A Fully-Distributed Approach to Concurrency Control in Replicated Database Systems

Mukesh Singhal
Dept. of Computer and Information Science
The Ohio State University
2036 Neil Avenue Mall
Columbus, OH 43210

Abstract

Existing algorithms for concurrency control in replicated database systems are semi-distributed because in these algorithms only one site completely executes an update and other sites just commit its writes. In this paper, we present a fully-distributed algorithm for concurrency control where each site completely executes every update. This approach has several features such as improved resiliency to different kinds of failures, higher parallelism, fast response to user requests, and low communication overhead. We present a performance model of a replicated database system and use it to study the performance of the proposed algorithm and the algorithm in [20]. The results of the performance study reveal that the proposed approach improves the performance at the cost of nominal I/O overhead.

1. Introduction

In a distributed database system several users concurrently access and modify data objects by running updates. In order to increase system throughput, actions from concurrent updates are interleaved and allowed to operate simultaneously on the database. In such an environment, some control is necessary to restrict the way actions from several concurrent updates are interleaved. A concurrency control algorithm controls interleaving of conflicting actions so that integrity of a database is maintained.

In a replicated database system, data objects are redundantly stored at all sites to enhance the reliability and responsiveness of the system. However, updates in a replicated environment are expensive because they involve extensive communication for synchronization. Several algorithms have been proposed to synchronize updates in a replicated database system. Depending upon whether synchronization is performed before or after an update makes an access to data objects, these algorithms are divided into two classes. Algorithms that perform synchronization before accessing data objects are referred to as pessimistic algorithms, e.g., [1, 2, 3, 10, 12, 18, 19, 20, and 23]. On the other hand, algorithms that perform synchronization after accessing data objects are referred to as optimistic algorithms, e.g., [5, 6, 14, 19, 22, and 24]. Some other algorithms have appeared in [9, 16], which are fault-tolerant to certain kinds of failures.

In existing algorithms for concurrency control in replicated databases, when a user submits an update to a site, the site performs synchronization, executes the update, writes the computed values into its copy of the database, and broadcasts the computed values to all other sites. On receipt of these values, other sites write them into their copies of the database. Thus, only one site completely executes an update, and other sites just execute update's write action. Due to this mode of update execution, these algorithms are semi-distributed.

In this paper, we present an algorithm for update synchronization in replicated database systems, where every update (i.e., transaction) is distributed to each site which completely executes it, i.e., the algorithm is fully-distributed. The algorithm allows higher parallelism in update execution because once an update is shipped to all sites, they can execute the update in parallel. Every site is more autonomous in execution of updates and there is less reliance among the sites, which makes it more robust to different kinds of failures. A site computes values for every update which has two consequences: First, there is no transfer of values of the data objects (usually huge) among sites; hence, messages exchanged are usually short. Second, a site does not wait for computed values of the data objects from other sites which has an important consequence on the update response time.

The rest of the paper is organized as follows. Next section introduces terminology, notations, and assumptions. In Section 3, we describe the basic idea - update transport - underlying the algorithm. Sections 4 and 5 contain informal and formal description of the algorithm, respectively. In Section 6, we discuss the properties of the algorithm. Section 7 deals with with failure aspects of the algorithm. In Section 8, we study the performance of the algorithm. Section 9 contains concluding remarks.

2. Preliminaries

A database D is a collection of data objects (d1, d2, ..., dM). Each data object di can take values from a specified domain. The state of a database is given by the values of its data objects. In a distributed database system, data objects are spread over a collection of autonomous sites, say S1, S2, ..., SN, which are connected by a communication network. In a replicated database system every site Si has a copy Di of the database D. Let di denote the copy of data object di at site Si. Then, Di = (d1, d2, ..., dM) for i = 1, 2, ..., N.

Database Consistency: In database D = (d1, ..., dM), certain semantic relationships must hold among its data objects [11]. Set of these relationships for database D is called consistency assertions of D. Database D is in a consistent state if the current values of its data objects satisfy all of its consistency assertions. In a replicated database (D1, D2, ..., DN), the notion of consistency has two aspects: internal consistency and mutual consistency [24]. Internal consistency deals with the semantics of data objects within a single data-
A database $D_i$ is internally consistent if its value satisfies all of its consistency assertions. Mutual consistency requires that all database copies $D_1,D_2, ... D_n$ have the same value, i.e., $D_i = D_j = ... = D_n$.

**Update, Conflicts, and Update Processing:** A user modifies a database $D = \{d_1, d_2, ... d_m\}$ by submitting an update to a site. An update $U$ is defined by a 3-tuple $U=(RS(U), WS(U), t(U))$, where (a) $RS \subseteq \{1,2,3, ..., M\}$ indicates the data objects which are read by $U$ and is referred to as the readset of $U$, (b) $WS \subseteq \{1,2,3, ..., M\}$ indicates the data objects that are written by $U$, and (c) $t(U)$ is a function that models the computation done by $U$. We assume that function $t$ maintains internal consistency.

For an update $U=(RS(U), WS(U), t(U))$, if $RS(U)$ denotes the set of values to be assigned to the data objects in $WS(U)$ along with the identifiers of the data objects and is referred to as the write-set-value of $U$. When used in a set operation, if $RS(U)$ denotes the WS. We denote readset and writset of update $U$ by $RS(U)$ and $WS(U)$, respectively. For any two updates $U'$ and $U''$, $U'$ is said to have r-w, w-r, or w-w conflict with $U''$ if $RS(U') \cap WS(U'') = \emptyset$, $WS(U') \cap RS(U'') = \emptyset$, or $WS(U') \cap WS(U'') = \emptyset$, respectively. Updates $U'$ and $U''$ are said to conflict if at least one of these conflicts exists between them.

A site $S_i$ executes an update $U$ in the following manner: (a) $S_i$ reads the data objects in $RS(U)$ (this action is denoted by $R(U)$), (b) $S_i$ computes values for data objects in $WS(U)$, and (c) $S_i$ writes the computed values onto the data objects in $WS(U)$ of its database copy $D_i$ (this action is denoted by $W(U)$). In subsequent discussion, subscript is dropped where it is obvious from the context or is unimportant. We assume that readset and writset of an update are known at the time of its inception in the system.

**Timestamps:** Every site $S_i$ has a logical clock $C_i$ which takes a monotonically non-decreasing integer value [17]. When an update $U$ is submitted at a site $S_j$, the $S_j$ increments $C_j$ by one and then assigns 2-tuple $(c_j, i)$ to $U$. The 2-tuple is referred to as the timestamp of $U$ and is denoted by $t(U)=c_j$. Every message contains the current clock value of its sender site, and when a site $S_i$ receives a message with clock value $t$, it sets $C_j$ to $\max(t+1,C_i)$. For any two timestamps $t_1=(c_1, i)$ and $t_2=(c_2, j)$, $t_1 < t_2$, if either ($t_1 < t_2$) or ($t_1=t_2$ and $c_1 < c_2$).

**Communication Medium:** There is no globally shared memory in the system, and all sites communicate solely via message exchanges. The algorithm is first described under the assumptions that the communication medium provides a FIFO channel of finite delay between any pair of sites, no site fails or acts maliciously, and the network does not get partitioned. Later in this paper, we discuss the impact of site and communication medium failures on the operation of the algorithm and suggest measures to cope with them.

3. Basic Idea

The algorithm is based on the following approach to updates execution: When a site receives an update from a user, it assigns the update a timestamp and transports the update and its timestamp to all other sites. After it, every site executes that update completely without any exchange of computed values of data objects. That is, every site performs synchronization and executes the read, compute, and write actions of every update, which makes it a fully-distributed algorithm.

One can use any optimistic or pessimistic technique for synchronization in this algorithm. In this paper, we embed the synchronization scheme of the algorithm proposed in [20]. That is, a site executes read action of an update $U$ after:

(i) the site has received a message with a timestamp larger than $t(U)$ from every other site, and

(ii) write action of all updates having their timestamps smaller than $t(U)$ and having w-r conflict with $U$ have been executed at that site.

A site executes write action of an update $U$ after:

(iii) read action of all updates having their timestamps smaller than $t(U)$ and having r-w conflict with $U$ have been executed at that site, and

(iv) write action of all updates having their timestamps smaller than $t(U)$ and having w-w conflict with $U$ have been executed at that site.

Since the algorithm in [20] is based on the semi-distributed model of update execution, we refer to it as the SDA (Semi-Distributed Algorithm) and since the algorithm proposed in this paper is based on the fully-distributed model of update execution, we refer to it as the FDA (Fully-Distributed Algorithm) henceforth in this paper.

**Decoupling Effect:** Since every site computes the values to be written by an update, an important aspect of the FDA is that a site does not have to wait for computed values from other sites. Because of the absence of wait for computed values, there is a decoupling effect among the sites. As we discuss below, this decoupling has a significant impact on the response time of updates since it reduces blocking delays due to conflicts.

**Consequence of Decoupling Effect:**

We explain the effect of decoupling by comparing the blocking delays due to r-w conflict in the SDA [20] and the FDA. Since both algorithms use the same synchronization rules, any difference in their performance is attributed to their different style of update execution.

Consider an update $U$ at site $S_i$ for which the condition (i) holds, but the condition (ii) does not hold. This means $U$ intends to read some data objects whose values is not up-to-date as of $t(U)$ at $S_i$, i.e., a set of updates $(U', U'', ... , U_p)$ with their timestamps smaller than $t(U)$ have yet to modify some data objects in $RS(U)$ at site $S_i$. Update $U$ remains blocked until write action of every update in the set $(U', U'', ... , U_p)$ is executed at site $S_i$. For the sake of simplicity of argument, we assume that update arrival rate is the same at each site. Therefore, with probability $(N-1)/N$ an update in the previous set is remote to site $S_i$, i.e., it originated at a site other than $S_i$.

In the SDA, with probability $(N-1)/N$ values for an update in the set $(U', U'', ... , U_p)$ are computed by some site and with probability $1/N$ values are computed locally. Receiving values from remote sites may involve large delays because after a site has computed the values, they have to propagate through the communication network before they reach $S_i$. Thus, after condition (i) has been satisfied, condition (ii) may take long to hold and update $U$ may remain blocked for long time. This dependency among sites for computed values lengthens the blocking delays and response time of updates. This phenomenon, in general, occurs in all previous algorithms because a site waits for computed values from other sites.

In the FDA, however, after condition (i) holds for update $U$, it no longer waits for a message from remote sites for condition (ii) to hold. This is because site $S_i$ has received all updates up to timestamp $t(U)$ from remote sites, and can compute values of the data objects in $WS(U)$ locally. Since
such computation only requires access to CPU and secondary storage (disk) at $S_i$, when condition (i) holds, amount of time elapsed before condition (ii) holds depends upon the speeds of CPU and disk rather than on the speed of communication network.

Therefore, in semi-distributed algorithms blocking delays due to conflicts depend upon network propagation delay, whereas in fully-distributed algorithms blocking delays due to conflicts depend upon the speeds of CPU and I/O device. If CPU and disk are faster than the communication network, the FDA will have better response time and throughput characteristics. In section 6, we discuss results of a performance study which confirms the effect of decoupling on the response time of updates.

4. Informal Description of FDA

When a site $S_i$ receives an update $U=(rs, ws, f)$ from a user, it takes the following sequence of actions: it updates its clock and assigns a timestamp, say $ts$, to $U$; it sends an UPDATE($rs$, $ws$, $f$, $ts$) message to all other sites; and it saves the update with its timestamp. A site $S_i$ responds to the UPDATE($rs$, $ws$, $f$, $ts$) message in the following manner: it updates its clock; it returns a REPLY($ts$) message to $S_j$ (note that $ts$ is the updated timestamp at $S_j$); and it saves $(rs$, $ws$, $f$, $ts$).

A site $S_k$, $k=1,2,...,N$, executes the read and the compute actions of $U$ after it has received a message with a timestamp larger than $ts$ from all other sites and write action of all updates which have their timestamps smaller than $ts$ and which intend to modify some data objects in $ws$ have been executed at $S_k$. A site writes the computed values for $U$ onto the data objects in $ws$ of its copy of the database when read action of all updates which have their timestamp smaller than $ts$ and intend to read some data objects in $ws$ and write action of all updates which have their timestamp smaller than $ts$ and intend to modify some data object in $ws$ have been executed at that site.

5. Formal Description of FDA

Variables at site $S_i$

The FDA requires data structures to hold information about unexecuted or partially executed updates. A site $S_i$ maintains the following variables:

$C_i : 0=\ldots$; /* clock at site $S_i$. Initially $C_i$ is 0 */

ReadRequests; set of 4-tuples of the form $(rs$, $ws$, $f$, $ts$);

/* Each 4-tuple contains the readset, writset, update-function, and timestamp of an update which is yet to be executed at site $S_i$. Initially ReadRequests is empty */

WriteRequests; set of 2-tuples of the form $((ftr)$, $ts$);

/* Each 2-tuple contains the write-set-value, and timestamp of an update whose write-action is yet to be executed at $S_i$. Initially WriteRequests is empty */

Types of Messages

The FDA exchanges the following messages to transfer update information and to perform synchronization with other sites.

UPDATE($rs$, $ws$, $f$, $ts$): Site $S_i$ sends this message to all other sites when it receives an update $U=(rs$, $ws$, $f$) from a user that is assigned a timestamp $ts$.

REPLY($ts$): A site $S_j$ sends this message in response to an UPDATE message.

Synchronization Rules

An entry $(rs$, $ws$, $f$, $ts$) in ReadRequests is executed only when the following two conditions hold:

[R1] $S_i$ has received a message with timestamp larger than $ts$ from all other sites.

[R2] There is no entry $(rs$, $ws$, $f$, $ts)$ in ReadRequests such that $ts < ts$ and $ws \cap rs \neq \phi$, and there is no entry $(ftr_1)$, $ts_1$ in WriteRequests such that $ts_1 < ts$ and $ftr_1 \cap ftr \neq \phi$.

An entry $(ftr)$, $ts$ in WriteRequests, is executed only when the following two conditions hold:

[W1] There is no entry $(rs$, $ws$, $f$, $ts)$ in ReadRequests, such that $ts < ts$ and $ws \cap ftr \neq \phi$, and there is no entry $(ftr_1)$, $ts_1$ in WriteRequests such that $ts_1 < ts$ and $ftr_1 \cap ftr \neq \phi$.

[W2] There is no entry $(rs$, $ws$, $f$, $ts)$ in ReadRequests, such that $ts < ts$ and $rs \cap ftr \neq \phi$.

5.1. FDA at Site $S_i$

When a site $S_i$ receives an update $U=(ws$, $rs$, $f)$ from a user, it increments $C_i$ by 1, places an entry $(ws$, $rs$, $f$, $C_i)$ in ReadRequests, and sends UPDATE($rs$, $ws$, $f$, $C_i$) message to all other sites. When a site $S_i$ receives the UPDATE($ws$, $rs$, $f$, $C_i$) message, it sets $C_i$ to max($C_i$, 1+C_i), places an entry $(rs$, $ws$, $f$, $C_i, i)$ in ReadRequests, sends a REPLY($C_j$, $j$) message to site $S_j$, and calls procedure EXEC_READ (defined below). When $S_i$ receives the REPLY($C_j$, $j$) message, it sets $C_i$ to max($C_i$, 1+C_i), and calls procedure EXEC_READ.

procedure EXEC_READ; /* it executes the read action of all updates satisfying R1 and R2 */

begin
for every $(rs$, $ws$, $f$, $ts)$ in ReadRequests, such that R1 and R2 hold for it do
begin
ReadRequests:= ReadRequests - $(rs$, $ws$, $f$, $ts)$
read values of data objects in $rs$ from $D_i$
compute $ftr$
WriteRequests:= WriteRequests + $(ftr)$,
end;
if an entry in ReadRequests is executed then EXEC_WRITE; /*this procedure is defined below */
end;

procedure EXEC_WRITE; /* it executes the write action of updates satisfying W1 and W2 */

begin
for every $(ftr)$, $ts$ in WriteRequests, such that W1 and W2 hold for it do
begin
WriteRequests:= WriteRequests - $(ftr)$,
assign values in $ftr$ to corresponding data objects in $D_i$
end;
if an entry in WriteRequests is executed then EXEC_READ;
end;

Since at a site $S_i$, several updates may be running concurrently and simultaneously accessing the shared data structures (e.g., ReadRequests and $C_i$), for correctness sake we
assume that concurrent access to the shared data structures is mutually excluded. Due to space constraints, we omit proof of correctness of the algorithm; interested readers should refer to [21].

6. Properties of the FDA

Higher Parallelism: The FDA exploits the data replication property in true sense to increase the scope of parallel execution of updates. This is because after an update has been transported to all sites, they can concurrently execute the update. This improves the performance because a site can obtain values for an update much quickly by executing that update by itself rather than waiting for those values to come from other sites through a (slow) communication network.

Higher Resiliency: To execute an update U in semi-distributed algorithms for concurrency control, a site S_i performs some communication with other sites for synchronization, computes some values, and sends these values to all other sites. There is some chance of failure when other sites participate in synchronization for update U and the instant when S_i sends out corresponding computed values to other sites. If site S_i fails before sending out computed values to other sites, then the execution of write action of U gets blocked at all other sites until they discover that S_i has failed and take some measures, or S_i recovers from failure and sends out the computed values to all other sites. Therefore, in semi-distributed algorithms, execution of an update has a window of vulnerability defined by the two instants discussed above.

In the FDA, a site does not rely on other sites for computed values, hence, window of vulnerability is absent in the FDA. When a site S_i receives an update U from a user, the broadcasted update to all other sites. Since a site completely executes every update, failure of S_i afterwards does not block the processing of U at other sites. Therefore, the FDA is more resilient to site failures.

Low Communication Overhead: Semi-distributed algorithms require exchange of the values of data objects to be modified by an update. Normally, a data object is a relation, network, tree or a partition of any of these and depending upon the granularity, it may be large. Therefore, in semi-distributed algorithms sites exchange huge volume of information. Moreover, some coding or translation may be necessary to transfer values of data objects in a message. The FDA is free from these problems because it does not require transfer of values of data objects among sites. Instead, it requires transfer of updates among sites. Since for an update, its description is usually much smaller than the size of data objects which it modifies, volume of information exchanged is low in the FDA.

Moreover, from the point of view of the security and privacy, it is safer to exchange update description rather than the value of data objects. For example, suppose an organization maintains information about its employees in a replicated database and the organization wants to give a raise of 15 percent to some of its employees. For the privacy of the employees, it is better to exchange an update indicating 15 percent raise rather than the modified salaries of the employees.

7. Site and Media Failures

So far in this paper, we have assumed that a site does not fail and communication medium is reliable and order preserving. Although fully-distributed nature of the FDA makes it robust against site failures and communication link failure, it is not completely immune to these failures. These failures may result in loss of a message or partitioning of the network. In this section, we suggest measures to cope with these failures. Due to space limitations, we keep the discussion brief, and a detailed treatment can be found in [21].

Loss of Messages: Loss of messages can be handled by using a timeout mechanism. When a site S_i sends an UPDATE message to S_j, it initiates a timer. If the timer expires before its REPLY message is received from S_j, S_i sends another UPDATE message to S_j. S_j responds to all subsequent duplicate UPDATE messages by sending REPLY messages to S_i.

Network Partitioning: A network is partitioned if the system gets divided into groups of sites such that any two sites in different groups can no longer communicate with each other. We assume that partitioning is clean; that is, either all the messages from the sites in one partition to the sites in other partitions reach their destinations or none. Also, sites learn about the partitioning and sites in its partition quickly.

In face of a network partitioning, the FDA can avoid blocking of update execution in the following way: A site S_i determines all sites which have separated from it due to network partitioning and avoids blocking of the execution of pending updates by pretending as if it has received a REPLY message from all these sites. For execution of subsequent updates (until the network recovers from partitioning), S_i does not send an UPDATE message to all the separated sites and excludes all such sites from condition R1. Thus, each partition functions like an independent database system.

When the system recovers from a partitioning, database in each partition can be consistently integrated by using the techniques described in [4, 7]. A common value for sets ReadRequests and WriteRequests is derived for all merged partitions, and at all the sites being merged, these sets are initialized to their respective common values.

Site Failures: A site can handle failure of other sites by not sending an UPDATE message to failed sites and excluding failed sites from condition R1. Since a failed site does not process any update, recovery of a failed site is straightforward. A site keeps a journal (also called log [115] or audit trail [251]) which is a sequence of actions performed on the database. When a site recovers from failure, its database copy is reinstated by executing the journal of an operational site. Sites ReadRequests and WriteRequests at a recovering site are initialized by the corresponding sets of an operational site.

8. Performance Study

In this section, we study and compare the performance of the FDA and SDA using an event-driven simulator. We consider the following performance measures in the study: update response time, system throughput, I/O device utilization, and number of messages exchanged.

8.1. Performance Model

The performance model considered in this paper is similar to the one proposed in [13]. We model arrival of updates at a site by a Poisson process with parameter λ. We assume that size of readset and writerset of updates are identical and independent random variables with Geometric distribution with mean n. Access to data objects is uniformly distributed across M data objects of the database. To reflect the correlation between data objects in readset and writerset of an update, we assume that with probability 0.5 a
data object in readset of an update also belongs to its writset. Processing and storage facilities at a site are modeled by the queueing network model depicted in figure 1. We assume that CPU and disk service time are exponentially distributed, and they serve requests in FCFS discipline. We model communication delay by an infinite server. Service time of the communication medium (i.e., time taken by a message to travel from one site to another site) is constant.

Parameters of the Model

The performance model has the following parameters:
(1) AA: Aggregate Arrival Rate (=N*h) is the rate of update arrivals in the system.
(2) CF: Conflict Factor (=n/m) characterizes the probability of conflict.
(3) TM: Message Propagation Time.
(4) TD: Disk Access Time.
(5) TC: CPU Update Compute Time.

In[21], it is shown that number of sites N and update arrival rate at each site \( \lambda \) can be merged to get aggregate arrival rate AA (=N*\( \lambda \)) which gives the net rate of update arrival in the system, and average size of readset/writset n and number of data objects M can be merged together to give conflict factor CF (=n/M) which is square root of the probability of r-w, w-r, or w-w conflicts between updates. We assume that the database system is homogeneous, i.e., all sites have identical values for all the parameters.

8.2. Simulation Results

The results of simulation study are displayed (in figures 2 through 9) by plotting the collected values of performance measures against the parameter that is varied. We studied the effect of change of AA (figures 2 and 3), CF (figures 4 and 5), TM (figures 6 and 7), and TD (figures 8 and 9) on the performance of the algorithms.

Response Time: Figures 2, 4, 6, and 8 show curves for average response time of the algorithms as a function of AA, CF, TM, and TD, respectively. We observe that the FDA has better response time characteristics than the SDA (except in figure 8). The FDA has improved performance because due to decoupling effect, blocking delays due to conflicts in this algorithm depend upon the speeds of CPU and disk rather than the speed of the communication medium (note that, in general, CPU and disk are faster than communication medium in these simulations). Figures 4 and 6 reveal that response time of the FDA is not affected by a change in CF and it increases linearly with respect to TM, respectively. The FDA exhibits this behavior because its performance is limited by the throughput of the disk, not by blocking delays (see section on "I/O Utilization"). Figure 8 reveals that when we consider the response time as a function of disk access time, the FDA performs worse than the SDA. This happens because the FDA has higher disk requirements (as each site also executes read action of every update).

System Throughput: Curves in figure 2 also reflect system throughput for these algorithms. System throughput is the same as aggregate arrival rate (AA) until the saturation point of a curve (because until the saturation point update arrival rate is the same as update departure rate). For a curve, its saturation point denotes the maximum throughput of the corresponding algorithm. For example, in figure 2 the maximum throughput of SDA and FDA are respectively, 7.3 and 19.6. The FDA has higher throughput because its performance is limited by the speeds of CPU and disk rather than the speed of communication medium.

I/O Utilization: Curves in figures 3, 5, 7, and 9 show I/O device utilization as a function of AA, CF, TM, and TD, respectively. Since the FDA has higher disk access requirements, curve FDA is higher and steeper. I/O utilization curves provide a plausible explanation why response time curves for these algorithms shoot up. A comparison of figures 2 and 3 reveals that response time curve FDA shoot up because the I/O device becomes bottleneck. Whereas, the response time curve SDA shot up because blocking delays due to conflicts become large (because when response time curve SDA shoots up, the I/O device is only 30 percent utilized). A comparison of figures 7 and 9 also reveals the same pattern.

Number of Messages: Number of messages exchanged per update execution is deterministic for these algorithms, therefore, it can be determined without running a simulation. The FDA requires (N-1) update messages and (N-1) reply messages resulting in 2*(N-1) messages per update execution. The SDA requires (N-1) request messages, (N-1) reply messages, and (N-1) update_commit messages resulting in 3*(N-1) messages per update execution. The FDA has low traffic of messages because it does not require a phase of message broadcast which SDA requires to transfer computed values to other sites.
9. Concluding Remarks

We have presented a fully-distributed algorithm to synchronize concurrent updates in a fully replicated database system. In the algorithm, a site does not wait for computed values of the data objects from other sites, which has an important consequence on update response time: a shift in the sensitivity of blocking delays from speed of communication medium to speeds of disk and CPU. Moreover, fully-distributed nature of the algorithm enhances site autonomy, improves the system reliability, results in low communication overhead, increases the scope for parallelism, and imparts higher security and privacy to a system.

We have used a performance model similar to one proposed in [15] to study the performance of the proposed algorithm and the algorithm in [20] using an event-driven simulator. The results of the performance study indicate that in most cases the proposed algorithm performs better (i.e., has smaller response time and higher throughput). An interesting result is that in the proposed algorithm, the performance is limited by the throughput of the I/O device, not by the blocking delays due to conflicts. Since throughput of the I/O device can be easily adjusted (by putting more disks or using disk of higher speed), the performance of the proposed algorithm (FDA) can be readily controlled. Whereas, in the SDA the performance is limited by blocking delays due to conflicts which are difficult to control.

The algorithm has processing overhead because each site completely processes an update. However, sacrifice in the power of CPU and disk may be worthwhile when so many other benefits accrue from it — specially an improvement in the performance and a control over the level of the performance (by adjusting disk speed). Furthermore, the object of the processing overhead may lose its vigor with the advent of main-memory databases [8], where access to data objects is relatively faster and read or write action of an update can be executed without accessing the disk. These features of main-memory databases make the proposed algorithm highly suitable in such environments.

References


Figure 6, Response Time vs. Message Propagation Time

Figure 7, I/O Utilization vs. Message Propagation Time

Figure 8, Response Time vs. Disk Access Time

Figure 9, I/O Utilization vs. Disk Access Time

AA = 3.0
CF = 0.474
TD = 0.025
TC = 0.0001

AA = 3.0
CF = 0.474
TD = 0.025
TC = 0.0001

AA = 3.0
CF = 0.632
TM = 0.5
TC = 0.0001

AA = 3.0
CF = 0.632
TM = 0.5
TC = 0.0001

360