexity is smaller. 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 $M_{prob}(k)$ be the expected number of messages sent per pseudocycle with the monotone probabilistic quorum implementation, and $M_{str}(k)$ be that with a strict quorum implementation, where the parameter k indicates the size of the quorums. Inspecting the code shows that the total number of messages sent per round is $2pmk + 2mk$. Each of the p processes reads each of the m vector components once, and each of the m vector components is written once. Each operation takes $2k$ messages. Then

$$M_{prob}(k) = 2c_n m(p+1)k \tag{2.1}$$

where c_n is the expected number of rounds per pseudocycle. And

$$M_{str}(k) = 2m(p+1)k \tag{2.2}$$

since a strict quorum system uses one round per pseudocycle.

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

We first consider quorum systems with high availability of $\Omega(n)$. For the probabilistic case, we set $k = \Theta(\sqrt{n})$, which ensures high probability of intersection between read and write quorums and also gives $\Omega(n)$ availability [30]. Plugging into Eqn. 1 gives

$$M_{prob} = \Theta\left(2c_n m(p+1)\sqrt{n}\right) = \Theta\left(mp\sqrt{n}\right) \tag{2.3}$$

since $1 < c_n < 2$ for all n when the quorum size is $\sqrt{n}$ (cf. Corollary II.7).

For the strict case, $\Omega(n)$ availability is only achieved when every quorum has size

$\lfloor \frac{n}{2} \rfloor + 1$. Setting $k = \lfloor \frac{n}{2} \rfloor + 1$ in Eqn. 2 gives,

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

which is asymptotically larger than M_{prob} for any value of p.

Now we consider quorum systems that have optimal load. For the probabilistic case, again we set $k = \Theta(\sqrt{n})$, which also gives optimal load. Then M_{prob} is the same as Eqn. 3. There exist strict quorum systems in which a priori sets of servers form the quorums (e.g., finite projective planes [28], a grid construction [13], etc.). Some of these systems have $k = \Theta(\sqrt{n})$, and $M_{str} = \Theta(mp\sqrt{n})$, which yields the same message complexity as the probabilistic case. However, it trades off with much lower availability.

G. Simulation Results for Expected Convergence Time

We have simulated systems of non-monotone and monotone random registers implemented using the algorithms from Sections D and 2 with a specific ACO. The simulation results shed some light on the following issues:

1. how much of an over-estimate is the upper bound derived in Corollary II.7 on the expected number of rounds per pseudocycle for the monotone case,
2. what is the convergence behavior in the original, non-monotone, case, and
3. what is the difference between the synchronous and asynchronous cases.

We took as our example application an all-pairs-shortest-path (APSP) algorithm presented in [38] 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

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

There are $p = n$ processes, and process i is responsible for updating the i-th row vector of $\mathbf{x}$, $1 \le i \le n$. 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, 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. 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

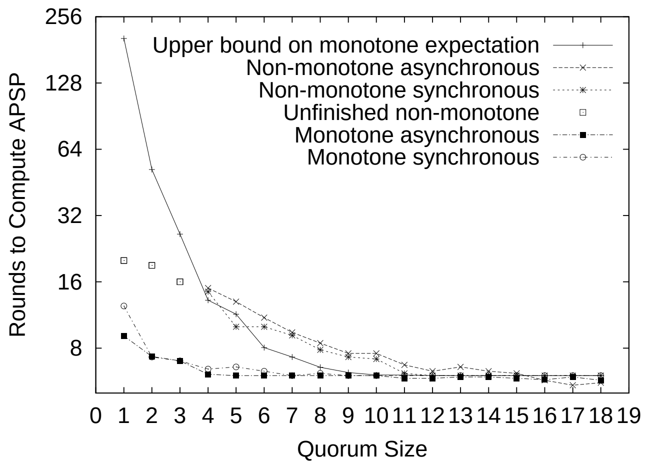


Fig. 4. Calculated Upper Bound and Simulation Results: Quorum Size vs. Rounds to Converge

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. [8, 39] for discussions of the issues involved in detecting termination for iterative algorithms.)

The upper bounds on the expected number of rounds until convergence in the monotone case for the various quorum sizes were calculated using the formula from Corollary II.7 and plotted in Figure 4. We simulated the four combinations of monotone/non-monotone and synchronous/asynchronous. For each of the four combinations, 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 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 quorum size). The phenomenon that asynchronous executions sometimes terminated 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 discrepancy between the calculated upper bound and the experimental value for the monotone case is quite large for very small quorums (e.g., 204 vs. 12.43 for synchronous and 9.08 for asynchronous executions when $k = 1$), but it decreases as the quorum size increases. One source of the overestimate is in the proof of Theorem II.5, where we did not take into account the fact that a read could obtain a value more recent than a given write without having to overlap any of that write's replicas.

The data indicates that the performance of the original algorithm is 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 4 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. Furthermore, for most of the other quorum sizes, the round numbers in the non-monotone case are larger than the computed upper bound for the monotone case.

With monotone executions, notice how a small quorum (say 4) is as good as a large one (large enough to be strict). This is in line with the intuition behind the original probabilistic quorum paper [30].

CHAPTER III

RANDOMIZED SHARED QUEUES APPLIED TO DISTRIBUTED OPTIMIZATION ALGORITHMS

A. Introduction

Quorum systems have been receiving significant attention because they provide consistency and availability of replicated data and reduce the communication bottleneck of some distributed algorithms (cf. [30] for references). The probabilistic quorum model [30] relaxes the intersection property of strict quorum systems, such that pairs of quorums only need to intersect with high probability. In Chapter II, *random registers* are defined as memory cells in which certain types of random errors can occur. It is shown in Chapter II that random registers can be used as an abstraction of probabilistic quorum systems. In particular, the typical access operations (read, write) are shown to have lower message complexity for random registers implemented with the probabilistic quorum algorithm of Malkhi et al. [30, 29] when compared to conventional shared memory implemented over strict quorum systems. At the same time, random registers inherit the known properties of the probabilistic quorum system, such as providing high availability and optimal load simultaneously [30]. Random registers were shown to be strong enough to implement an interesting class of iterative algorithms that converge with high probability.

In this chapter, we extend the results of Chapter II, which considers only read-write registers, to one of the fundamental abstract data structures: the queue. We propose a specification of a randomized shared queue data structure (*random queue*) that can exhibit certain errors — namely the loss of enqueued values — with some small probability. The random queue preserves the order in which individual processes enqueue, but makes no attempt to provide ordering across enqueuers. We show that this kind of random queue can be implemented with the probabilistic quorum algorithm of [29, 30].

Queues are a fundamental concept in many areas of computer science. A common application in distributed computing are message queues in communication networks. Many distributed algorithms use high-level communication operations, such as scattering or all-to-all broadcasts (cf. Chapter 1 of [8] for an overview). These algorithms can typically tolerate inaccuracies in the order in which the queue returns its elements, as the order of the elements in the message queue is typically impacted by the unpredictability of the communications network. Furthermore, we consider randomized algorithms, in which the queue elements contain data that can be incorrect or otherwise inappropriate with some probability. Algorithms of this type can typically tolerate the random disappearance of elements in the queue (with some small probability). We believe that this constitutes a large class of algorithms, which can take advantage of random queues and their benefits of optimal load and high availability. As an example of applications from this class, we analyze the behavior of a class of optimization algorithms [4], when used with random queues.

In [42, 41], Yelick et al. propose several irregular data structures and a *relaxed* consistency model for those data structures. For example, a *task queue* is an unordered collection of objects in which the priorities are locally, but not globally, observed. Such task queues can be used in the load balancing of the tasks of irregular applications. For the task queues of [42, 41], the randomization affects only the priorities. The number of enqueued tasks is always preserved. In [11], Chakrabarti et al. propose using distributed priority queues for the load balancing of parallel processors with dynamic scheduling algorithms. Again, the distributed queues affect the performance gain in a realistic execution environment compared to that with a centralized queue.

However, they do not specify any random behavior of queue operations.

B. Definitions

The data type of a shared object is defined by a set of operations and set of allowable sequences of those operations. In all other respects, the system model is the same as that in Section C of Chapter II.

C. A Random Queue

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

1. Specification of Random Queue

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

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

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

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

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

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

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

2. Implementation of Random Queue

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

The random queue algorithm (Algorithm 3) is based on the probabilistic quorum algorithm of Malkhi et al. [30]. There are r replicated memory servers.

The construction of Algorithm 3 proceeds in two steps. We begin by describing a random queue for the special case of a single enqueuer. The case of $n \geq 1$ enqueuers is implemented over a collection of n single enqueuer queues.

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

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

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

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

Algorithm 3a: Algorithm for client process – for single enqueuer and single dequeuer

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

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

Fig. 5. Algorithm 3: Implementation of p-Random Queue Q

Algorithm 3b: Algorithm for server process i, $1 \leq i \leq r$:

```
Initially local variable $Qcopy$, a queue, is empty
when $\langle \text{enq}, v, T \rangle$ is received from client $j$:
    enqueue $(v, T)$ to $Qcopy$
    send $\langle \text{ack} \rangle$ to client $j$

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

Algorithm 3c: Algorithm for a dequeuer extension for $n > 1$ enqueuers

```
Initially local variable $i = 0$, shared queue $Q = (Q_1, \ldots, Q_n)$
                                        // an array of $n$ single enqueuer queues
when Deq($Q$) occurs
    $i := (i \mod n) + 1$
    SingleDeq($Q_i, v$) // $v$ is value returned by SingleDeq
    Ret $(Q, v)$ // response to application
```

Fig. 5. Continued

dequeue from it.

Deq() checks the next queue in sequence. The round-robin sequence used in Algorithm 3 can be replaced by any other queue selection criterion that queries all queues with approximately the same frequency. The selection criterion will impact the order in which elements from the different queues are returned. However, it does not impact the probability of any given element being dequeued (eventually), as the queues do not affect each other, and the attempt to dequeue from an empty queue does not change its state.

3. Analysis of Random Queue Implementation

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

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

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

$$\mathbf{Pr}(\bigwedge_{i=1}^{m_z} Y_i = a_i) = \prod_{i=1}^{m_z} \mathbf{Pr}(Y_i = a_i) \tag{3.1}$$

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

$$\mathbf{Pr}(\bigwedge_{i=1}^{h} Y_i = a_i) = \mathbf{Pr}(Y_h = a_h | \bigwedge_{i=1}^{h-1} Y_i = a_i) \cdot \mathbf{Pr}(\bigwedge_{i=1}^{h-1} Y_i = a_i) \tag{3.2}$$

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

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

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

$$\mathbf{Pr}(Y_h = 1 | Y_j = 1 \wedge \bigwedge_{i=j+1}^{h-1} Y_i = 0) \quad = \quad \mathbf{Pr}(R \cap S_h \neq \emptyset | \bigwedge_{i=j+1}^{h-1} R \cap S_i = \emptyset)$$

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

$$\begin{aligned} &= \mathbf{Pr}(R \cap S_h \neq \emptyset) && (3.3) \\ &= \left(1 - \frac{\binom{r-k}{k}}{\binom{r}{k}}\right) = p \end{aligned}$$

(3.3) is because quorums are chosen independently.

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

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

By the formula of total probabilities, $\mathbf{Pr}(Y_h = 1) = p$. Thus, returning to (3.2):

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

Mutual independence (3.1) follows from this by induction. □

Theorem III.2 *Algorithm 3 implements a random queue.*

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

D. Application of Random Queue: Go with the Winners

In this section we show how to incorporate random queues to implement a class of randomized optimization algorithms called Go with the Winners (GWTW), proposed by Aldous and Vazirani [4]. We analyze how the weaker consistency provided by random queues affects the success probability of GWTW. Our goal is to show that the success probability is not significantly reduced.

1. The Framework of GWTW

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

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

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

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

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

We consider a distributed version of the GWTW framework (Algorithm 4), which is a modification from the parallel algorithm of [33]. Consider an execution of Algorithm 4 on n processes. At the beginning of the algorithm (stage 0), B particles are evenly distributed among the n processes. Since, at the end of each stage, some particles may be removed and some particles may be added, the processes need to communicate with each other to perform load balancing of the particles (global exchange). We use shared-memory communication among the processes. In particular, we use shared queues to distribute the particles among processes. Between enqueues and dequeues in Algorithm 4, we need some mechanism to recognize the total number of enqueued particles in a queue. It can be implemented by sending one-to-one messages among the processes or by having the maximum possible number of dequeues per stage. (Finding more efficient, yet probabilistically safe, ways to end a stage is work in progress.)

When using random queues, the errors will affect GWTW, since some particles disappear with some probability. However, we show that this does not affect the

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

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

Local variable: integer s, initially 0.

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

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

Fig. 6. Algorithm 4: Distributed Version of GWTW Framework

performance of the algorithms significantly. In particular, we estimate how the disappearance of particles caused by the random queue affects the success probability of GWTW.

2. Analysis of GWTW with Random Queues

We now show that Algorithm 4 when implemented with random queues will work as well as the original algorithms in [4].

We use the notation of [4] for the original GWTW algorithm (in which no particles are lost by random queues): Let X_v be a random variable denoting the number of particles at a given vertex v. Let S_i be the number of particles at the start of stage i. At stage 0, we start with B particles. Then $S_0 = B$ and $S_i = \sum_{v \in V_\ell} X_v$, for $i > 0$, where V_ℓ is the set of all vertices at depth ℓ. Let $p(v)$ be the chance the particle visits vertex v. Then $a(j) = \sum_{v \in V_j} p(v)$ is the chance the particle reaches depth j at least. $p(w|v)$ is defined to be the chance the particle visits vertex w conditioning on it visits vertex v. The values $s_i, 1 \leq i < \ell$ are constants which govern the particle reproduction rate of GWTWs. The parameter κ is defined to express the "imbalance" of the tree as follows: For $i < j$, $\kappa_{ij} = \frac{a(i)}{a^2(j)} \sum_{v \in V_i} p(v) a^2(j|v)$, and $\kappa = \max_{0 \leq i < j \leq d'} \kappa_{ij}$.

Aldous and Vazirani [4] prove

Lemma III.3

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

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

More precisely, given any original GWTW tree T, we define a modified tree T', which accounts for the effect of the random queues. Given a GWTW tree T, let T'

be defined as follows: For every vertex in T, there is a vertex in T'. For every edge in T, there is a corresponding edge in T'. In addition to the basic tree structure of T, each non-leaf node v of T has an additional child w in T'. This child w is a leaf node. The purpose of the additional leaf nodes is to account for the probability with which particles can disappear in the random queues in Algorithm 4.

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

Given a vertex v of T, let $\bar{p}(v)$ denote the probability that Algorithm 4, when run with a single particle and without reproduction, reaches vertex v. The term "without reproduction" means that the distribution mentioned in the first "for" loop of the algorithm is such that the number of added particles is always zero. The main property of the construction of T' is:

Fact III.4 *For any vertex v of the original tree T, $p'(v) = \bar{p}(v)$. Furthermore,*

$$\mathbf{Pr}(\textit{Algorithm 4 reaches depth } \ell) = p \cdot \mathbf{Pr}(\textit{GWTW on } T' \textit{ reaches depth } \ell)$$

for any $\ell \geq 0$.

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

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

For the second statement, it is sufficient to note that

$$\mathbf{Pr}(\text{Algorithm 4 reaches depth } \ell) = \sum_{v\in V_\ell} \bar{p}(v) = \sum_{v\in V_\ell} p'(v) = p \cdot \sum_{v\in V'_\ell} p'(v).$$

□

We can now analyze the success probability of Algorithm 4 (a combination of GWTW and random queues) by means of analyzing the success probability of baseline GWTW on a slightly modified tree. This allows us to use the results of [4] in our analysis. In particular,

Lemma III.5

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

Proof. We apply Lemma III.3 to the GWTW process on T' and show that $\kappa' = \kappa/p$ and $a'(i) = p^{i-1}a(i)$ for all i. Note that for any $i \le \ell$ and $v \in V_i$, $p'(v) = p(v)p^i$. Thus, for any $1 < i \le \ell$

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

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

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

□

In order to allow a direct comparison between the bounds of Lemmas III.3

and III.5, it is necessary to relate the constants $(s_i)_{1\leq i<\ell}$ and $(s'_i)_{1\leq i<\ell}$. These constants govern the particle reproduction rate of GWTW and can either be set externally or determined by a sampling procedure described in [4]. If we set $s'_i = p^{i-1}s_i$ then the expectations of Lemmas III.3 and III.5 are equal and the variance bounds are within a factor of p of each other. The variance bound is used in [4] in connection with Chebyshev's inequality to provide a lower bound on the success probability of GWTW. It follows that the negative effect of random queues on the GWTW variance bounds can be compensated for by increasing the number B of particles at the root by a factor of $1/p$.

CHAPTER IV

SHARED INFORMATION MANAGEMENT WITH QUORUMS IN MOBILE AD HOC NETWORKS

A. Introduction

We conjecture that randomization is a good approach to develop distributed algorithms in unpredictable and resource-poor communication environments, such as mobile ad hoc networks.

A **mobile ad hoc network (MANET)** consists of mobile computing entities that communicate with each other through wireless links, and has no fixed static infrastructure.

As discussed in [17], mobile ad hoc networks differ from mobile cellular telephone networks as follows. Mobile cellular telephone networks consist of two types of communication components: Firstly, there are base stations which serve to maintain location table registers and store location databases. Base stations are not mobile and communicate among each other to forward the information regarding call requests. The second type of components are cell phones, which are mobile and communicate with the base stations to make and receive calls.

Typically, MANETs also consist of two types of functional components: Firstly, there are special participants, which perform administrative functions similar to those performed by the base stations in cellular telephone networks (e.g. maintaining the location database). The difference of this special participant from the base station is that the participant itself is a mobile entity, i.e. it does not have a fixed location. The other class of functional components consists of mobile entities, which correspond to the cell phones in the cellular telephone network, and which communicate with the special participant to get the needed information. These classes are 'functional' in the sense that a single physical device may participate in the network in both roles.

Thus the difference between the mobile ad hoc network and the mobile cellular telephone network is that in the latter, the location databases are stored in fixed (i.e. non-mobile) locations, whereas in the former, no fixed infrastructure exists.

In mobile ad hoc communication environments, managing the mobility so as to keep track of the current location of mobile hosts is an important problem. In [17], an ad-hoc mobility management scheme is proposed, which routes most packets through arbitrary participants. This reduces the danger that the special participants may become a bottleneck. The role of the special participants is limited to storing location tables and computing routes through the general network. As described in [20], the *information dissemination* problem in ad hoc wireless networks is to track the location of each mobile node, and to gather information on the state of each mobile node.

We abstract those problems described above as an information sharing problem over distributed shared variables. The mobile hosts communicate with each other using shared variables. Since mobile ad hoc networks have no fixed infrastructure, every mobile host must be capable of serving as a distributed shared information server. The shared variable is a single-writer and multiple-reader register, which is replicated over every mobile host. Quorum based replica systems have been proposed in [17, 20] for this problem.

The proposed scheme in [17] is a quorum system based scheme for dynamic distributed construction of the location information database. In [20], Karumanchi et al. also propose a quorum based solution for the information dissemination problem in partitionable mobile ad hoc networks. To alleviate the problem of query failures, a set of heuristics is used in selecting servers for updates and queries, by maintaining a list of servers that are believed to be unreachable.

The quorum systems employed in the two papers [17, 20] are **strict** quorum systems, meaning that every pair of quorums intersect. A strict quorum system can be constructed with the smallest quorum size of $O(\sqrt{n})$. Even though it may provide a good complexity measure (relatively small cost to keep the needed information on a quorum of size $O(\sqrt{n})$), such quorum systems may suffer from unbalanced load and low availability in the face of node failures or unreachable nodes [30].

We propose to apply the probabilistic quorum system of Malkhi et al. [29, 30]. The probabilistic quorum system relaxes the quorum intersection property such that every pair of quorums intersect with high probability [30]. This implies that with some small probability, the probabilistic quorum system may return outdated information. This appears to be a tolerable problem with respect to location information. Movement is continuous and typically slow in relation to the relevant distances. It is already shown in [20] that the strict quorum systems can also return outdated information in mobile ad hoc communication environments. Furthermore, probabilistic quorum systems provide optimal load and high availability simultaneously, breaking the tradeoff between load and availability of strict (or traditional) quorum systems [30].

It is shown in Chapter II that random registers can be used as an abstraction of probabilistic quorum systems. In particular, the typical shared memory access operations (read, write) are shown to have lower message complexity for random registers implemented with the probabilistic quorum algorithm of Malkhi et al. [30, 29] when compared to conventional shared memory implemented over strict quorum systems. At the same time, random registers inherit the known properties of the probabilistic quorum system, such as providing high availability and optimal load simultaneously [30].

In this chapter, we apply the random register model of Chapter II to implement the location database to manage the mobility of mobile hosts in mobile ad hoc networks. Furthermore, we compare the probabilistic quorum based implementation with the strict quorum based implementation of [20] by way of simulations.

We perform simulations to answer the following questions:

- How does the probabilistic quorum model perform in mobile ad hoc communication environments?
- Do the probabilistic quorums perform better than the strict quorums in practical and dynamic communication environments such as mobile ad hoc networks?
- Can we characterize the behavior of those different quorum systems in regards of different scenarios for the communication environment?

We describe in detail the different algorithms to implement the quorum based replica systems in Section B.

To compare the performance of different quorum implementation algorithms, we define several performance measures in Section C.

In Section D, we explain the simulation setup, including the protocols used in each layer of the network, the sets of different parameters simulated, and so on. Then we display the simulation results and discuss our observations. We also propose possible extensions of the probabilistic quorum algorithm to take topology information into account when dynamically constructing quorums, in order to increase efficiency without harming the quorum intersection property too much. Furthermore, we justify how to take advantage of mobility of the mobile hosts, hoping that the information carried on the mobile hosts is gradually propagated as the mobile hosts move around.

B. The Algorithms

We define the quorum-based shared information database system in mobile ad hoc networks as follows.

The shared information is stored in single-writer, multiple-reader variables. The operations performed on those variables are **update** and **query**, which correspond to the conventional shared memory operations, write and read, respectively.

Each mobile host acts as both functional entities – a client and a server.

As a server, the mobile host h keeps a replica X^h of the shared information database X. When h receives an `update(`j`,`v`,`t`)` message from the mobile host j, h updates its replica ${X^h}_j$ with the new value v and the new timestamp t. Then h sends j an **Ack** message to acknowledge the update. When h receives a `query(`j`)` message from g, h sends g a response message with the value and timestamp of ${X^h}_j$.

As a client, to perform an update, the mobile host h increases its timestamp t by one, chooses a quorum Q, and sends out `update(`h`,`v`,`t`)` messages to every mobile host q in Q. When h has received all the **Ack**s from every q in Q, the update operation is said to be *complete*. To perform a query for j's data, h chooses a quorum Q, and sends out `query(`j`)` messages to every q in Q. When h receives all the `response(`v`,`t`)` messages, the query operation is said to be complete. It then chooses the v associated with the largest timestamp t, out of all the responses and its own local copy. Here we ensure the monotonicity of the shared memory systems that is exploited in Chapter II.

When h tries to choose a quorum, we count this as an *attempt*. When h has successively gotten (or constructed) a quorum, we count this as a *success*. We study different strategies in choosing quorums in the following subsection.

1. Strategies in Selecting Quorums

Firstly, we discuss three different strategies in selecting quorums in strict quorum systems.

In [20], a finite projective plane (FPP) based quorum construction is used: Let n be the number of mobile hosts serving as the replica servers. It is assumed that n is a perfect square. Then we can place the n servers in a $\sqrt{n}$ x $\sqrt{n}$ square grid. In the FPP quorum construction, a quorum can consist of either a row or a column, i.e., there can be a priori rule to choose a read quorum from the set of rows and a write quorum from the set of columns, or vice versa. We name this FPP quorum construction as **SQ1**. Thus, with SQ1, the quorum size is $\sqrt{n}$. SQ1 guarantees that there is always one member in the intersection of a pair of read and write quorums. However, to increase the availability of the quorum system in the mobile ad hoc network, Karumanchi et al. [20] union one row and one column to construct a quorum. We will call such quorum construction as **SQ2**. With SQ2, there are n a priori quorums formed and there are at least two members in the intersection of any pair of quorums. And the quorum size is $2 \cdot \sqrt{n} - 1$

Furthermore, to get better performance, Karumanchi et al. keep a list of unreachable hosts (*unreachable nodes list: UNL*) in case the network partitions or some crash failures of servers occur. Three kinds of heuristics are used when choosing quorums: (1) Eliminate the quorums that have any of those nodes in the UNL, and then uniformly randomly select one from the remaining quorums (Eliminate-Then-Select: ETS). (2) First select a quorum uniformly at random, and then eliminate those nodes in the UNL from the chosen quorum (Select-Then-Eliminate: STE). (3) Use ETS for updates and STE for queries (Hybrid).

We name the Hybrid strategy as **SQ3**. In this chapter, we perform simulations on SQ1, SQ2, and SQ3, as strict quorum implementations.

For probabilistic quorum systems, we first eliminate the hosts in the unreachable list, and then uniformly randomly choose k servers to dynamically form a quorum of size k, where k is an input parameter to the quorum system. We address in Section C, the issue when k servers are not available.

C. Performance Measures

We employ four kinds of measures in order to compare the performance of quorum based shared memory systems in mobile ad hoc networks:

- The *shared memory recency rate*: indicates the correctness or consistency of the distributed shared memory system. We count the number of outdated values returned by query operations. We define outdatedness as follows: a query to variable x is *outdated* if it returns a value with a timestamp that is older than the timestamp of the most recent *complete* (cf. Section B) update of x. In our simulation, the simulator compares the timestamps simultaneously at the time when the reply is given back to the application. Let $\mathcal{N}_c$ denote the number of completed query operations and $\mathcal{N}_o$ the number of outdated values returned by those queries. Then the recency rate is measured as $\frac{\mathcal{N}_c - \mathcal{N}_o}{\mathcal{N}_c}$.

- The *quorum availability rate*: indicates the fault-tolerance of the quorum system. It reflects network link stability, network partition possibility, and crash failures of the nodes. Let $\mathcal{N}_a$ be the number of attempts to choose quorums and $\mathcal{N}_s$ the number of successes in choosing quorums. Then the availability rate is measured as $\frac{\mathcal{N}_s}{\mathcal{N}_a}$.

  This way of measuring the availability rate is inappropriate for SQ1 and SQ2 because they do not utilize the UNL when choosing quorums and instead blindly choose a random quorum out of the a priori set of quorums. Thus, the availability rate for SQ1 and SQ2 would be always one ($\mathcal{N}_a = \mathcal{N}_s$). Therefore, for SQ1 and SQ2, we count the number of timeouts as the number of quorum failures. The timeout feature we adopt is explained in detail in Section D. Let $\mathcal{N}_f$ be the number of failed shared memory operations due to timeout. Then the availability rate for SQ1 and SQ2 is computed as $1 - \frac{\mathcal{N}_f}{\mathcal{N}_a}$.

- The *average completion time per operation*: indicates responsiveness of the shared memory system. Let $\mathcal{T}$ denote the total time for shared memory operations and $\mathcal{N}_m$ the number of completed shared memory operations. Then the average time per operation is measured as $\frac{\mathcal{T}}{\mathcal{N}_m}$.
- The *shared memory system throughput*: measures how efficiently the system performs. It also reflects the network load. The throughput is measured as the number of completed shared memory operations per unit time (second).

D. Simulation

In our simulation, we use the ns-2 Network Simulator [40] with CMU/Monarch group's mobility extension [14]. We use TCP (Reno) for the transport layer and DSR (Dynamic Source Routing) for the routing (network) layer, IEEE 802.11 MAC protocol for the link layer, and Two Ray Ground Radio Propagation model for the physical layer. The transmitter range is set as 250 meters.

We modified the telnet application so that it incorporates the shared memory subsystem layer between the actual application that invokes update/query shared memory operations and the communication network layer. Each application process invokes update and query operations periodically. An application can have at most one pending query operation and one pending update operation simultaneously. The sequence of update operations is independent of the sequence of query operations at each process. The first update for process P_i is invoked $2.05/(i+1)$ seconds after the process starts, while the first query for the process is invoked $2.1/(i+1)$ seconds after the process starts. Subsequently, the invocation of the next update (resp., query) is scheduled for 2 seconds after the invocation of the current update (resp., query). If the current operation (query or update) has not yet finished, then the invocation is rescheduled for one second later, until it succeeds.

The essential role of the shared memory subsystem (SMS) is to interpret the memory operation invoked by the application process, to generate a set of messages to a quorum, and to handle the response messages gotten from the quorum to generate the result of the memory operation for the application process. We implemented the shared memory subsystem with the four algorithms (PQ, SQ1, SQ2, and SQ3) described in Section B, ran each implementation with sets of parameters, and compared their performance based on the four measures explained in Section C.

The operations time out if some member(s) of the quorum does not respond in 10 seconds. Then a new quorum is chosen and the time-out operation is repeated. Each application keeps the unreachable nodes list (UNL) by trying to find if there exists a path to every other node in the system. This is done using the route request function of DSR. The UNL is updated every second.

We assume there are $n = 25$ mobile nodes in the system, placed at random in a 2-D rectangular area. The maximum speed of each mobile node is 4 meter/sec. and each of the 25 nodes acts as both client and server of the shared memory subsystem. The area where the nodes may move around is varied from 300x300 m^2 to 1000x1000 m^2 with an increment of 100 meters in each direction. We ran each simulation for 1200 seconds. For the PQ implementation, we vary the quorum size from 1 to 15.

Each plot in the figures is the computed average of seven runs on seven different movement scenarios. We used the Random Waypoint mobility model from [9]. Figures 7 to 10 show how the different quorum sizes perform differently. Figures 11 and 12 display the different performance of the three strict quorum implementations. In figures 13 and 14, we compare the SQs and the PQ of the corresponding quorum sizes (which are 5 and 9 in this simulation).

1. Probabilistic Quorum Implementations

For the PQ implementation, we simulated two different schemes: with and without timeout of the shared memory operation. In Figures 7 to 10, the left-hand side figures are without timeout feature, i.e., once an update or query operation is invoked and the corresponding messages are sent out, the shared memory subsystem indefinitely waits until all the responses are gotten back. It was an optimistic design of the memory system because the SMS hoped that there would not have been much movement of nodes since last computation of the unreachable nodes list (UNL). However, the simulation results revealed that this optimistic design is not appropriate for the mobile ad hoc network system we simulate. Along with the assumption that there is not more than one update and query operation pending per node, this optimistic implementation yielded very poor performance in terms of throughput. This phenomenon is displayed in the left-hand side of Figure 8.

Thus, we adopted a timeout feature, so that when a certain amount of time (10 seconds in this simulation) has elapsed since the current pending memory operation was invoked, the SMS assumes some member(s) of the quorum is currently unreachable, crashed, or the link has failed, and aborts the operation. Then the SMS retries to perform the operation by sending out messages to the newly chosen quorum. The timeout feature increased the throughput greatly, as shown in Figure 8, with a little compensation of recency rate (cf. Figure 7). The recency rate, however, is almost the same for both schemes as long as the network is in a reasonable size of area so that the network is not partitioned a lot; with timeout, the recency rate starts degrading as the area and the quorum size become large (cf. 800x800 or larger area with quorum size 12 or more). For small quorum sizes (say, 1 to 4), the timeout scheme yields better recency rate in most cases.

As can be seen in Figures 9 and 10, using the timeout feature smoothed out the average time per operation and quorum availability rate, so that we can better predict the performance of the system based on the system parameters such as the quorum size and the area. The timeout feature helped a lot to get better average time per operation.

The quorum availability rate clearly displays how network partition affects success in choosing random quorums. As the area gets larger, the quorum availability rate significantly drops. Even in small areas, it is often not possible to choose large quorums.

Overall, it was noticed that having higher throughput by use of the timeout feature enabled us to obtain better statistics of the behavior of the system. One observation here that was not explained by the theoretical results shown in Chapter II is that having a large quorum size does not result in a better recency rate of the shared memory system for mobile ad hoc networks with network partitions.

2. Strict Quorum Implementations

All three strict quorum implementations use the timeout feature similar to that of the probabilistic quorum implementation. SQ1 and SQ2 do not use the unreachable nodes list (UNL). They first choose a quorum randomly out of the a priori constructed set of quorums and send out the corresponding messages. Then if timeout occurs, it is considered (for some reason) that the quorum choice failed and the operation

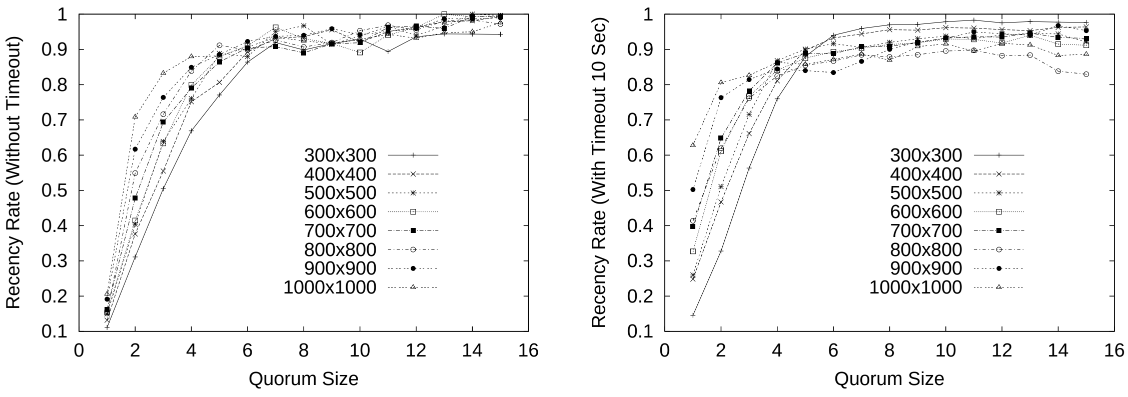


Fig. 7. Probabilistic Quorum: Quorum Size vs. Recency Rate

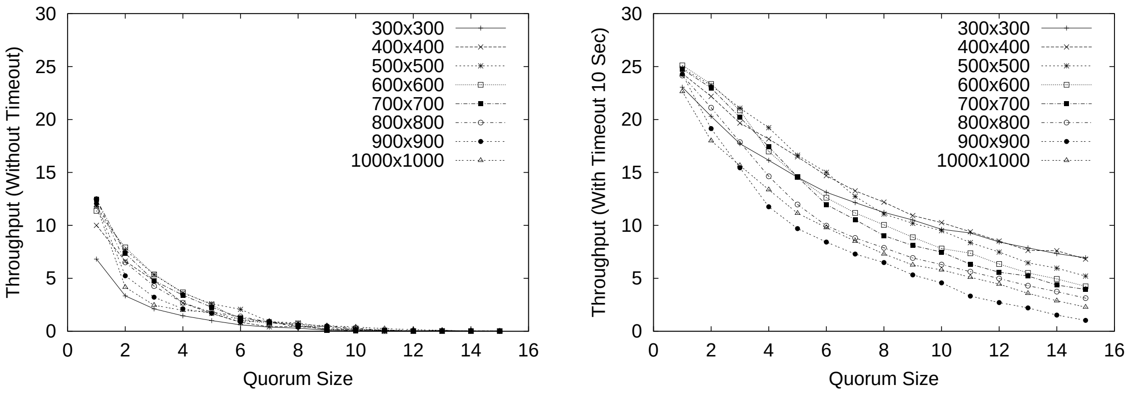


Fig. 8. Probabilistic Quorum: Quorum Size vs. Throughput

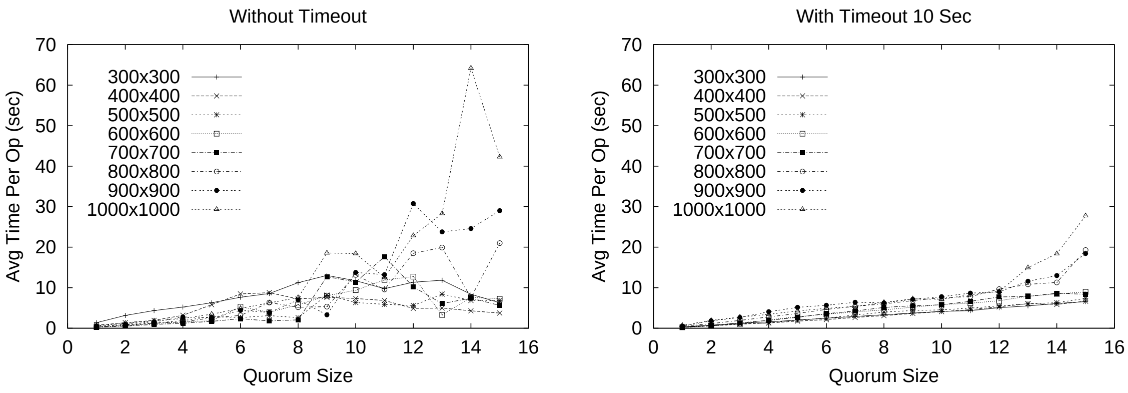
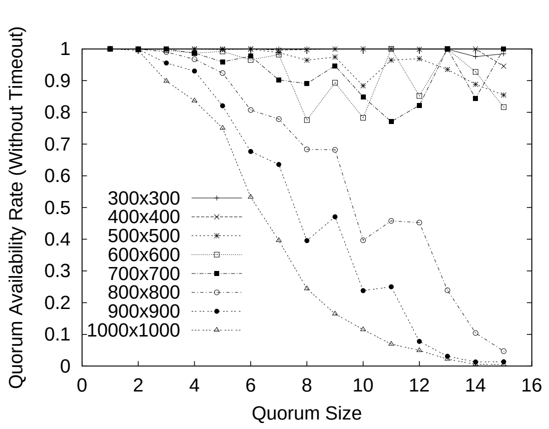


Fig. 9. Probabilistic Quorum: Quorum Size vs. Avg Time Per Op



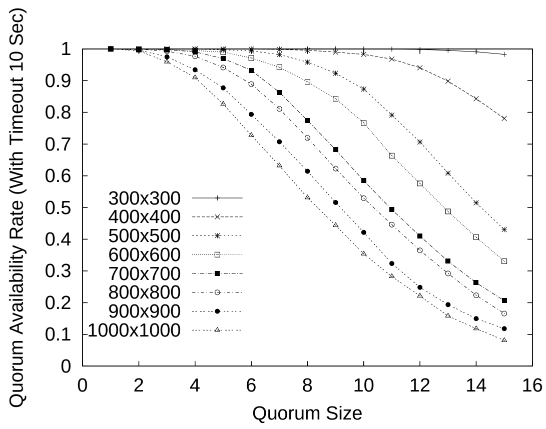
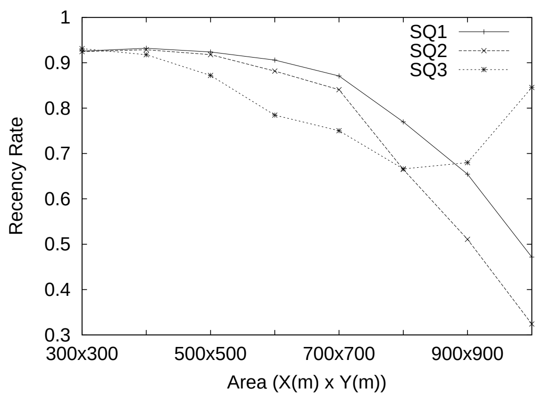


Fig. 10. Probabilistic Quorum: Quorum Size vs. Quorum Availability Rate



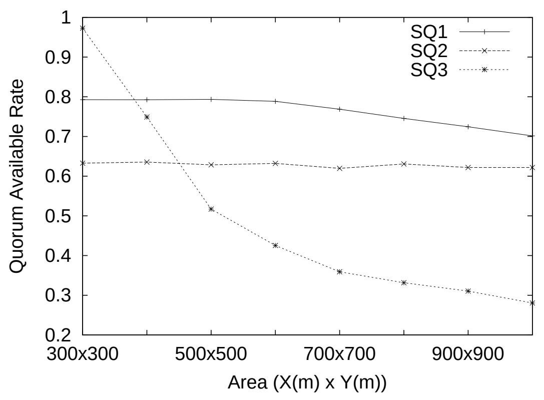


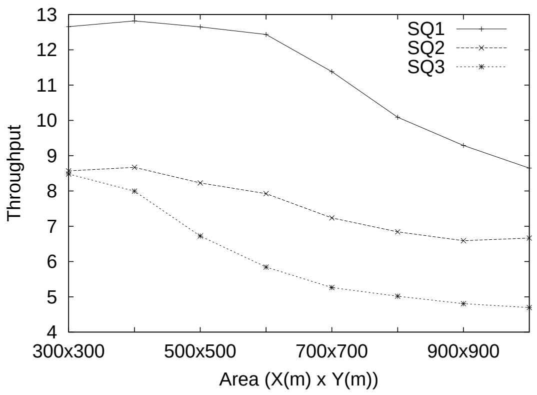
Fig. 11. Strict Quorums: Area vs. Recency Rate and Quorum Availability Rate

is restarted by choosing a new quorum. This occasion of timeout is accounted as the available rate. SQ3 uses UNL to implement the hybrid scheme with different heuristics: for a query operation, choose a quorum and then eliminate those members in UNL from the quorum, and for an update operation, first eliminate those quorums with a member in UNL and then choose a quorum out of the remaining quorums. And then it uses timeout to avoid any indefinite waiting situation resulting from the movement of nodes.

In the left-hand side figure of Figure 11, SQ1 shows quite good recency rate in relatively small area (say, up to 600x600). It shows better recency rate than SQ2 in all areas, and better than SQ3 in reasonably large areas (say, up to a little less than 900x900). Considering that SQ1 has quorum size 5 and 5 possible choices of quorums, and SQ2 has quorum size 9 and 20 possible choices of quorums, this was an unexpected observation. SQ2 has very low recency rate as soon as the network starts getting partitioned. As the area gets large such as 900x900, SQ3 performs better than the others, when SQ1 and SQ2 degrade quickly. Even though SQ3 has worst recency rate in the mid-sized areas, it displays some lower bound, which indicates that it can guarantee some bounded deterioration of recency rate.

The right-hand side figure of Figure 11 show that SQ1 and SQ2 have almost constant quorum availability rates. In this simulation, the heuristics adopted in SQ3 perform poorly. The quorum availability rate of SQ3 drops significantly as the area increases.

In the left-hand side figure of Figure 12, SQ1 shows the best performance of the three in terms of throughput. However, it is affected by the area, and noticeably drops as the area gets large (more than 600x600). This is shown again with the average time per operation in the right-hand side figure, where the time of SQ1 starts increasing with the area of 600x600. SQ2 performs poorly in terms of average time



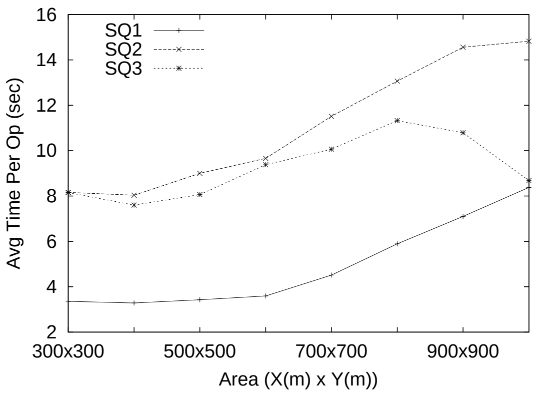
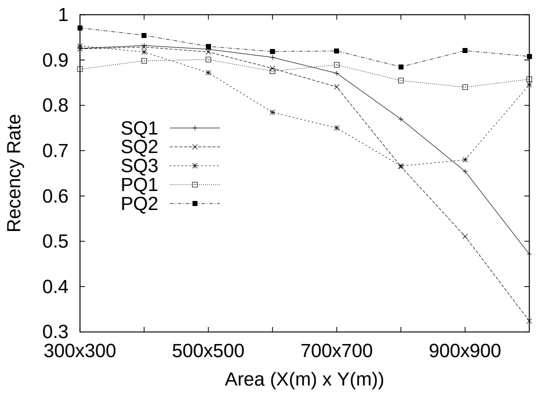


Fig. 12. Strict Quorums: Area vs. Throughput and Avg Time Per Op



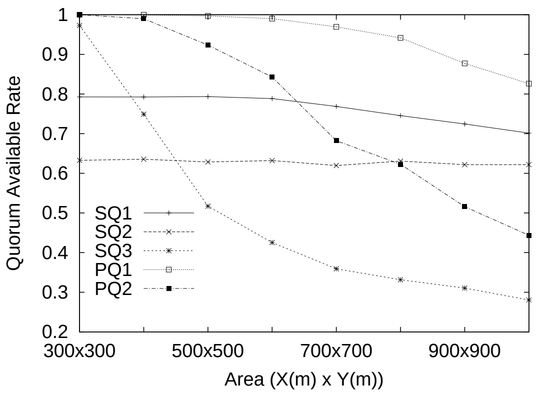


Fig. 13. Strict and Probabilistic Quorums: Area vs. Recency Rate and Availability Rate

per operation.

Overall, in a reasonable area (of size less than 600x600), SQ1 has quite good performance. SQ3 does not show in our simulation, the effectiveness of adopting the heuristics. Furthermore it seems to be affected a lot by system environment such as area and system parameters such as time interval to update the UNL, and so on.

Now we compare those strict quorum implementations with the probabilistic quorum in the following section.

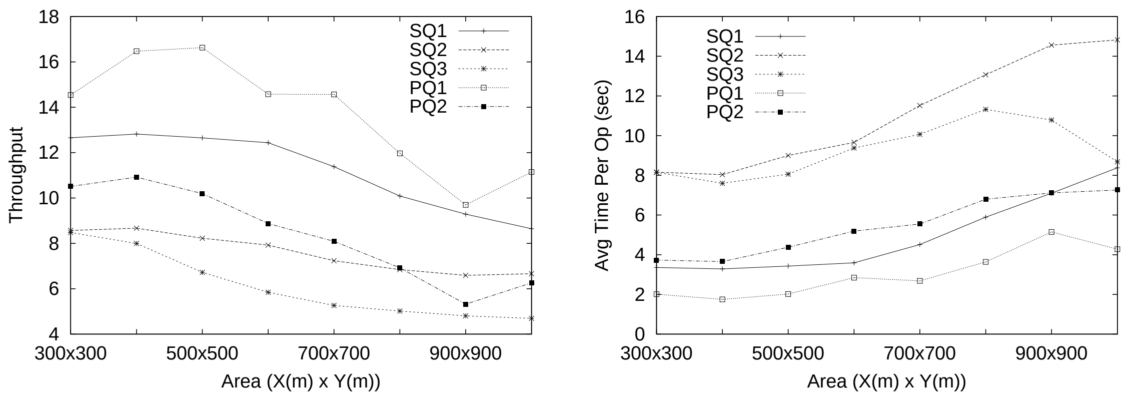


Fig. 14. Strict and Probabilistic Quorums: Area vs. Throughput and Avg Time Per Op

3. Comparison of Strict and Probabilistic Implementations

To compare the probabilistic quorum implementation with the three strict quorum implementations, we re-plotted the two quorum sizes 5 (same size as SQ1's) and 9 (same size as SQ2's and SQ3's) from the probabilistic quorum case together with the plots of SQs. We denote the plot of probabilistic quorum size 5 as PQ1 and that of size 9 as PQ2.

In Figure 13, SQ1 has better recency rate than that of PQ1 in areas smaller than 700x700, but as the area gets larger, SQ1's recency rate drops quickly, when PQ1's stays almost constant. Similar phenomenon happens with SQ2 and PQ2, and PQ2 is superior than any others, in terms of recency rate. PQ1 and PQ2 show better recency rate than SQ3 even in very large area of 1000x1000.

In terms of quorum availability rate, PQ1 is the best, as shown in the right-hand side in Figure 13. This concurs with the theoretical result discussed in Chapter II. With the areas up to 600x600, PQ2 is still better than SQs, but as the area becomes larger, PQ2's availability rate drops significantly, yielding worse performance than SQ1 and SQ2, but still better than SQ3.

Figure 14 shows again PQ1 is the best in regards of throughput. PQ2 is better than SQ2 in areas smaller than 800x800, but worse than SQ2 in larger areas, and better than SQ3 in all areas.

The average time per operation shown in the right-hand side of Figure 14 displays that PQ1 is again the best. SQ1 is better than PQ2 until the area is within 900x900, but seems to have increasing time per operation as the area becomes larger. Both PQ1 and PQ2 show better performance than SQ3. However, it is noted that even in the best case of PQ1, it takes about 2 seconds to complete one memory operation. Even though this time includes the time for retries by the shared memory system, when considering this time as the response time to the application, it will be unrealistically slow to use this memory system in implementing time-critical applications. Thus, this shared memory system is desirable for applications that do not involve intensive shared memory operations.

Overall, probabilistic quorum implementations show improved performance in all measures with some trade-offs between recency rate and quorum availability rate, for example.

We would like to design a new quorum implementation algorithm that guarantees as high recency rate as PQ2 and provides as high quorum availability rate and throughput as PQ1, while keeping the average response time small. One observation from the simulation is that many occasions of low performance resulted from the lack of knowledge about the topology changes of the network. This can be easily explained from the simulation results we have discussed so far. Thus, we propose a new scheme for quorum construction: a *topology-sensitive quorum system (TSQ)*, which takes topology information into account when dynamically constructing quorums. Based on the observations from the simulation results, we conjecture that TSQs may be able to take advantage of topology information in order to increase efficiency without harming the quorum intersection property too much. Furthermore, we expect that TSQs would take advantage of mobility of the mobile hosts to propagate information. It remains as future work to design a concrete algorithm to implement the TSQ and analyze its performance either theoretically or experimentally (or both).

CHAPTER V

SUMMARY AND FURTHER RESEARCH

A. Dissertation Summary

In this dissertation we performed three avenues of research related to randomized memory models and their applications in distributed computing. In Chapter II, we have suggested two specifications of randomized registers 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 [29, 30]. Furthermore, our specifications can be used to implement a significant class of iterative algorithms [38] of practical interest. We evaluated the performance of the algorithms experimentally as well as analytically, computing the convergence rate and the message complexity.

In Chapter III, we have proposed a specification of a randomized shared queue data structure (random queue) that can exhibit certain errors — namely the loss of enqueued values — with some small probability. The random queue preserves the order in which individual processes enqueue, but makes no attempt to provide ordering across enqueuers. We showed that this kind of random queue can be implemented with the probabilistic quorum algorithm of [29, 30]. We identified, as potential applications of random queues, distributed algorithms that have a particular pattern of interprocess communication using message passing and that can tolerate a small number of message loss and inaccuracies in the order in which messages arrive. In these algorithms, the message passing communication is to be replaced by enqueuing and dequeuing information on random queues. We believe that this constitutes a large class of algorithms, which can take advantage of random queues and their benefits of optimal load and high availability. As an example of applications from this class, we analyzed the behavior of a class of combinatorial optimization algorithms (Go With the Winners [4]).

In Chapter IV, we have applied the probabilistic quorum system of Malkhi et al. [29, 30] to implement the shared information database system in mobile ad hoc networks. An application of such shared information system is location information database for mobility management of the mobile hosts. First we compared two different implementations of probabilistic quorum: with or without timeout feature. The simulation results showed that employing timeout feature is more desirable to yield better performance. Then we compared, by way of simulation, the probabilistic quorum implementation with the timeout feature, to the three strict quorum implementations (SQ1, SQ2, and SQ3), which are discussed in [20]. The observations are: our probabilistic quorum implementation performs better than the strict quorums, and the topology change due to the continuous movement of nodes has a significant effect on the performance of quorum systems. Thus, understanding the node movement pattern and the system environment of the mobile ad hoc network in consideration is critical in providing an appropriate implementation algorithm for the shared database system in the mobile ad hoc network.

B. Further Research

In this section we describe a few directions along which the research in this dissertation can be extended.

1. Random Registers

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

Another direction is how to design more powerful read-write registers and other data types in our framework. Malkhi et al. [30] 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.

This dissertation 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.

2. Random Queues

Another possible class of applications for a random queue is randomized Byzantine agreement algorithms in which the set of faulty processes can change from round to round (e.g. Rabin's algorithm [35, 31]). Random errors in the queue can be attributed to faulty processes. Issues to be resolved include how to adapt the message passing algorithms to the situation when too few messages are received; also whether probabilistic quorum algorithms in [30] that tolerate Byzantine failures can be exploited here.

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

3. Applications on Mobile Ad Hoc Networks

It remains as a future work to design and analyze an algorithm for a quorum system that takes topology information into account so as to obtain better performance without harming the quorum intersection property too much.

It also remains as a future work to investigate whether read/write registers are the most suitable data structure for the location information database in mobile ad hoc networks, or if different (new) data structures may provide more efficient construction of such database.

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, pp. 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*, pp. 9–30, Oct. 1991.

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

[5] 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, pp. 65–89, Feb. 1998.

[6] H. Attiya and R. Friedman, "A Correctness Condition for High-Performance Multiprocessors," *Proc. 24th ACM Symposium on Theory of Computing*, pp. 679–690, 1992.

[7] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* New York: McGraw-Hill, 1998.

[8] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation.* Englewood Cliffs, N.J.: Prentice Hall, 1989.

[9] J. Broch, D. A. Maltz, D. B. Johnson, Y. Hu, and J. Jetcheva, "A Performance Comparison of Multi-hop Wireless Ad Hoc Network Routing Protocols," *Proc.*

ACM/IEEE Int. Conf. on Mobile Computing and Networking, pp. 85–97, Oct. 1998.

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

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

[12] D. Chazan and W. Miranker, "Chaotic Relaxation," *Linear Algebra and Its Applications*, vol. 2, pp. 199–222, 1969.

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

[14] CMU Monarch Project Team, "The CMU Monarch Project's Wireless and Mobility Extensions to *ns*," Aug. 1999. Available from `http://www.monarch.cs.cmu.edu/`.

[15] 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, pp. 103–120, 1997.

[16] R. Gupta, S. Smolka, and S. Bhaskar, "On Randomization in Sequential and Distributed Algorithms," *ACM Computing Surveys*, vol. 26, no. 1, pp. 7–86, Mar. 1994.

[17] Z. J. Haas and B. Liang, "*Ad Hoc* Mobility Management With Uniform Quorum Systems," *IEEE/ACM Trans. on Networking*, vol. 7, no. 2, pp. 228–240, Apr. 1999.

[18] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, July 1990.

[19] P. Jayanti, T. Chandra, and S. Toueg, "Fault-tolerant Wait-free Shared Objects," *J. ACM*, vol. 45, no. 3, pp. 451–500, May 1998.

[20] G. Karumanchi, S. Muralidharan, and R. Prakash, "Information Dissemination in Partitionable Mobile Ad Hoc Networks," *Proc. IEEE Symp. on Reliable Distributed Systems*, pp. 4–13, Oct. 1999.

[21] L. Lamport, "How to Make a Multiprocessor That Correctly Executes Multiprocess Programs," *IEEE Trans. Comput.*, vol. C–28, no. 9, pp. 690–691, Sep. 1979.

[22] L. Lamport, "On Interprocess Communication, Part I: Basic Formalism," *Distributed Computing*, vol. 1, no. 2, pp. 77–85, 1986.

[23] L. Lamport, "On Interprocess Communication, Part II: Algorithms," *Distributed Computing*, vol. 1, no. 2, pp. 86–101, 1986.

[24] H. Lee and J. L. Welch, "Brief Announcement: Specification, Implementation and Application of Randomized Regular Register," *Proc. 19th ACM Symposium on Principles of Distributed Computing (PODC)*, p. 338, July 2000.

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

pp. 21–28, Apr. 2001.

[26] H. Lee and J. L. Welch, "Brief Announcement: Randomized Shared Queues," To appear in *Proc. 20th ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 2001.

[27] H. Lee and J. L. Welch, "Randomized Shared Queues Applied to Distributed Optimization Algorithms," To appear in *Proc. 12th Int. Symposium on Algorithms and Computation (ISAAC)*, Dec. 2001.

[28] M. Maekawa, "A $\sqrt{n}$ Algorithm for Mutual Exclusion in Decentralized Systems," *ACM Transactions on Computer Systems*, vol. 3, no. 2, pp. 145–159, May 1985.

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

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

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

[32] M. Naor and A. Wool, "The Load, Capacity and Availability of Quorum Systems," *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pp. 214–225, 1994.

[33] M. Peinado and T. Lengauer, "Parallel 'Go with the Winners' Algorithms in the LogP Model," *Proc. 11th Int. Parallel Processing Symposium (IPPS)*, pp. 656–664, 1997.

[34] D. Peleg and A. Wool, "The Availability of Quorum Systems," *Information and Computation*, 123(2), pp. 210–233, 1995.

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

[36] N. Shavit and A. Zemach, "Diffracting Trees," *ACM Trans. on Computer Systems*, vol. 14, no. 4, pp. 385–428, Nov. 1996.

[37] N. Shavit and A. Zemach, "Combining Funnels," *Proc. 17th Annual ACM Symposium on Principles of Distributed Computing,* pp. 61–70, 1998.

[38] A. Üresin and M. Dubois, "Parallel Asynchronous Algorithms for Discrete Data," *J. ACM,* vol. 37, no. 3, pp. 558-606, July 1990.

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

[40] VINT Project Team, "The *ns* Manual," Feb. 2001. Available from `http://www.isi.edu/nsnam/ns/`.

[41] K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C. Wen, "Data Structures for Irregular Applications," *Proc. DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems,* June 1993. Available from `http://www.cs.berkeley.edu/projects/parallel/castle/multipol/`.

[42] K. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C. Wen, "Parallel Data Structures for Symbolic Computation," *Proc. Workshop on Parallel Symbolic Languages and Systems,* Oct. 1995. Available from `http://www.cs.berkeley.edu/projects/parallel/castle/multipol/`.

VITA

The author, Hyunyoung Lee, was born in Seoul, Republic of Korea on May 29, 1964. She received her B.S. degree in computer science from Ewha University in 1987 and then worked for Korean Air, Information Systems Department, in the area of operating system support and system administration from 1987 to 1990. After received her M.S. degree in computer science from Ewha University in 1992, she taught as a lecturer in the Department of Information Science and Department of Computer Engineering at Ajou University from 1992 to 1993. She received her M.A. degree in computer science from Boston University in 1998.

In September 1997, Hyunyoung Lee joined Texas A&M University and started working toward her Ph.D. degree in computer science under the supervision of Dr. Jennifer L. Welch. Her research interests include theory of distributed computing, distributed algorithms and systems, fault-tolerant computing and mobile computing, and parallel computing and parallel algorithms.

Hyunyoung Lee's current address is: Department of Computer Science, Texas A&M University, College Station, TX 77843-3112.

The typist for this thesis was Hyunyoung Lee.