An Efficient Implementation of the Quorum Consensus Protocol

M. L. Liu D. Agrawal A. El Abbadi

Department of Computer Science

University of California

Santa Barbara, CA 93106

1 Introduction

While distributed database systems have been the subject of extensive research, its acceptance in the industry remains tenuous [22]. A key reason for this phenomenon is that for such a system to function properly, it must perform correctly in the presence of communication and site failures. Although many protocols which address failures have been proposed, their implementation has not been widespread. For example, few in the the family of *replica control* protocols have ever been put into practice [24, 22] . This lack of acceptance is largely due to (i) the difficulty involved in assessing the performance impact of the protocols, and (ii) the high complexity typically associated with such protocols. In particular, there are not enough existing distributed database applications to provide a sound basis for empirical judgements that must be exercised by the designer of a distributed database system. While analytical models can be useful for performance studies in lieu of practical measurements, they may not be enough to provide sufficient guidelines to help a designer faced with the difficult decision of choosing from a set of available implementation options. Existing papers on protocols for distributed database systems generally explore performance properties in terms such as number of messages or number of log forces, or data availability. However, typically the properties are derived in the isolated context of the protocol being studied. On the other hand, a designer must be mindful of the dynamic performance of the system running a divergence of protocols. For example, on practical systems, the performance of the replica control protocol has dynamic ramifications on the performance of other protocols such as the transaction execution protocol. Hence, while analytical properties of individual protocols provide useful insights on isolated components of a distributed database systems, a more comprehensive performance model is needed in practice.

In the absence of a full implementation, simulation becomes a viable alternative for a comprehensive performance model. Indeed simulation has been employed in many existing studies on protocols for distributed database systems. [2, 8, 16, 23]. In our study, we use a simulation model to study the performance of a number of replica control protocols, including the popular *primary copy* protocol and the less widely accepted *quorum consensus* protocol. However, when we set about to implement the quorum consensus protocol, we were confronted with a myriad of options for the simple task of collecting a quorum, a task required by the protocol but the implementation detail of which is not generally specified. As mentioned above, analytical properties of a protocol itself is often insufficient to help a designer to exercise his options. Indeed, in our previous study [20] we have demonstrated how, in the case of *atomic commitment protocols*, the dynamic behavior of the system can be counter-intuitive. In this paper, we explore a number of implementation approaches for the quorum consensus protocol by using a simulation model to evaluate their relative performance.

The paper is organized as follows: In Section 2, we begin with an overview of replica control protocols. The four implementation approaches for implementing the quorum consensus protocol are described in Section 3. We describe the simulation model in Section 4 and present the results in Section 5. The paper concludes with a discussion of our findings.

2 Replica Control

Replicated data is employed in distributed database systems to enhance data availability: multiple copies of a critical data item is maintained, typically on separate sites, so that the data item can be retrievable even if some copies of the data item cannot be accessed due to system failures. However, this benefit of data availability is only realized at the cost of elaborate algorithms which hide the underlying complexity of maintaining multiple versions of a single data item. Ideally, an application should not be cognizant of the existence of replicated data, so that it is written as if for a non-replicated database. This criteria is relatively easy to achieve during normal system operation; but is much more difficult to deal with in the face of system failures, the existence of which is the main motive for replicating data in the first place. The difficulty lies in keeping the replicas consistent with each other in the face of system failures while at the same time maximizing the data availability. The algorithms which address these problems are called replica control algorithms [4].

A transaction T originates at a site, called the *coordinator*, which forwards the operations contained in T to the appropriate database sites, called the *participants*, where the data is stored. Each operation either *reads* from or *writes* to a data item. In a replicated database, the coordinator translates each Read and Write on a data item x in such a manner that an interleaved execution of transactions is equivalent to a serial execution of those transactions on an unreplicated database. This property, known as *one-copy serializability* [4], is the correctness criteria for transaction execution on a replicated database system. On a system with replicated data, each Read(x) is translated by the *transaction coordinator* into Read(x_{A_1}), ..., Read(x_{A_m}), where $x_{A_1}$is the replica of x at site A_1. Similarly, each Write(x) is translated into Write(x_{B_1}), ..., Write(x_{B_n}) [4]. The composition of the read set $R = \{A_1, ..., A_m\}$ and the write set $W = \{B_1, ..., B_n\}$ is protocol-dependent. Below we will discuss two main families of such protocols.

2.1 The Write-All Available Approach

In the absence of failures, one-copy serializability can be easily achieved by the read-one-write-all approach. In this approach, the read set R contains any replica of x, while the write set W contains all copies of x. To optimize read response time, R can be set to contain the replica of x that is nearest to the transaction coordinator. In the presence of failures, however, this approach results in write operations being blocked or aborted even if a single copy of the data item is unavailable. Hence the write-all approach is modified instead to write into all copies available to the transaction coordinator; such protocols are termed *write-all-available* protocols. The most commonly known protocol of this genre is the *primary copy* protocol [3, 28]. The protocol is designed for systems where a primary copy of each data object is located at one site, while replicas of the object are distributed among other nodes. A read operation is directed to the primary copy only, while a write operation results in updates of the primary copy as well as all available secondary copies. Unavailable secondary copies receive the update on a deferred basis.

The primary-copy protocol is perhaps the most widely accepted replica control protocol in practice: It has been employed in existing implementations such as Distributed INGRES [29, 28] and the HARP distributed file system [21]. The protocol's primary appeal is in its conceptual simplicity, and, in environments where the primary copy can be chosen to be the closest to the transaction coordinator, the protocol yields the best response time for read operations. However, in some implementations, the primary copy protocol does not guarantee data consistency in the presence of network partitioning. This is due to the possibility that different partitions of a network may adopt inconsistent views of the designation of primary copy for a given data object. Thus partition A may view replica x_a as the primary copy, while partition B regards replica x_b as the primary.

In other implementations, the protocol has been enhanced to tolerate network partitioning by maintaining a systemwide consistent view of the replica configuration. In INGRES [28], the configuration is maintained systemwide in an "up list". Based on the up list, a group of replicas for a data object can consistently determine which of the replica is the current primary replica. On the other hand, the Harp distributed file system [21] uses the concept of *view changes*, first introduced in [10], to maintain a consistent view of the organization among an individual groups of replicas. In general, an elaborate algorithm is required for the primary copy protocol to guarantee a consistent view of the replica configuration. The reconfiguration mechanism entails a two-phase protocol similar to the two-phase commit protocol to attain consensus among the sites or replicas regarding the group configuration.

2.2 The Quorum Consensus Protocol

The quorum consensus (QC) algorithm [14] is an inherently simple protocol which guarantees data consistency against all non-byzantine failures [17], including network partitioning. QC has received much attention from researchers . In the QC algorithm, a non-negative weight is assigned to each replica of data object x. A read threshold RT and a write threshold WT are also defined for x such that both $2 * WT$ and $(RT + WT)$ exceed the total weight of all copies of x. A read quorum of x is any set of copies of x with a weight of at least RT; a write quorum of x is any set of copies of x with a weight of WT. By design, each write quorum of x has at least one copy in common with every read quorum or any other write quorum. A write operation to x, write(x), is translated into a set of Write operations on each copy of *some* write quorum of x; while a read(x) is translated into a set of reads on each copy of *some* read quorum of x. In addition, a version number is maintained by each copy. With each read or write, the maximum version number returned by the copies in a quorum is then the latest version number and the copy or copies associated with this version number contain(s) the most current value of x. For a Write, all copies in the write quorum are updated and also assigned an incremented version number. As the weights assigned to the copies can be seen as "votes" from them, the algorithm is also known as the *Voting* algorithm. In its simplest form, a quorum is a majority set – in which case the algorithm is known as *Majority Voting* [27].

QC is an elegant algorithm which provides one-copy-serializability even in the presence of network partitioning. It does so without requiring complicated management of network configurations among the copy sites which is required for other replica control algorithms such as virtual partitions [11] or Primary Copy [28, 21]. It also does not require special treatment to recover copies, as outdated copies will not have the latest version number and so will not be read but will be over-written. In spite of these advantages, the QC algorithm has some well-known drawbacks [5]. First, it requires multiple reads for each read. Second, it requires a large number of copies $(2n+1)$ to tolerate n site failures, and, finally, all copies of x must be known in advance (to determine the weights and hence quorum configurations). The first drawback is due to the requirement that for each read operation, QC dispatches multiple read requests, and requires that a read quorum responds before the operation can be completed. In comparison, algorithms such as the Primary Copy require only the reading of one replica, which may be located on a site which minimizes the access time (that is, the primary copy is a local copy, or a replica at a site local to the transaction.) We argue that in a client-server environment (the kind of environment that is becoming prevalent and hence considered in this paper), where the concept of local copies is not applicable and all replicas are located at remote sites, the advantage of local copies no longer holds. Moreover, since multiple reads dictated by the QC protocol are performed in parallel, the response time for the multiple reads should not far exceed that for a single read, assuming that the sites are load-balanced and there is no excessive message processing latency. The second drawback will be generally present in all algorithms that tolerate partitioning failures. Finally, the third drawback has been addressed by some enhancements to QC which allow dynamic reconfiguration of the quorums.

3 QC Implementation Approaches

We consider QC to be a viable replica control algorithm. Next, we evaluate different implementations of the QC protocol

In QC, the coordinator translates each read and write on a data item x into reads and writes on copies: A write(x) is translated into a set of writes on each copy of *some* write quorum of x; while a read(x) is translated into a set of reads on each copy of *some* read quorum of x. Note that for each read or write, it is sufficient to perform the operation on one particular quorum, although the operation can be performed on a superset of the quorum.

Although it is straightforward to define a quorum, the implementation for performing an operation on a quorum is less well defined. Although the basic idea is for the coordinator to dispatch an operation to a set of the copy sites such that responses are received from at least a quorum, it is not clear how this translates in coding. In the rest of this section, we describe four implementation approaches to gather a quorum for each operation. We start with the most naive approach and subsequently propose alternative approaches, ending with one which offers both efficiency and high data availability. Later, we describe simulation experiments that were conducted to evaluate these approaches.

3.1 Implementation Approach 1: Send-To-All

In the most naive approach to implement QC, the coordinator dispatches each operation to all copy sites, advancing to the next operation as soon as responses from a site quorum are gathered. The algorithm is shown in Figure 1.

Let Q be the set of all possible quorums for the operation.
 Dispatch operation to all copy sites;
 R = ϕ;
 Start timer for timeout;
 While (not timed-out) and (R does not contain any member in Q)
  If (response from site s received), $R = R + s$;
 If (timeout) abort transaction;

Figure 1: Send-to-All Implementation

Note that the timeout is used by the coordinator to avoid indefinite blocking in the event of site failures. The coordinator waits for a predetermined timeout period after it dispatches the operation requests, at which point it aborts the transaction if responses from at least one quorum of copy sites has not been collected during the period.

An obvious drawback of the above approach is the large number of messages, $2m$, required, where m is the number of copy sites for the object accessed in the operation. Let q be the quorum whose responses are collected. Although only $\mid q \mid$ responses from the copy sites are gathered by the coordinator, up to m, $m \geq \mid q \mid$ responses are collectively sent by the replica sites. Thus $m - \mid q \mid$ of the responses for each operation are unneeded. These excessive messages are undesirable, since they contribute to message processing latency. Moreover, these excess responses cannot be simply ignored by the coordinator. For example, on a system which uses locking for concurrency control, a lock for the object accessed in an operation may have been acquired at a copy site s whose response arrives too late. If the response is simply ignored by the coordinator, it is possible for the object to remain locked at s indefinitely. There are two alternatives to handle the situation: (i) the coordinator can generate a message to s to cancel the operation, thereby allowing the lock associated with the operation, if any, to be released at s as soon as possible; (ii) the coordinator includes s in the transaction's membership set, that is, the set of sites that are to be involved in the atomic commitment protocol at the end of the operations.

Alternative (i) adds to the message overhead. In addition, measures must be taken to ensure that the operation-cancelling message eventually arrives at s even in the presence of failures. There are other complications as well with this approach: Site s will have to maintain a record of the lock(s) acquired for each operation of a transaction, so that it will know which specific lock to release when an operation-cancellation message arrives. For fault tolerance, such a record must be non-volatile; that is, it requires a disk access after each operation. On the other hand, alternative (ii) inflates the transaction's membership set, which adds to the message and disk-access overhead during the atomic commitment protocol for the transaction, and increases the probability of the transaction being blocked indefinitely if failures occur during the atomic commitment. Hence, the naive approach, although simplistic on the surface, leads to complications and also leads to excessive message and disk-access overhead.

3.2 Implementation Approach 2: Send-As-Needed

In the previous approach, the coordinator sends out m requests for each operation, when, in the absence of failures, only $\mid q \mid$ such requests ($\mid q \mid \leq m$) are necessary during the execution of the transaction to maintain one-copy serializability. However, in order to achieve low transaction response time, most implementors will instead choose to access only the minimum number of copy sites during the transaction's execution. Hence, rather than dispatching a request to all copy sites, an alternative approach is for the coordinator to send it to a randomly selected quorum q. If the responses do not return after a timeout period, then additional sites, if any, will be contacted until responses from a quorum has been obtained. The operation fails when no quorum has responsed. The algorithm is shown in Figure 2.

When there are no failures, only $\mid q \mid$ operation requests and responses are needed for the i^{th} operation, resulting in a total of $2q$, messages for each operation in the transaction. If the size of each quorum selected is the smallest possible, then the message overhead is optimal using this approach. Unfortunately, this approach has the same complications as the previous approach if failures are considered. Complications arise when a selected quorum for an operation fails to respond collectively: compensating actions are then necessary to address the late or missing responses, just as with the previous approach.

```
Let Q be the set of all possible quorums
of copy sites which hold the accessed object.
R = φ;
Repeat
  select q ∈ Q such that most of q's members are in R;
  Q = Q − q;
  send operation request to the sites in q;
  wait (for a timeout period), adding all sites which responses to R;
until (R contains a quorum) or (Q == φ);
If (Q == φ) and not all responses received
  operation has failed; abort transaction;
```

Figure 2: Send-As-Needed Implementation

3.3 Implementation Approach 3: Send-to-Random-Quorum-Only

One way to avoid the complications mentioned with the previous approach is simply not to attempt any retry. That is, for each operation a random quorum is chosen by the coordinator to dispatch its requests. If all sites in the quorum respond, then the coordinator proceeds with the next operation. Otherwise, the operation has failed and the transaction is aborted. The algorithm is shown in Figure 3.

```
Let Q be the set of all possible quorums of copy sites which hold the accessed object.
   Randomly select q ∈ Q;
   Send operation request to the sites in q;
   Wait (for a timeout period) or (for all responses to have arrived),
     whichever comes first;
     If (not all responses received)
     operation has failed; abort transaction;
```

Figure 3: Send-to-Random-Quorum-Only Implementation

The message overhead of this approach is optimal, as no excessive operation requests are dispatched by the coordinator. This algorithm also results in a much simplified implementation. With the approaches previously discussed, a late response from a copy site may arrive while the transaction is still active but the controller has proceeded to a subsequent operation. In contrast, under this approach, a late response from a copy site always arrives when the transaction has already been aborted. As a result, the normal protocol for aborting a transaction will ensure the releasing of the locks acquired for the transaction at all participant sites, including the late-responding copy site(s). Consequently, no special measures are required to cancel the original request at the late-responding copy site(s).

While this simplified approach reduces the complexity of the implementation considerably, there is a major drawback. During a failure, a randomly chosen quorum may include one or more sites that have failed. Such a selection will lead to the abort of the transaction when responses from the failed sites can not be collected by the coordinator. As a result, this approach can lead to unnecessary data unavailability during failures, which is precisely what the provision of replicated data is supposed to enhance. Hence, applied indiscriminately, this approach is unacceptable in terms of data availability.

3.4 Implementation Approach 4: Estimated-Uplist

In this approach, rather than randomly selecting a quorum set for each operation, the coordinator makes its selection based on a list of copy sites with which it assumes it can communicate. We term such a list an *estimated uplist*. The algorithm can then be expressed as follows:

Let Q be the set of all possible quorums of copy sites which hold the accessed object,
and U be the set of sites in the current estimated uplist.
randomly select $q \in Q$ such that $\forall s \in q, s \in U$;
If (no such q exists), then the operation has failed; abort transaction;
else
send operation request to the sites in q;
wait (for a timeout period) or (for all responsed to be arrived)
If (not all responses received)
operation has failed; abort transaction;

Figure 4: Estimated-Uplist Implementation

Assuming that the estimated uplist is reasonably current, each quorum selected by a coordinator will contain sites that are accessible. Consequently, the coordinator should be able to gather the necessary set of responses to render the operation successful even in the presence of failures. We argue that the information for a transaction coordinator site to compile the estimated uplist used in our algorithm is available on most modern-day distributed computer systems. On the Internet, for example, hosts constantly exchange messages to monitor topological changes: the maintenance of such information is necessary for the purpose of network routing [6]. Furthermore, fault-tolerant systems such as the Tandem systems use a scheme where hosts regularly exchange "I'm alive" messages to monitor the status of each other [7]. Furthermore, many distributed algorithms, such as the primary copy protocol, also make use of a configuration of communicable hosts which is, in their case, managed by the protocol itself.

Given that the information for compiling an uplist is already available, we assume that the compilation of such a list at each coordinator site does not add significantly to the overhead of maintaining replicated data. Note that we do not require that the uplist be accurate, hence the term "estimated uplist." Although, as we shall see later, it is desirable for the list to be as accurate as possible, the correctness of the execution of the transaction is enforced by the QC protocol itself.

3.5 Discussion

As with the send-to-random-quorum-only approach, the estimated-uplist approach simplifies the implementation of QC. However, in order for the approach to be acceptable, it should yield data availability that is at least comparable to that of the other approaches. In the following discussion, we quantify data availability as the ratio of the number of committed operations over the total number of attempted operations during a simulation run.

In the absence of failures, most operations can be expected to complete successfully under all four approaches, including the estimated-uplist approach. Under the estimated-uplist approach, an operation can complete successfully even if the uplist is not up-to-date, so long as the uplist contains a quorum. We therefore expect the data availability under Approach 4 to be comparable to that under the other approaches. Due to their smaller message overhead, one may expect Approaches 2-4 to perform better in terms of transaction throughput (to be defined later) than Approach 1.

The comparison is less obvious in the presence of failures. It is conceivable that Approaches 1 and 2, because of their provision for retries, may yield better availability than Approaches 3 and 4, both of which abort a transaction if a single attempt to gather a quorum fails. With Approach 4, the probability for the success of an attempt to gather a quorum depends on the accuracy of the uplist: An uplist which erroneously indicates that site s is communicable will lead to the abort of a transaction when its controller dispatches a request to s. On the other hand, this risk of aborting transactions unnecessarily disappears once the uplist has caught up with the current status of the copy sites. Hence, soon after the occurrence of a failure, there will be a period during which a number of transactions may be aborted unnecessarily due to out-of-date information in the uplist. These unnecessary aborts will reduce data availability, but their negative effects may be offset by the savings in overhead.

4 The System Model

To study the issues raised above, we employed a distributed database simulation testbed to carry out the experiments. The simulation model described in this paper was implemented using the general-purpose simulation programming language MODSIM II from CACI Products Co., La Jolla, California [9]. MODSIM II is a high-level language which supports object-oriented programming and discrete-event simulation. The model is a large, process-based simulation model. Each site is an object, as is each transaction. Methods are defined for these objects to implement the functional modules of the model. Statistics for individual transactions are carried in the data fields of the objects for the transactions, and are accumulated using statistic procedures provided by MODSIM II. In our experiments, the means of the desired measurements are obtained by using the method of *batch means* [19, 13, 18]. The results reported in the paper are within the 90 percent confidence intervals for the quantities measured: the size of the confidence intervals of those measurements was within a few percent of the mean in almost all cases.

Our testbed models a loosely-coupled, distributed system consisting of a fixed number of database server sites and a fixed number of database client sites. Sites are connected by a communication network. Each site has nonsharable volatile storage as well as nonvolatile storage (disks) and one or more CPUs. Each server manages and stores a subset of the database, known as a *database partition*. The database is a collection of files which are replicated. The database partitions are disjoint and the mapping from database partitions to servers is known to all sites. Only the server with a file in its storage can directly access the file. Client sites operate independently of the server sites. For this study, the two sets of sites are mutually exclusive. A transactions emanates from a client site, which is the coordinator site of the transaction. Each client site has disk storage (for logs) and one or more CPUs, but contains no data partition. Table 1 summarizes the key parameters of the database model:

Parameter	Meaning
NumSites	Total number of sites in system
NumClientSites	Number of client sites in the system
NumFiles	Total number of files in the database
FileSize_i	Number of pages in file i

Table 1: Key Parameters of Database Model

In our experiments, each transaction is composed of a set of operations each of which is a tuple of the following form:

$$\langle Operation, Filenumber, BeginningPageNumber, NumberOfPages\rangle\ .$$

An operation is either a read or a write applied to the specified page(s) in the specified file. The operations associated with a transaction are ordered and are executed sequentially. Transactions are generated by a global module with exponentially distributed interarrival time. Once generated, a transaction is assigned to a client site according to a uniform distribution. Table 2 summarizes the key parameters associated with the generation of transactions.

Parameter	Meaning
NumClasses	Number of classes of transactions
ArrivalRate	Intertransaction arrival rate
ClassFrac_i	Fraction of transactions of class i
FileCount_i	Number of files accessed
WriteProb_i	Probability of writing a file without reading
UpdateProb_i	Probability of reading then updating a file
NumWritePages_i	Average number of pages written for each file
NumReadPages_i	Average number of pages read for each file

Table 2: Transaction Generation Key parameters

Note that multiple classes of transactions are supported: each class can be associated with a set of attributes: ClassFrac, FileCount, WriteProb, and UpdateProb. For example, a class of read-only transaction can be generated by associating with it the attributes of WriteProb=0 and UpdateProb = 0, while a separate class of transactions each of which both reads and writes can be generated by associating with it a non-zero WriteProb and Updateprob. When a transaction is generated, it is randomly assigned (based on FileCount) a value for the number of files it will access; for each such file, it is then randomly determined (based on WriteProb and UpdateProb) if the file will be read from/write to. Hence the average length of a transaction in a particular class, in terms of the number of operations it contains, is a function of the input parameters FileCount, WriteProb, and UpdateProb for that class. Each site holds three types of physical resources: CPU, disks (for file storage), and log disks (for system logs). Forced logs are executed on the log disk, which supports fast sequential access. Furthermore, each site maintains a cache for the data disk access. Table 3 summarizes the key parameters associated with the system resources.

Parameter	**Meaning**
NumCPU	Number of CPU per site
NumDisks	Number of data disks per site
MinDiskTime	Minimum disk access time
MaxDiskTime	Maximum disk access time
HitRate	Cache hit probability
InitWriteCPU	Time to initiate a disk write
MsgCPUTime	Time to process a message before sending or after receiving
LogDiskTime	Sequential log write time
LogPageSize	Number of log records per page

Table 3: System Resources Key Parameters

The model simulates the occurrence of failures using a Poisson distribution with a mean failure arrival rate and a mean failure duration. Failures do not overlap at one site; that is, a site will not be subject to more than one failure at a time. Failures are uniformly distributed among the server sites. Failures are *fail-stop* [25]: all activities at the failed site cease upon a failure. Key information needed for the recovery of the site is assumed to be retained in stable storage. Site failures are implemented by interrupting all activities occurring at the failed site. The data structures and the resources at the site are then manipulated to simulate the effect of a site failure. Site recoveries are implemented in a "coarse-grain" fashion. The model does not maintain a log for each site. Upon recovery, a site aborts all transactions that were not in a *prepared* state. All other transactions are recovered by waiting for a period of time (set at three times the message propagation time) to simulate the communication delay required for the recovered site to enquire about the status of the transaction, then committing or aborting the transaction as appropriate. Table 4 summarizes the key parameters associated with site failure generation:

Parameter	**Meaning**
SiteFailRate	Interarrival rate of site failures
SiteFailLen	Time for a site to be down during repair

Table 4: Site Failure Key Parameters

Our model assumes a local-area network where the sites are fully connected. Messages are reliably delivered in first-in, first-out order. The key parameters are described in Table 5.

4.1 Functional Modules and Parameter Settings

Figure 5 describes the functional modules of the system model. Each module is briefly described as follows:

Parameter	Meaning
MsgPropTime	Message propagation time
CommTimeout	Time out interval for communication timeout

Table 5: Communication Key Parameters

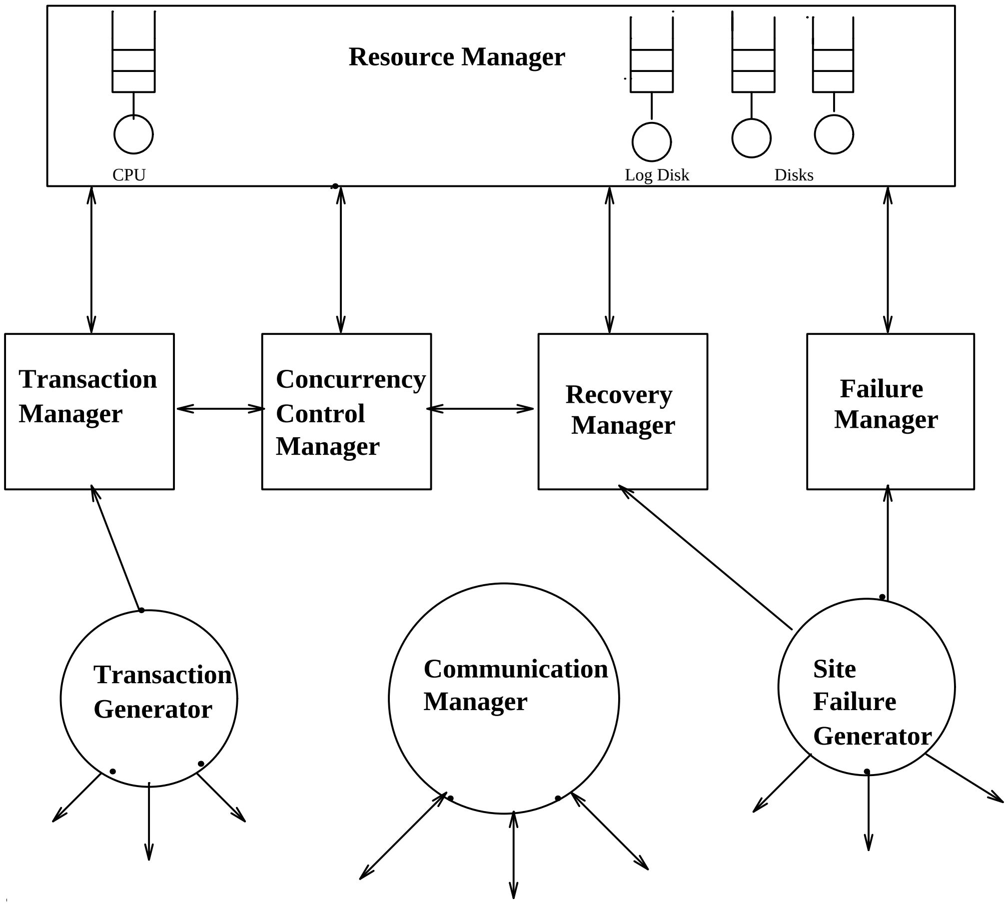

Figure 5: Functional Modules of Simulation Model

1. Transactions Generator: This is a global module which generates transactions and distributes them to the client sites. Transactions are generated independent of the processing at the client sites.

2. Site Failure Generator: This global module generates site failures and distributes them to the server sites. Failures are generated independent of the processing at each sites.

3. The Communication Manager: This global module serves as a switch for routing messages from one site to another.

4. The sites: Each site is a module which in turn consists of the following components:

   (a) Transaction Manager: At each client site, this component is responsible for overseeing the execution of a transaction originating from that site. At each server site, this component is responsible for accepting transaction requests from the coordinator site.

   (b) Concurrency Control Manager: At each server site, this component provides concurrency control for transactions that have been submitted to the site. For our model, the strict two-phase locking algorithm [12, 5] is used. Deadlocks avoidance is implemented by using a timeout interval based on the following heuristic [2] :

   $$TimeoutInterval = \mu(G) + k * \sigma(G)$$

   where $\mu(G)$ is the average lock request response time, $\sigma(G)$ is the standard deviation of the response time, and k is a weighting factor. Hence $TimeoutInterval$ is dynamically adjusted to reflect on-line estimation of lock request time.

   (c) Resource Manager: At each site, this component coordinates the consumption of site resources such as CPUs, disks, and log disk.

   (d) Failure Manager: At each server site, this component takes appropriate actions to effect a fail-stop failure upon receiving a site failure from the Site Failure Generator.

   (e) Recovery Manager: At each server site, this component coordinates the activities necessary to recover transactions interrupted by site failures. After-image logging is assumed for transaction recovery.

Parameter settings is a difficult issue in simulating a distributed environment. Due to the large number of parameters, it is not possible to vary all of them in our experiments. Typically, only one or two key parameters are varied while all other parameters are held constant. In our study, we will vary the transaction length, that is, the number of operations per transaction. For our experiments, the key parameter settings are described in able 6.

Our choice of parameters for the database and transactions are chosen so as to be comparable to the existing simulation studies such as [8]. The choice of message propagation time is in keeping with [26] and with a recent study [1] conducted on Sun workstations interconnected via ethernet and employing Unix UDP protocol. The scale of disk access time relative to message propagation time is in accordance with [15].

For the tractability of the simulation, the number of sites and the database size are set to smaller values than systems in practice. Likewise, the failure rate and duration are greatly exaggerated in our experiments: Practical systems experience failure rates and failure lengths measured in terms of months and hours respectively, which are not commensurate with the time scale (in seconds) of our simulation model. Thus, our experimental results are obtained from a small system with exaggerated failures, and some of the results (those from runs with failures) therefore reflect a worst-case scenario. At the other extreme, the results obtained from runs without failures reflect a best-case scenario on the same system. Since we are primarily interested in comparing the

Parameter Settings			
Parameter	**Setting**	**Parameter**	**Setting**
NumSites	12	NumClientSites	4
NumFiles	32	FileSize_i	200 pages
NumReplicates	2		
NumClasses	2	ArrivalRate	8-16 per time unit
ClassFrac_i	.5	FileCount_i	6
WriteProb_i	0.2, 0.0	UpdateProb_i	1.0, 0.0
NumWritePages_i	1.0, 0.0	NumReadPages_i	1.0, 1.0
NumCPU	1	NumDisks	1
MinDiskTime	0.01 time unit	MaxDiskTime	0.03 time unit
HitRate	0.8	InitWriteCPU	0.002 time unit
LogPageSize	100 log records	LogDiskTime	0.01 time unit
MsgCPUTime	0.004 time unit	MsgPropTime	0.001
SiteFailRate	0-0.2 failure per time unit	SiteFailLen	2.0 time unti
CommTimeout	5.00 time unit		

Table 6: Key Parameter Settings

performances of the protocols in the no-failure case against their performance in the presence of failures, we believe the two sets of results, representing the two extremes of the spectrum, serve to provide insights to the behavior of these protocols on a practical system.

In summary, the model is configured as follows:

- The system simulates 12 sites, four of which are client sites and the remaining are database sites.
- There are 32 files of 200 pages each. Each file has three replicas which are evenly distributed among the server sites.
- At each site, strict two-phase locking is applied to individual pages of the files for concurrency control.
- Transactions are generated using Poisson distribution rate of 16, then assigned to the client sites at a uniform distribution.
- Transactions are evenly distributed between two classes: a read-only class and one which may contain writes. Each transaction accesses a number of files, and for each file accessed, one page is written to or read from.
- Each site has its own set of resources, including a CPU, a disk for data storage, and a log disk. Disk access time averages 0.02 time unit. To simulate buffering, a hit rate of .8 is assumed for disk access.
- Sites exchange messages at an overhead of 0.001 time unit for message propagation, and 0.004 time unit for message processing.
- Server sites fail at a Poisson distribution rate of 0.02. Failures are uniformly distributed among the server sites and are *fail-stop*, i.e., all activities at the failed site cease upon a failure. Key information needed for the recovery of the site is assumed to be retained in stable storage.

- Communication timeout is set to 5 time unit.

4.2 Implementation of the QC Approaches

- For all four approaches, we implement quorums as majority sets.
- With the Send-All approach (Approach 1), locks acquired for excess operation requests are freed at commit time.
- With the Send-As-Needed approach (Approach 2), each read/write from/to x is translated by a coordinator into a set of reads and writes from/to x dispatched to a randomly selected set of q sites in a predetermined, global replica list, where q is the size of a majority of the replica set of x. If $n < | q |$ responses are received after a timeout period, the coordinator dispatches $| q | - n$ additional requests to the next copy sites in the replica list. As with the Send-All approach, locks acquired for excess operation requests are freed at commit time.
- With the Random Quorum approach (Approach 3), a coordinator dispatches an operation to a randomly selected set of Q copy sites of the accessed object.
- With the Estimated-Uplist approach (Approach 4), the *uplist*s at all sites are updated after a period t after each site failure/recovery. We set t at three times the message propagation time to approximate the time required for a probe to detect the failure/recovery. The coordinator dispatches an operation to a randomly selected set of Q copy sites that are "up" according to the *uplist* at the coordinator's site.

4.3 Performance Measurements

We measured the following quantities:

1. Response time of a committed transaction: This is a measurement of the elapsed time between when a transaction is initiated at its home site and when it is completed at the home site.
2. Number of messages per transaction: This is a measurement of the total number of messages sent divided by the number of completed transactions.
3. System throughput: The number of committed transactions per run divided by the time duration of the run.
4. Data availability: The number of committed operations over the total number of operations issued.

5 Simulation Experiments

Our objective is to find a viable and efficient implementation for Quorum Consensus. In the absence of failures, Approaches 2 through 4 should have comparable performances, since any chosen quorum will result in a successful operation (assuming that an adequate timeout is used by the coordinator). Under Approach 1, the operations will also be successful, but the transaction response time may be longer due to the additional message overhead. Less predictable are the performances of the four approaches when site failures are taken into account. Our experiments were designed to measure such performances. The following subsections describe the simulation results. We will first describe the results of a base case of failure rate set to 0.06. Subsequently, we contrast the results with those obtained by reducing the failure rate to 0 (that is, no failure) and also those obtained by increasing the failure rate to 0.2.

5.1 Messages per Transaction

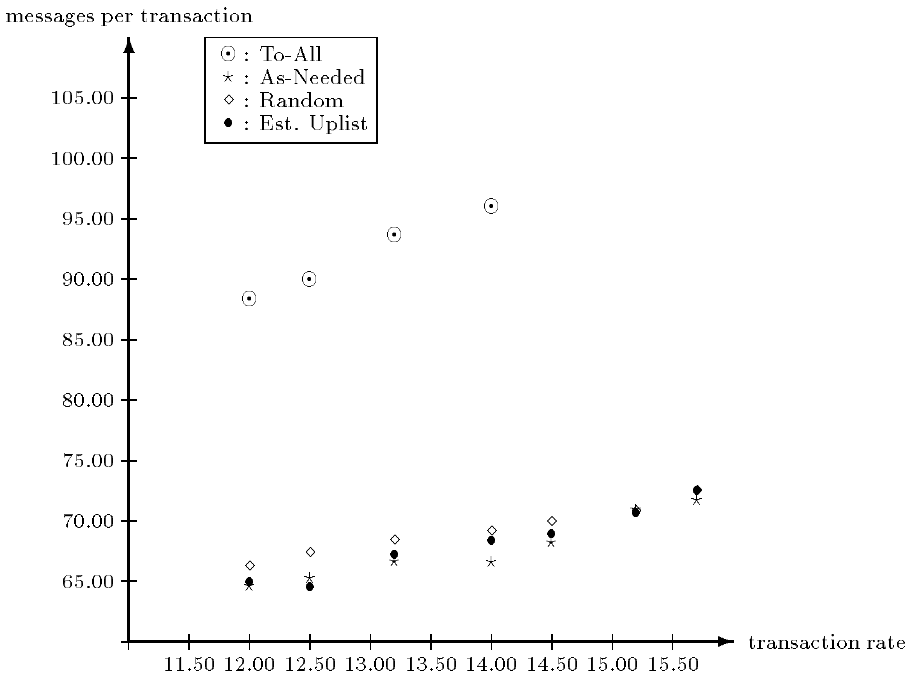

Figure 6: Messages per Transaction, FailRate=.06

Our experiment results highlight the excessive message overhead imposed by the Send-to-All approach. Compared with the other three approaches that we studied, it requires by far more number of messages per committed/aborted transaction, as illustrated in Figure 6 At transaction rate = 12, the Send-To-All Approach averages 87 messages per transaction, compared to 65 for the Estimated uplist approach. At transaction rate = 14, the gap is increased: it is 95 for the Send-To-All Approach compared to 65 for the Send-As-Needed approach and 67 for the Estimated-Uplist approach. With all four approaches, the number of messages per transaction rises as the transaction arrival rate increases, as the contention causes more and more transactions to abort. The increase under Approach 1 is the most pronounced.

5.2 Transaction Response Time

Our experiments indicate that the Estimated-Uplist approach has a lower average transaction response time than Send-To-All and Send-As-Needed. It is, however, higher than that of the Send-To-Random-Quorum approach, as shown in Figure 7. The Send-To-All approach consistently has the highest response time, as a result of the CPU bottleneck generated by the excessive number of messages required for each operation of a transaction.

As expected, the Send-As-Needed approach has a higher response time than the Estimated-Uplist approach. The findings are consistent with our qualitative analysis of these protocols: Under the Send-As-Needed approach, during failures, each operation of a transaction is likely to require a second round of messages in the coordinator's attempt to gather a quorum, and so the average response time of a committed transaction is likely to exceed that of the Estimated-Uplist, which always requires only one round of messages for quorum collection. It may be surprising that Approach 3 yields a better response time than the Estimateded-Uplist approach. The explanation

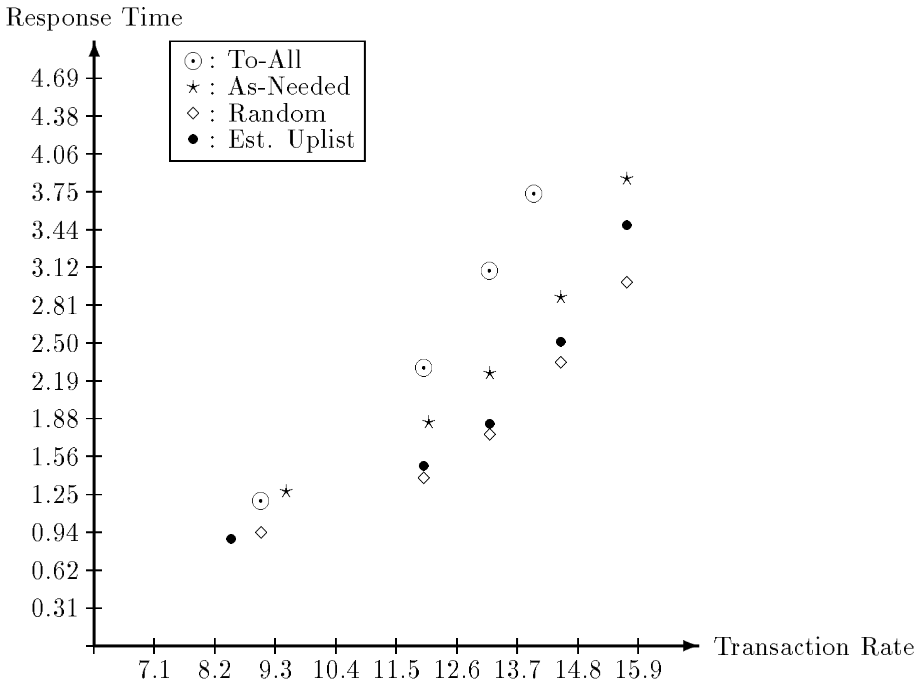

Figure 7: Transaction Response Time,FailRate=.06

of this outcome will be provided in the next subsection, when we analyze the throughput results.

5.3 Throughput

In our experiments, the Estimated-Uplist approach and the Send-As-Needed approaches yield the highest throughput, with the Estimated-Uplist being slightly higher.
Fig. 8 compares the throughput of the four approaches. The better throughput of the Estimated-Uplist approach in comparison to the Send-As-Needed approach is attributable to the smaller number of messages it requires in the presence of failures. The low throughput of the Send-to-All approach is again a result of the excessive message overhead that this approach imposes. Finally, the throughput of the Approach 3 is consistently lower than the Estimated-Uplist approach. Recall that Approach 3 selects a random target set for its quorum gathering, and aborts the transaction if the entire target set does not respond before the timeout. In our model, given the relatively large number of operations (≥ 6) in each transaction, the probability of a timeout is very high during a failure. Thus, in our experiments, under this approach, most of the transactions that are processed during a failure are aborted. This results in the low overall throughput. It also explains the low response time reported in the previous subsection: The transactions that are committable during failures are most probably small in number of operations. Such transactions have short response times, which in turn leads to lower overall transaction response times. In other words, while the Send-To-Random-Quorum approach appears to yield a better average response time, it only does so at the expense of low transaction throughput.

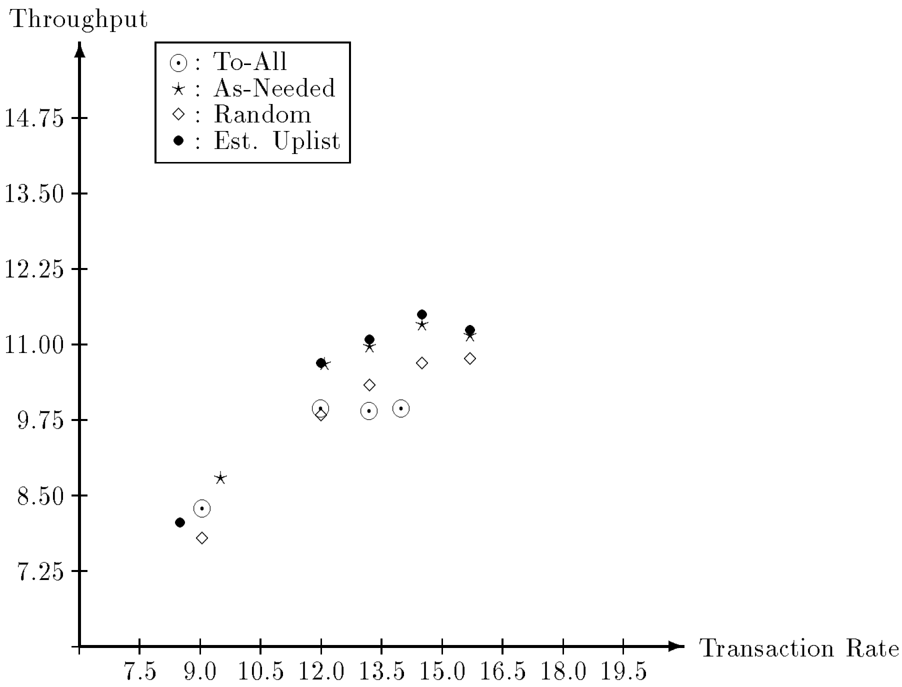

Figure 8: Transaction Throughtput,FailRate=.06

5.4 Data Availability

Figure 9 illustrates the data availability under the four different approaches. The send-as-needed and estimated-uplist approaches are comparable in availability, which that of the other two approaches (to-all and random) are lower in general. Thus our results indicate that the better performance of the estimated-uplist approach in terms of response time and throughput is not compromised by inferior data availability.

5.5 Results, Failure Rate = 0.2

The results above were obtained on our simulation system with failures generated at a poisson rate of 0.06. Interested in how the frequency of failures affect the outcomes, we repeated the experiments with the failure rate increased to 0.2. The results are shown in Figure 10.

Here we see that the performance of the send-to-all approach becomes more erratic when failures are frequent, but its response time consistently lacks behind the other approaches, and its throughput is inferior in general. The send-to-random-quorum approach still enjoys a deceptively better average response time, but again at the expense of low throughput. The send-as-needed approach has data availability which is generally close to that of the estimated-uplist approach, but its response time and throughput are considerably inferior. The results indicate that the estimated-uplist approach remains optimal when the failure rate is increased from 0.06 to 0.2.

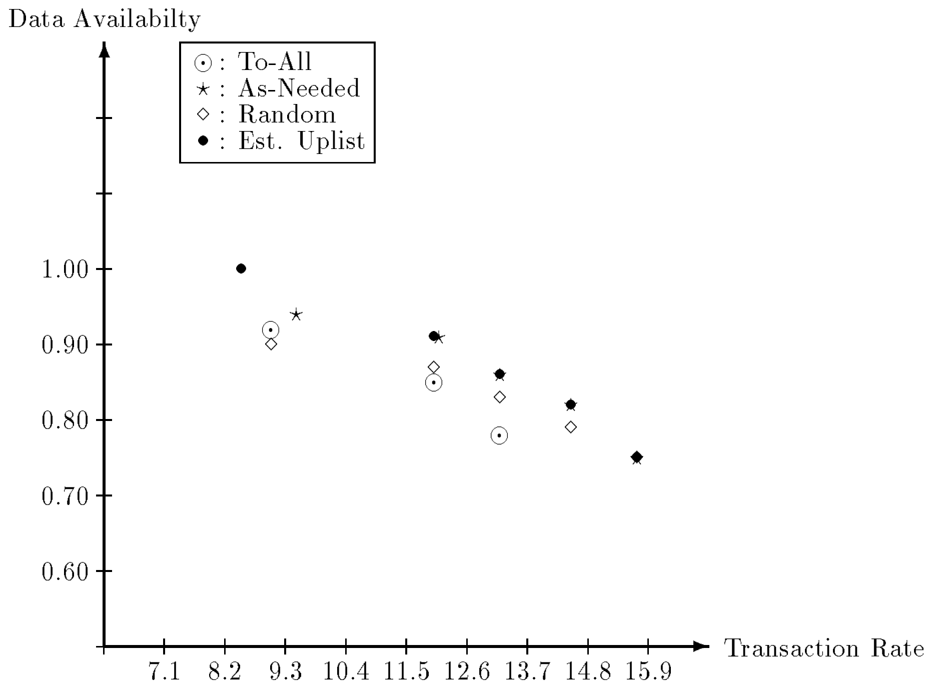

Figure 9: Data Availability,FailRate=.06

5.6 Results, No Failures

While the performance of a protocol in the presence of failures is important, it will be of little value if the protocol performs poorly when there is no failure. Thus, we again repeated our experiments to measure the behaviors of the four implementation approaches when failures are absent. In the absence of failures, the four approaches are equivalent in terms of data availability. Figure 11 illustrates the response time, throughput, and data availability of the four approaches when there are no failures.

Note that the results for the send-as-needed and the send-to-random approaches are the same under the circumstances, and their results vary little from those of the estimated-uplist approach. This is to be expected, as the algorithms for the three approaches essentially converge when all sites are up. In contrast, the message overhead of the to-all approach remains detrimental in the absence of failures. Hence our experiment results verify that the estimated uplist approach remains efficient during normal operation as well.

6 Conclusions

A simulation model was used to evaluate the viability of a simple but efficient way to implement the Quorum Consensus protocol for replica control. The implementation approach employs an estimated list of "up" sites to send out requests to one specific quorum set only. This approach simplifies the implementation for the collection of a quorum for each operation in a transaction. Simulation results show that this approach yields better performance and at least comparable availability when contrasted with other implementation approaches of the protocol.

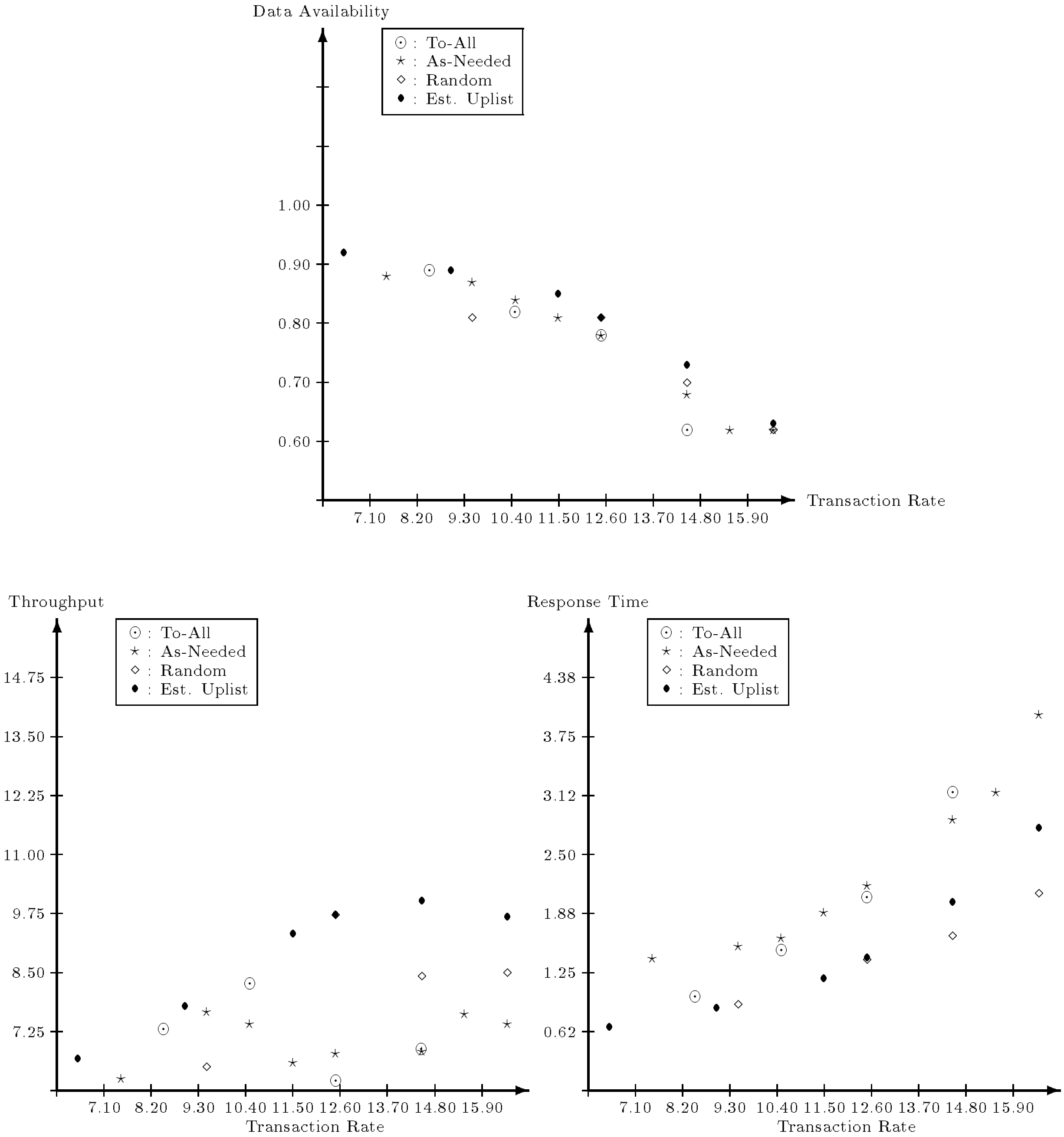

Figure 10: Performances, failure rate = 0.2

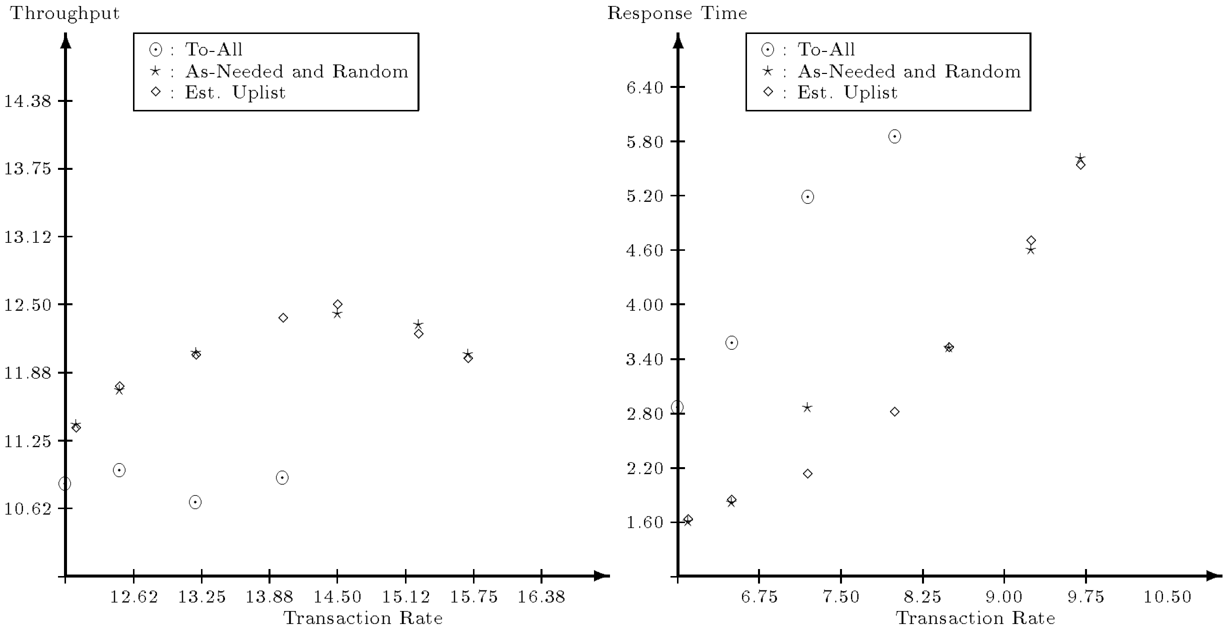

Figure 11: Performances, No failures

References

[1] Debbie A. Agarwal. Personal Communication on Ethernet Performance Measurements. Department of Electrical and Computer Engineering, University of California, Santa Barbara, May 1993.

[2] R. Agrawal, M. Carey, and M. Livny. The Performance of Alternative Strategies for Dealing with Deadlocks in Database Management Systems. *IEEE Transactions on Software Engineering*, SE-13(12):1348–1363, December 1987.

[3] P.A. Alsberg and J.D.Day. A principle for resilient sharing of distributed resources. *Proceedings of the Second International Conference on Software Engineering*, October 1976.

[4] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, Reading, Massachusetts, 1987.

[6] D. Bersekast and R. Gallager. *Data Networks.* Prentice-Hall, 1992.

[7] D. Davcevand W.A. Burkard. Consistency and recovery control for replicated files. *Proc. 10th ACM Symposium of Operating Systems Principles*, pages 87–96, 1985.

[8] M. Carey and M. Livny. Conflict Detection Tradeoffs for Replicated Data. *ACM Transactions on Database Systems*, 16(4):703–746, 1991.

[9] CACI Products Company. *MODSIM II Reference Manual.* CACI, La Jolla, CA, 1993.

[10] A. El Abbadi, D. Skeen, and F. Cristian. A nonblocking quorum consensuss protocol for replicated data management. *Proceedings 4th ACM Symposium on the Principle of Database Systems*, March 1985.

[11] A. El Abbadi and S. Toueg. Maintaining Availability in Partitioned Replicated Databases. *ACM Transactions on Database Systems*, 14(2):264–290, 1989.

[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):624–633, November 1976.

[13] Domenico Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

[14] D. K. Gifford. Weighted Voting for Replicated Data. *ACM Transactions on Database Systems*, 16(4):150–159, 1979.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

[16] D.D.E. Long J.-F. Pâris and A. Glockner. A realistic evaluation of consistency algorithms for replicated files. *Proceedings 21st Annual Simulation*, pages 121–130, 1988.

[17] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Trans. Prog. Lang. Syst.*, 4(3):382–401, July 1982.

[18] S. Lavenberg. *Computer Performance Modeling Handbook*. Academic Press, 1983.

[19] Averill M. Law and David Kelton. *Simulation Modeling and Analysis*. McGraw Hill, 1991.

[20] M. L. Liu, D. Agrawal, and A. El Abbadi. The performance of two-phase commit protocols in the presence of site failures. *To appear in The 24th Annual International Symposium on Fault-Tolerant Computing*, 1994.

[21] B. Oki and B. Liskov. Viewstamped Replicataion: A New Primary Copy Method to Support Highly-Available Distributed Systems. *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[22] M. T. Ozsu and P. Valduriez. Distributed Database Systems: Where Are We Now? *Computer*, 24(8):68–78, August 1991.

[23] J.-F. Pâris and D.D.E. Long. The performance of available copy protocols for the management of replicated data. *Proceedings 11th Performance Evaluation*, pages 9–30, 1990.

[24] A. Reuter. Performance and Reliability Issues in Future DBMSs. In *Lecture Notes in Computer Science: Database Systems of the 90s*, pages 294–315. Springer-Verlag, 1990.

[25] R. Schlichting and F. B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1982.

[26] M. Scott and A. Cox. An Empirical Study of Message-Passing Overhead. In *ICCC 7th International Conference on Distriubted Computing Systems*, pages 536–543, 1987.

[27] J. Seguin, G. Sergeant, and P. Wilms. A majority consensus algorithm for the consistency of duplicated and distributed information. *Proc. IEEE Int. Conf. Distributed Computing Systems*, pages 617–624, 1979.

[28] M. Stonebraker. Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRESS. *IEEE Transactions on Software Engineering*, pages 188–194, May 1979.

[29] M. StoneBraker and E. Neuhold. A distributed data base version of ingres. *Proceedings 2nd Berkeley Workshop on Distributed Data Bases and Computer Networks*, May 1977.