The Performance of Two-Phase Commit Protocols in the Presence of Site Failures

M. L. Liu D. Agrawal A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

Much of the existing literature on the *two phase commit* protocol is restricted to discussing and analyzing the protocol (and its variants) in the absence of failure. Very little, especially in quantitative terms, has been written about its performance in the presence of site failures. In this study, we use a simulation testbed of a distributed database system to quantify the differences in the performances of four 2PC protocols (the generic 2PC, *presumed commit*, *presumed abort*, and *early prepare*). Our study covers both the no-failure case and the case of site failures. We present a number of interesting experiment results. One is that the performance of these protocols is highly dependent on the message-processing latency at the transaction coordinator site. Another is that the presumed abort protocol does not necessarily yield better performance in the presence of site failures.

1 Introduction

In distributed database systems, an *atomic commitment* protocol is needed to terminate global transactions consistently in the system. In particular, the protocol must ensure that all sites involved in the execution of a transaction either all commit or all abort the transaction. Much has been written about *atomic commitment* protocols. In practice, the commonly used protocol of this genre is the *two phase commit* (2PC) protocol [9] and a host of its variants, such as the *presumed commit* [13], *presumed abort* [13], and the *early prepare* [19] protocols. In general, these protocols attempt to minimize execution overhead, in terms of message traffic and log writes, to optimize performance and costs. Although the *two phase commit* protocol has been studied extensively for a long time, the topic is of sufficient significance that several refinements of the protocol have continued to emerge [10, 8]. Furthermore, much of the existing literature on this protocol is restricted to discussing and analyzing the protocol (and its variants) in the absence of failures. Very little, especially in quantitative terms, is available about its performance in the presence of site failures.

In a distributed database system, a transaction T originates at a site, which forwards the operations contained in T to the appropriate database sites where the data is stored. Depending on the outcome of the operations, a *commit* or an *abort* operation is issued to all the participating sites to terminate T. To preserve data integrity, it is necessary for all participating sites to consistently perform a single logical action, either commit or abort. An *atomic commitment protocol* is an algorithm which ensures this consistency. In the absence of failures, such an algorithm can be easily devised by a simple exchange of messages. However, additional measures must be introduced into the algorithm when failures are taken into consideration, as any message exchange between processors is subject to disruption for an indeterminate time period. Unlike in a centralized environment where failures are all-or-nothing, failures in a distributed environment result in significant problems. In particular, the atomic commitment protocol must continue to work correctly when some sites have failed while others continue to function. The type of failures that occur in the distributed system also has a significant effect on the performance of the protocol. In particular, the two-phase commit protocol and its variants may block when a site fails, that is, the protocol may not be able to terminate due to the failure of some site, and the termination of the transaction may have to wait until such time when that site recovers. The three-phase commit protocol [17, 16, 18] (another type of atomic commitment protocol) overcomes this problem in the case of site failures, but may still block when network partitions occur.

In this paper, we concentrate on site failures and the 2PC protocols and its variants. We use a simulation testbed of a distributed database system to quantify the differences in the performances of the four 2PC protocols: the generic 2PC, *presumed commit*, *presumed abort*, and *early prepare* protocols. A host of metrics are used for the comparisons; the key ones include: (i) transaction response time, (ii) message count, (iii) log write count, (iv) restart ratio, and (v) system throughput. Our study covers both the no-failure case and the case of site failures. We present a number of interesting results related to 2PC and its variants.

2 Atomic Commitment Protocols

In this section, we review the two phase commit protocol for atomic commitment of transactions. In particular, we summarize the overhead of the two phase commit protocol in terms of the number of messages and forced writes to the log. Since message and log-write overheads delay the completion of a transaction, many modifications to the two phase commit protocol have been proposed. We review the three main variants: the presumed commit, the presumed abort, and the early prepare protocols.

2.1 Two Phase Commit

Two phase commit (2PC) [13] is the simplest *atomic commitment protocol*, and is the most widely used in practice. 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. In its generic form, 2PC proceeds in two phases. In the first phase, the transaction coordinator exchanges one round of messages with the participants to solicit a "vote" on whether to commit or abort the transaction. Commit must be by unanimous votes. In the second phase, the coordinator exchanges another round of messages with all participants to commit or abort the transaction as voted. No change is made in the state of the data at a participant site until the second phase, i.e., when the participant receives a commit or abort message from the coordinator. To allow for the possibility of failures, each site individually records in a *log* (stable storage that survives crash failure) when there is a state change in the 2PC protocol during its execution. Such a log is used by a site when it recovers from a failure to complete its role in 2PC in a manner consistent with the execution of the protocol during its failure. For instance, a *prepare* record is written to the log when a participant casts a *yes* vote in the first phase. Since failures can occur at any time, these log records must be force-written. That is, written immediately or "flushed" to a nonvolatile storage that survives failure. Hence they are referred to as *forced* log records.

The casting of a *yes* or *no* vote is a significant point in 2PC. A participant site can unilaterally decide abort prior to the point when it votes *yes*. After this point, a participant site is said to be in an *uncertain* state, and it must wait until it hears the decision from the coordinator to abort or commit accordingly. Should a failure occur after a participant has entered the uncertain state, special recovery action for the transaction must be taken. The *prepare* record in the log serves as a reminder. Other state changes which require an entry in the log are:

- The coordinator writes a *commit/abort* record in the log after it has made a decision.
- A participant writes a *commit/abort* record in the log after it receives the coordinator's decision.

Let N be the number of read and write operations in the a transaction and C be the number of participants involved in the execution of the transaction, then 2PC requires $4C$ messages and $2C + 1$ number of log forces for a transaction which commits. (For an explanation of the message and log force counts, please refer to the Appendix.)

A transaction can abort during the execution of its operations or during its atomic commitment protocol. The former occurs when the execution of an operation fails (due to deadlocks or site failure, for example). The latter occurs, when, during the commit protocol, either (i) a site failure or communication timeout disrupts the protocol or (ii) when one or more "no" vote is received by the coordinator. The message and log-force requirement for transactions which abort during the atomic commit protocol is bounded by the requirement for committed transactions. For example, if a site failure occurs at a participant site after the transaction's operations have been executed, the coordinator's request for votes will timeout and the coordinator will decide to abort. In this case, another additional round of messages will be exchanged to abort the transaction, resulting in a total of 4C messages (minus one for each participant which failed to vote) and $2C + 1$ log forces (minus one for each participant which failed to vote). On the other hand, if a transaction aborts before commit (for instance, due to a failure which occurs during its execution), then no voting is necessary, and only one round of ($2C$) messages needs to be exchanged, accompanied by $C + 1$ log forces.

2.2 Presumed Commit

In the generic 2PC, a participant site does not update the data modified by a transaction until it receives a Commit from the coordinator. To ascertain that all participants have been informed of the decision (abort or commit), the protocol requires that the participants acknowledge the coordinator's abort/commit message in the second phase. These acknowledgements accounts for C of the $4C$ messages.

The Presumed Commit (PC) protocol [13] eliminates the need for these acknowledgements. At the point when a participant casts its vote to commit, it updates the data (however, it does not release the locks on the data; locks will be discussed later in this paper). When the coordinator receives all yes votes, it presumes that the data at all participant sites are already in a committable state, and thus requires no acknowledgement for its commit message. On the other hand, if the decision is to abort, the coordinator must wait for the acknowledgements from the participants to ascertain that any update to the data has been undone. By eliminating the acknowledgements in the second phase, PC reduces the message count to $3C$ in the case of committed transactions. As a result, the number of log forces is also reduced to $C + 2$. In the case of an aborted transaction, the protocol requires the same overhead as with the generic 2PC: $2C$ messages and $C + 1$ log forces.

2.3 Presumed Abort

Presumed Abort (PA) [13] is the counterpart of PC. The coordinator assumes that the participant sites are prepared to abort at the point when they cast their votes. That is, the participant sites make no update to the data until they receive a commit message from the coordinator. Thus, if the decision is to abort, the coordinator can assume that the data at all participant sites are already in an aborted state, and hence requires no acknowledgement for its abort message. As a result, an aborted transaction requires one less round of messages under PA than under 2PC. On the other hand, if the decision is to commit, the coordinator must gather acknowledgements from the participants, in which case its message and log force overhead is the same as that for 2PC.

2.4 Early Prepare

The Early Prepare (EP) protocol [19] goes one step beyond PC: the coordinator assumes that each participant site is prepared to commit after it receives an acknowledgement for each operation. That is, each participant site immediately updates the data written by the transaction before it acknowledges a write operation. Doing so allows the coordinator to eliminate the one round of messages in the first phase of PC. However, this reduction in message overhead comes at a cost: At the end of each operation, a participant site must ensure that the data is in a recoverable/committable state. There are two important ramifications: (i) A log force is now necessary following each operation (to record that the data is in a committable state), (ii) Since a participant site enters the uncertain state after each operation, the uncertain period, during which a communication failure requires special recovery action for atomic commitment, is longer than with the other 2PC protocols that we have discussed. For a committed transaction, C messages and $N+2$ log forces are required, where N is the number of operations in the transaction. For an aborted transaction, the message requirement is the same as that for 2PC, while the log force requirement is $N'+2+C$, where N' is the number of operations completed at the time of the abort.

2.5 Summary

Table 1 summarizes the message and log overhead under the four protocols. Note that N is the average number of operations performed by each transactions and C is the average of the number of participants involved per transactions.

Committed Transactions		
Protocol	**No. of Messages**	**No. of Log Forces**
2PC	4C	2C+1
PC	3C	C+2
PA	4C	2C+1
EP	C	N+2

Table 1: Overhead for Committed Transactions

As can be deduced from the table above, PC and EP have the lowest message overhead for committed transactions. For aborted transactions, however, the PA protocol has the lowest overhead. Since most transactions are expected to commit in the absence of failures, the cost of committing will dominate that of aborting. Thus, in terms of message overhead alone, EP clearly is the best among all atomic commit protocols. However, there is another factor which has to be taken into consideration: the overhead of log forces. The number of log forces that EP requires is order of N, the number of operations, while the log-force requirement for PC, PA, and 2PC is order of C, the number of participants. For transactions that are sufficiently long such that $N \gg C$, we expect PC to outperform EP. The tradeoff between message overhead and log forces is crucial in the performance of these protocols and hence this issue will be explored later in this paper. During failures, many transactions are likely to be aborted due to the loss of communication between a coordinator and a participant. For example, if a coordinator fails to receive a response for an operation from a participant site within a predetermined timeout period, it proceeds to abort the transaction. Since the PA protocol has the lowest overhead with aborted transactions, a reasonable question to ask is whether PA has a better performance when failures are considered. The answer is obviously no if failures occur infrequently, as any gain by the protocol during the infrequent outages is overwhelmed by its relative inefficiency when there are no failures. But what if failures are frequent? Will PA outperform the other protocols then? This is another issue that we will be analyzing through simulations.

3 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 [5]. 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* [12, 7, 11]. 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 not replicated for this study. 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 2 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 2: 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:

$$\langle Operation, FileNum, BeginningPageNum, NumOfPages\rangle \; .$$

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 3 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	Prob. of writing a file w/o reading
UpdateProb_i	Prob. of reading then updating a file
NumWritePages_i	Avg no. of pages written each file
NumReadPages_i	Avg no. of pages read each file

Table 3: 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.

The model simulates the occurrence of failures at server site only using a Poisson distribution with a mean failure arrival rate and a mean failure duration. For our study, the client sites never fail. Failures do not overlap at one server 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* [14]: 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 5 summarizes the key parameters associated with site failure generation:

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 4: System Resources Key Parameters

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

Table 5: 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 6.

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

Table 6: Communication Key Parameters

4 Functional Modules and Parameter Settings

Figure 1 describes the functional modules of the system model. Each module is briefly described as follows:

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. A client site never fails.

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

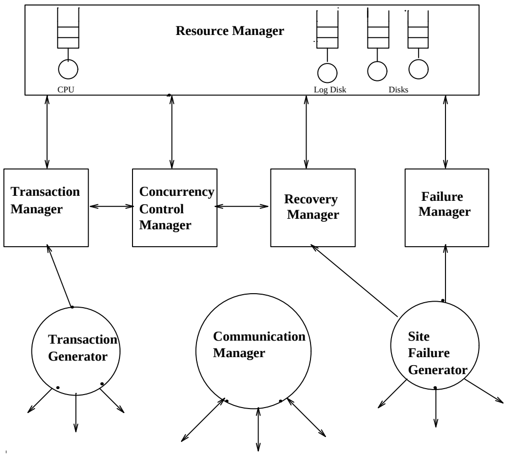

Figure 1: Functional Modules of Simulation Model

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 [6, 3] 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. A transaction waiting for more than the timeout period is aborted.

   (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 are 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. For example, in our experiments to observe thrashing, we will vary the multiprogramming level, that is, the transaction arrival rate. On the other hand, when we study the tradeoff between PC and EP, we will vary the transaction length, that is, the number of operations per transaction. For our experiments, the key parameter settings are described in Table 7.

Our choice of parameters for the database and transactions are chosen so as to be comparable to the existing simulation studies such as [4]. The choice of message propagation time is in keeping with [15] and with a recent

Parameter Settings	
NumSites	12
NumClientSites	4
NumFiles	32
FileSize_i	200 pages
NumClasses	2
ArrivalRate	16-50 per time unit
ClassFrac_i .5	
FileCount_i	2-28
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 ,2, 4
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	0.02 failure per time unit
SiteFailLen	2.0 time unit
CommTimeout	5.00 time unit

Table 7: Key Parameter Settings

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 [9].

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 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. Files are not replicated.
- 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.s

We measured the response time of a committed transaction which is a measurement of the elapsed time between when a transaction is initiated at its coordinator site and when it is completed at the same site. Commit time is a measurement which is the time it takes between the initiation of the commit protocol and when it is completed at the coordinator. For each committed transaction the averages of the number of messages and number of log forces is measured. We also compute the amortized message count per committed transaction which is a measurement of the total number of messages sent during a run divided by the total number of committed transactions during that run. This measurement differs from the average number of messages in that the message count reflects the overhead spent on aborted/blocked transactions. In the same vein we compute the amortized log forces per committed transaction. The restart ratio is the ratio of the number of aborted transactions over the number of committed transactions during a run. We measure the committed transaction throughput, which is the number of committed transactions per run divided by the time duration (in seconds) of the run. Similarly, the aborted transaction rate is measured primarily to analyze the Presumed Abort protocol.

5 Experiments and Results

5.1 Experiment 1: Thrashing

One of the first experiments we carried out, as a validation of our model, was to observe the well-known thrashing phenomenon in a database system which uses locking. Locking, as is used in our model, causes thrashing because of delays due to lock conflicts. As the number of transactions in the system increases, throughput (measured on committed transactions) rises up to a point, then falls.

Figure 2 is a throughput curve plotted from experiment results in the case of CPU=2, FileCount=20, and no failures. The number of transactions in the system is increased by varying the transaction rate. Note that in this setting, the arrival rate of 16 yields the highest throughput for the given parameter settings.

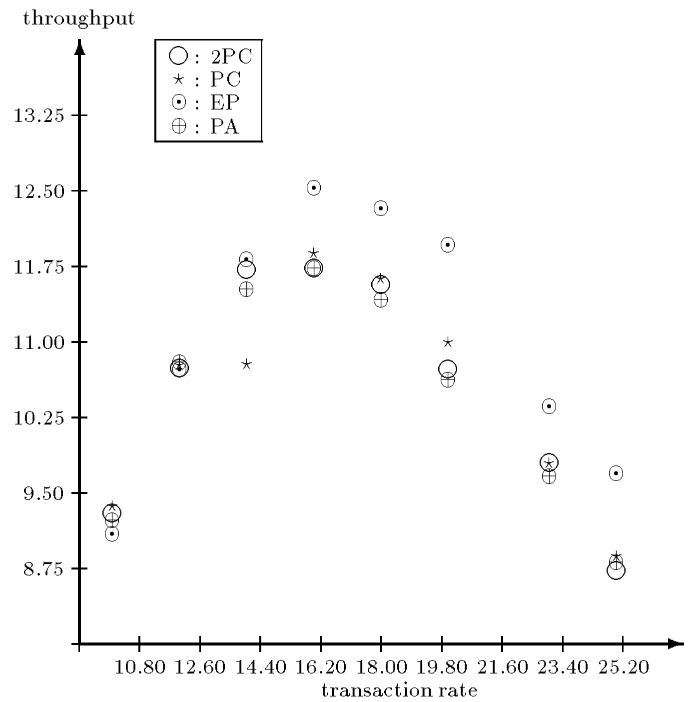

Figure 2: System Throughput:CPU=2;FileCount=20

5.2 Experiment 2: Performance of the Protocols in the Absence of Failures

Based on our earlier discussion, it is clear that the EP and the PC protocols, because of their low message and log-force overheads, should yield better performances in the absence of failures. Less obvious is the tradeoff between these two protocols as the transaction lengths increase.

Simple calculation based on the message propagation time, message processing time, and disk access time would indicate that, as the transaction length increases, the overhead for the forced logs under EP should eventually mitigate its savings in message overhead. Thus one would expect the PC protocol to yield better throughput than EP for transactions beyond a certain length.

Figs. 3 shows our experiment results when the four protocols are applied to transactions of increasing lengths, while the transaction arrival rate is held at 16. The arrival rate of 16 is chosen because it is the rate which yields the maximum throughput in Fig. 2 and hence the results are expected to reflect the performance of the protocols under key load.

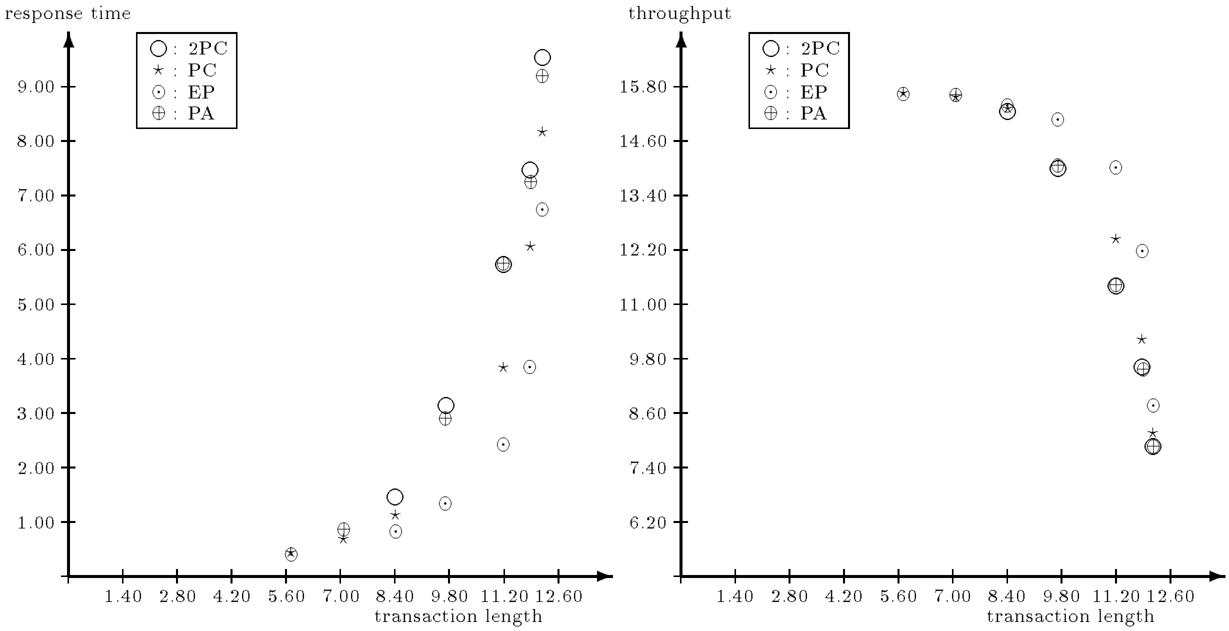

Figure 3: Protocol Performances, No Failures,CPU=1

Contrary to our expectation, no crossover between the EP and the PC protocols occurred. EP continues to have better response time and throughput even as the transaction lengths are increased. To explain this paradox, we traced the execution of a transaction consisting of seven operations executed under EP and PC. The time lines of the two executions show that, under PC, there is a much higher delay between the arrival of a message and its processing at the coordinator site, that is, the message processing latency at the coordinator site is significantly higher under PC as compared to under EP. This is a direct consequence of the larger number of messages required for PC, each of which must be processed by the CPU at both its sender site as well as its receiver site. Under PC, our experiments show that the average waiting queue length at the client sites far exceeds its counterpart under EP. As the transactions become longer, that is, the number of operations in each transaction increases, the log-force overhead indeed increases under EP. At the same time, however, the message latency rises under PC to greatly increase its message overhead. Furthermore, the increase in message latency under PC is concentrated at the coordinators site, while the rise in forced logs under EP is distributed among the participants sites. In our model, the increase in log-force overhead under EP is less dramatic than the increase in message overhead under PC. Hence the lack of crossover. Note that this result arises in spite of the order-of-magnitude difference between the log-force time(0.01) and the message propagation/processing time (0.001/0.004 respectively) in our model.

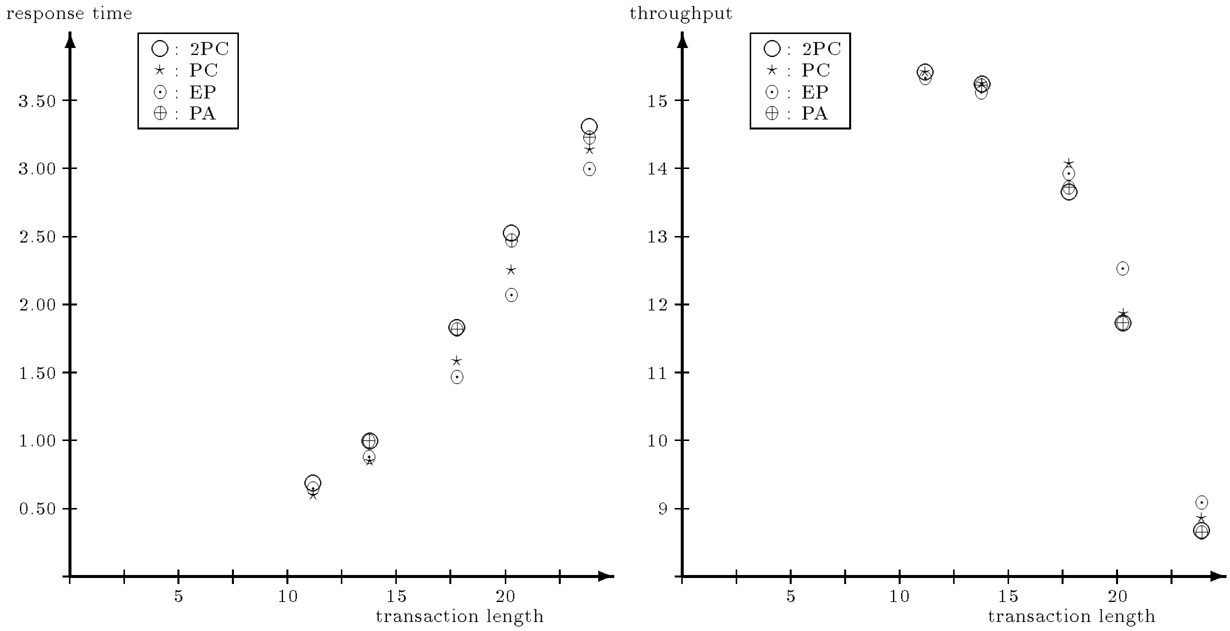

Figure 4: Protocol Performances, No Failures,CPU=2

One way to reduce message processing latency at the coordinator site is to increase the availability of CPU. We repeated the above experiment with the number of CPUs at each coordinator site set to 2 and then again at 4. Figure 4 shows the response time and throughput when CPU is set to 2. At transaction length of 14 and over, the behaviors of the protocols follow the same pattern as in the case of CPU=1: EP, incurring the least amount of message processing latency at the coordinator site, enjoys faster response time and highest throughput in comparison to the other protocols. However, there is a change in the pattern when the transaction length is 14 or less. With the increased availability of CPU, the message processing latency is sufficiently reduced when transactions are short (≤ 14), so that the greater log overhead associated with EP now becomes a significant factor. Consequently, EP's performance no longer dominates, and, in fact, is slightly inferior to that of PC. For example, EP's response time is 0.85 for transaction length 11, compared to 0.65 for PC. This result is counter-intuitive, since we had expected EP to perform better than PC when N is small. This can be explained because the static analysis does not account for the queuing delays and overheads that occur at runtime.

When the number of CPUs at each coordinator site is further increased to 4, the message processing latency is effectively eliminated under all protocols, for transactions of length up to 27. As a result, PC consistently exhibits the best performance in figure 5. Based on this set of experiment results, we conclude that the EP protocol will generally yield better performance if the message processing latency at the coordinator sites of the transaction is significant. Otherwise, there is a crossover point between the EP and the PC protocols as the transaction length increases, beyond which PC dominates.

5.3 Experiment 3: Performance of the Protocols in the Presence of Failures

In transaction processing, it is generally desirable to choose an atomic commit protocol which optimizes performance in the no-failure case, since failures are expected to be infrequent in the normal mode of operation.

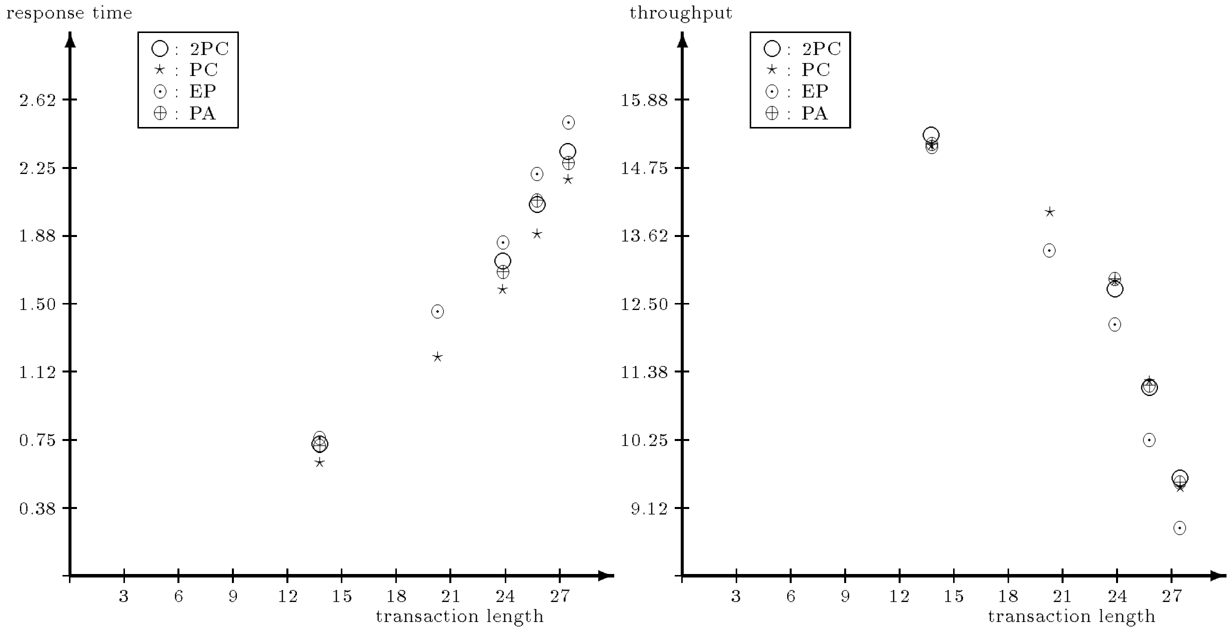

Figure 5: Protocol Performances, No Failures,CPU=4

We have already seen that, in the no-failure case, the message complexity of the protocol is a key consideration. Whether the same holds when failures are taken into consideration is not immediately apparent.

In the presence of failures, a large number of transactions can be expected to abort. Specifically, all transactions which involve a failed site will be either aborted by the coordinator when it fails to receive a response from a participant site, or blocked if the coordinator cannot proceed until it has collected all the necessary responses.

All four protocols that we have looked at are subject to blocking. In 2PC, a transaction for which the coordinator has decided commit/abort will block until all acknowledgements are received from the participants. In PC and EP, this blocking occurs only when the coordinator has decided abort, while in PA it occurs only when the coordinator has decided commit. In EP, because a participants immediately enters the uncertainty period after each operation, the probability of a transaction becoming blocked is increased when the participant site is subject to failure. Hence there is some reason to believe that the performance of EP may be negatively affected by failures. Yet another factor which may affect the comparative performances of these protocols when failures are considered is the relative efficiency of the PA protocol in processing aborted transactions. Since aborted transactions are expected to be numerous during a failure, it is reasonable to expect the performance of PA to improve during these periods.

To evaluate the performance of the protocols when failures are taken into account, we injected failures into the model, using the technique and parameter settings as described in Sections 3 and 4 We obtained results from a set of experiments with CPU=1 at all sites and with the parameter FileCount set to 4 for all transactions. To increase the multiprogramming level for the model, we varied the transaction arrival rate.

Our experiment results are presented below in figures 6 through 8.

In Figure 6 we contrast the response time of the protocols when failures are absent and present respectively. Note that the response time for EP is roughly equivalent in the two cases, whereas the response time for the other protocols is significantly reduced in the presence of failures. During failures, a large portion of the active

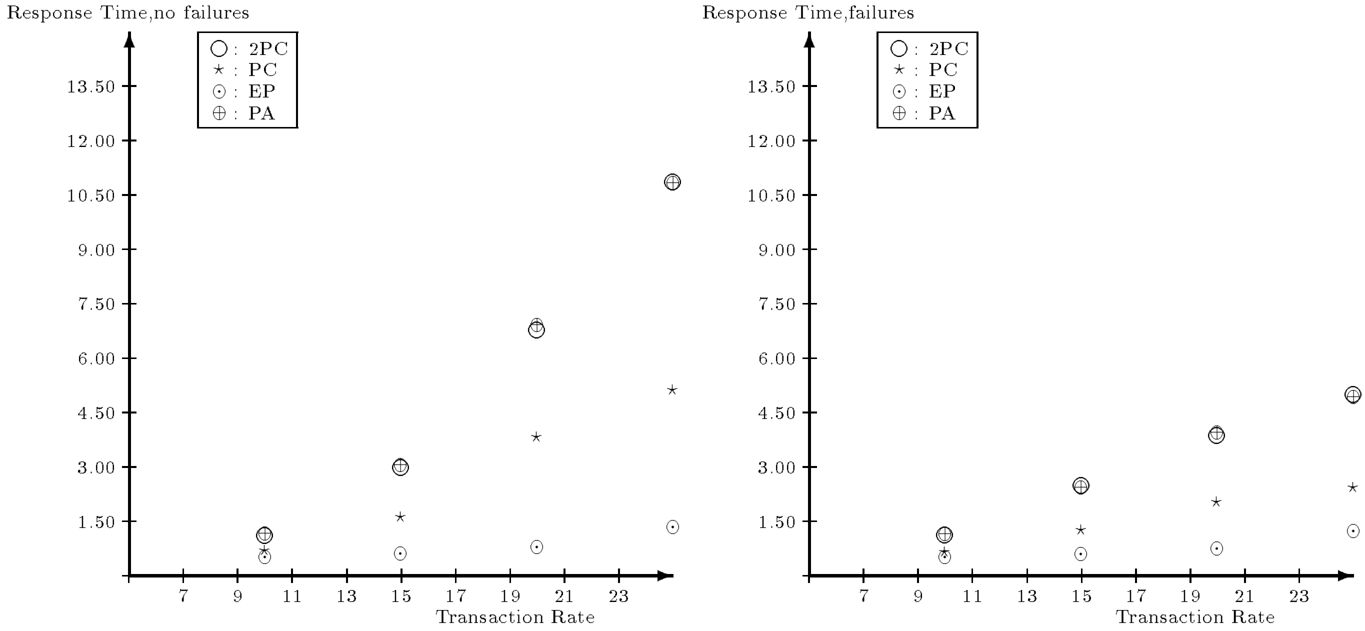

Figure 6: Response Time, no failures vs. failures

transactions are aborted, in effect reducing the multiprogramming level. For the 2PC, PC, and PA protocols, the large number of aborted transactions during failures result in a significant reduction in message processing latency at the coordinator sites. Benefiting from the reduced message processing overhead, the committed transactions thus enjoy a better response time when failures are included. Since the message processing latency is far less under EP, as described in Section 5.2, the failures make very little difference on the transaction response time.

Figure 7 compares the throughput of the committed transaction in the no-failure and the with-failure cases. As can be expected, all four protocols show a drop in the average number of committed transactions per time unit. Although the failure rates are exaggerated in our model, the ratio between the total amount of failure time over the total amount of run time is small, hence the reduction in throughput during failures does not result in a pronounced drop in the overall throughput. Thus, the committed throughput under EP ranges from 9.9 to 23.9 when transaction rate increases from 10 to 25, but drops to 9.7 and 23.2 respectively when failures are introduced. Likewise, PA's committed throughput falls from 16.9 when there is no failure to 16.2 with failure, when the transaction rate is 25.

Note that EP remains the best performing protocol when failures are considered, especially when the multiprogramming level is high. In particular, PA never outperforms EP. Among 2PC, PC and PA, the committed transaction throughput is comparable with or without failures; however, PC shows a slightly better throughput when there are no failures (see data points at transaction rate = 25), while PA gives the best throughput when failures are considered (16.14 for PA, 15.66 for PC, and 15.9 for 2PC, for transaction rate = 25). Hence our results show that the gain in performance by PA during the (relatively brief) durations of the failures is not sufficient to offset its relative inefficiency (due to message processing latency) during normal operation, but is sufficient to give it a slight advantage as compared to PC.

To observe PA's relative efficiency in aborting transactions, we measured the average aborted transaction throughput under the four protocols, as shown in Figure 8. Under EP, there are few aborted transactions when

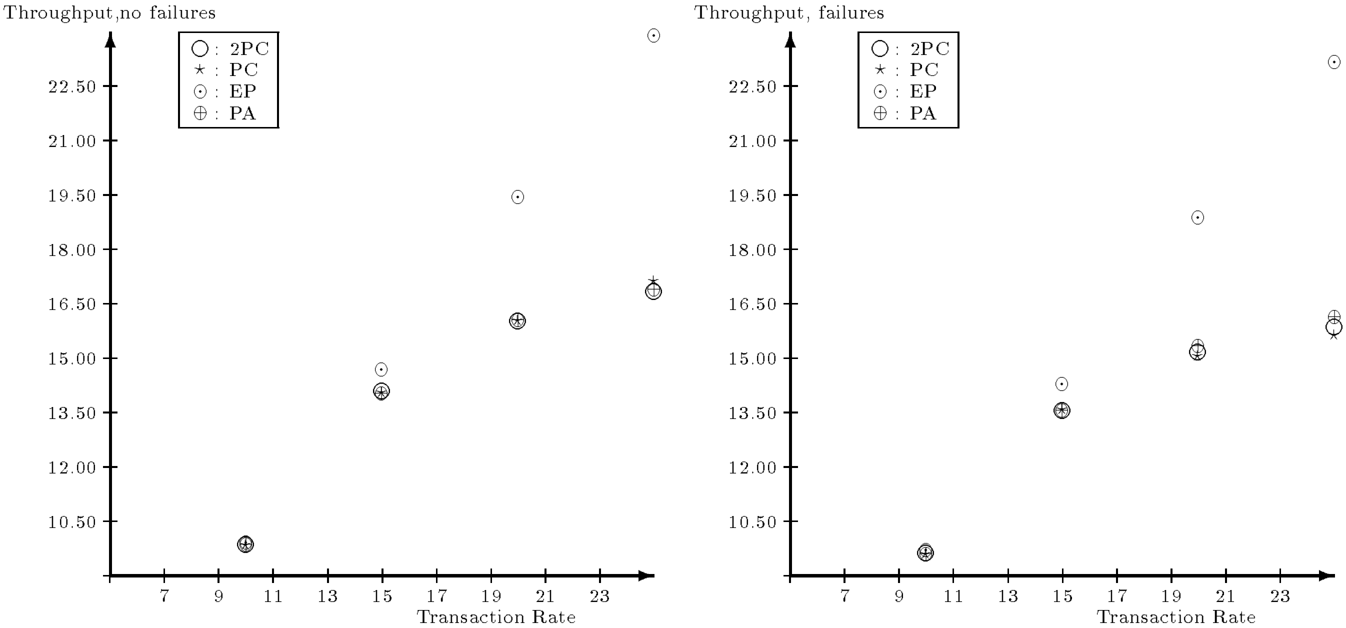

Figure 7: Committed Transaction Throughput, no failures vs. failures

there are no failures; while under the other protocols there are increasing number of aborted transactions as the multiprogramming level increases. At transaction rate=10, the aborted throughput is insignificant under all four protocols. At 15, EP's aborted throughput is 0.08, while that for the others have risen to about 0.7. At 20, EP's aborted throughput is 0.6 compared to over 3.5 under the other protocols. The gap widens at transaction rate=20, with the aborted throughput at 7.6, 7.3, 0.6, 7.5 respectively. The relatively large number of aborted transactions under the non-EP protocols in the absence of failures is due to the previously described message processing latency, which results in the timeout of some transactions. As the latency grows worse when the number of concurrent transactions increases, the gap widens between EP and the other protocols.

When failures are introduced, the aborted throughput increases under all four protocols. In the case of EP, the abort throughput ranges from 0.23 at transaction rate=10, to 1.15 at transaction rate=25. With PA, the range is 0.34 to 8.5. The ranges for PC and 2PC are (0.29,8.3) and (0.32,8.75). Although the differences are slight, the results do show that, among the non-EP protocols that we tested, the PA protocol yields better throughput, both for committed and aborted transactions, when failures are present. We attribute this slight improvement in throughput under PA to its relative efficiency in handling aborted transactions.

The reason why EP's performance does not deteriorate significantly as a consequence of blocked transactions is as follows: In our model, failures are of a finite length. As a result, a transaction only stays blocked for a relatively short duration. Consequently, the response time of blocked transactions (which are small in numbers anyway) do not significantly impact the overall transaction response time. In fact, even when we exaggerate the failure rates and failure lengths for the model (by setting SiteFailLen to 0.2 and SiteFailLen to 5.0, resulting in virtually a continuous presence of one or more site failures), and shorten the transactions by setting FileCount to 2, PA never outperforms EP, as illustrated in Figure 9).

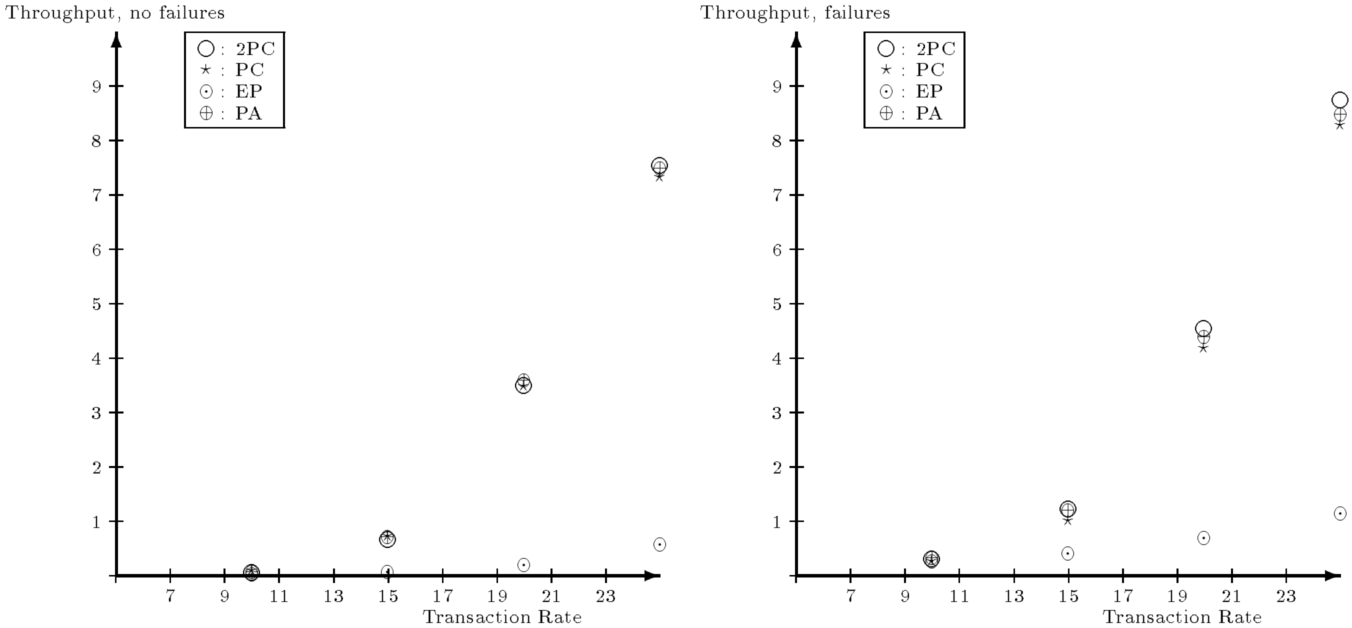

Figure 8: Aborted Transaction Throughput, no failures vs. failures

6 Conclusions

The conventional approaches optimize two phase commit protocols by minimizing the number of messages and/or log forces in the protocol. In this paper, we use a simulation system and demonstrate that the performance of 2PC is not solely linked with the number of messages and log forces. Our findings are that message overhead plays a significant role in the performance of 2PC. As a result, we find that for a broad range of database workload settings, a non- conventional atomic commitment protocol, Early Prepare, outperforms 2PC and its variants. Our study was conducted for both the cases of with and without failures. If, on the other hand, the effects of message overheads are offset through increased processing power (for example, multiple CPUs), the protocols with less number of log forces perform better. In conclusion, we would like to point out that our results highlight the dynamic aspects of real distributed database system, and hence captured by our simulation. These aspects are not accounted for by the more traditional static algorithmic analysis of protocols as was illustrated in Table 1.

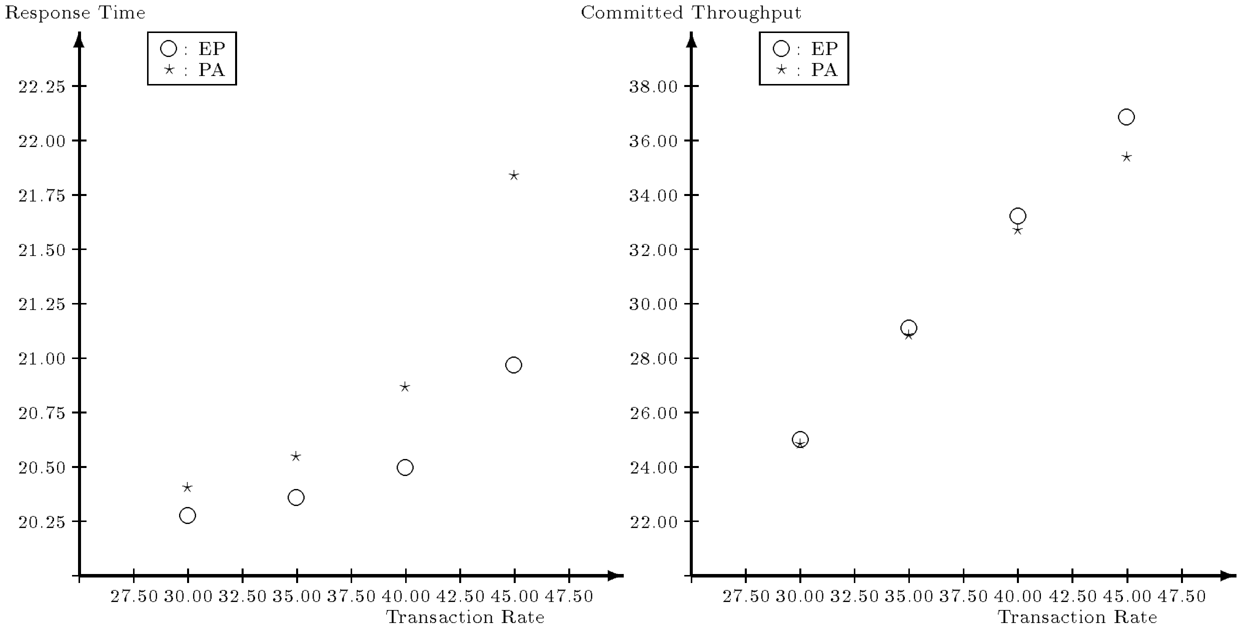

Figure 9: Performances, Extensive Failures, Short Transactions

Appendix

Outline of the 2PC Protocol and Its Variations

In this appendix, we outline the basics of 2PC and its variants [13][19].
(Note: N = number of operations in a transaction, C = number of participants involved in a transaction, N' = number of operations completed before a transaction aborts, and C' = number of participants involved in a transaction prior to its abort. TC = Transaction Coordinator)

1. **Execution Protocol using Two Phase Commit**
   For a committed transaction:
   (a) TC sends an *operation* message for each operation to the appropriate participant. (N messages are sent, but since they are not related to the commit protocol, they are not included in the protocol's message requirement.)
   (b) Each participant replies after executing the request.
   (c) TC sends a *prepare* message to each participant. (C messages)
   (d) Each participant votes yes, forces a *prepare* log record, and sends TC a *yes* vote. (C force, C messages)
   (e) TC forces a *commit* log record that contains the transaction membership and sends a *commit* message to each participant. (1 force, C messages)
   (f) Each participant forces a *commit* log record, sends an acknowledgement to TC, and commits the transaction. (C forces, C messages)

(g) TC appends (but *not* forces) to its log an *end-transaction* log record.

For an aborted transaction:

(a) TC sends an *operation* message for each operation to the participant, before the transaction is to be aborted.

(b) Each participant replies after executing the request.

(c) TC forces an *abort* log record, and sends an *abort* message to each participant. (1 force, C' messages)

(d) Each participant forces an *abort* log record, sends an acknowledgement to TC, and aborts the transaction. (C' forces, C' messages)

(e) Upon receiving the acknowledgements from the participants, TC appends to its log an *abort-transaction* log record.

2. **Execution Protocol using Presumed Commit**

For a committed transaction:

(a) TC sends an *operation* message for each operation to the appropriate participant.

(b) Each participant replies after executing the request. (N messages)

(c) TC forces a *membership* log record and sends a *prepare* message to each participant. (1 force, C messages)

(d) Each participant forces a *prepare* log record, and sends TC a *yes* vote. (C forces, C messages).

(e) TC forces a *commit* log record, sends a *commit* message to each participant, and forgets about the transaction. (1 force, C messages)

(f) Each participant appends to its log a *commit* log record, and forgets about the transaction.

For an aborted transaction:

(a) TC sends an *operation* message for each operation to the appropriate participant

(b) Each participant replies after executing the request. (N' messages)

(c) TC forces an *abort* log record, sends an *abort* message to each participant, and waits for their acknowledgements. (1 force, C' messages)

(d) Each participant forces an *abort* log record, sends an acknowledgement to TC, and aborts the transaction. (C' forces, C' messages)

(e) Upon collecting all the acknowledgements, TC appends to its log an *abort-transaction* log record.

3. **Execution Protocol using Presumed Abort**

For a committed transaction:

(a) TC sends an *operation* message for each operation to the appropriate participant.

(b) Each participant replies after executing the request.

(c) TC sends a *prepare* message to each participant. (C messages)

(d) Each participant forces a *prepare* log record, and sends TC an *yes* vote. (C forces, C messages)

(e) TC forces a *commit* log record, and sends a *commit* message to each participant. (1 force, C messages)

(f) Each participant forces to its log a *commit* log record, sends a *acknowledgement* message to the TC, and forgets about the transaction. (C forces, C messages)

(g) Upon collecting all the acknowledgements, the TC appends to its log an end-transaction log record.

For an aborted transaction:

(a) TC sends an *operation* message for each operation to the appropriate participant.

(b) Each participant replies after executing the request.

(c) TC forces an *abort* log record, sends an *abort* message to each participant, and forgets about the transaction. (1 force, C' messages)

(d) Each participant forces a *prepare* log record, and aborts the transaction. (C' forces)

4. **Execution Protocol using EP**

For a committed transaction:

(a) TC forces a membership record, then sends an *operation* message for each operation to the appropriate participant. (1 force)

(b) Each participant executes the requested operation, forces a *prepare* log record, and replies to the TC. (N forces)

(c) TC forces a *commit* log record, sends a *commit* message to each participant, and forgets about the transaction. (1 force, C messages)

(d) Each participant appends to its log a *commit* log record, commits the updates, then forgets about the transaction.

For an aborted transaction:

(a) TC forces a membership record and sends an *operation* message for each operation to the appropriate participant. (1 force)

(b) Each participant executes the requested operation, forces a *prepare* log record, and replies to the TC. (N' forces)

(c) TC forces an *abort* log record, sends a *abort* message to each participant, and waits for their acknowledgements. (1 force, C' messages)

(d) Each forces an *abort* log record, and sends an acknowledgement to the TC. (C' forces, C' messages)

(e) Upon collecting all the acknowledgements, the TC appends an *abort-transaction* record to its log, and forgets about the transaction.

References

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

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

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

[6] 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.

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

[8] A. Citron G. Samaras, K. Britton and C. Mohan. Two-Phase Commit Optimizations and Tradeoffs in the Commercial Environment. In *Proceedings Ninth International Conference on Data Engineering*, pages 520–529, April 1993.

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

[10] B. Lampson and D. Lomet. A New Presumed Commit Optimization for Two Phase Commit. Technical report, Digital Equipment Corp., February 1993.

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

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

[13] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM Transactions on Database Systems*, 11(4):378–396, December 1986.

[14] 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.

[15] 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.

[16] D. Skeen. *Crash Recovery in a Distributed Database Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 1982.

[17] D. Skeen. Non-blocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 133–147, June 1982.

[18] D. Skeen. A quorum based commit protocol. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, February 1982.

[19] J. Stamos and F. Cristian. A Low-Cost Atomic Commit Protocol. In *Proceedings of Ninth Synposium on Reliable Distributed Systems*, October 1990.