Design of Replicated Real-time Database Simulator

Hala Abdel hameed
Faculty of Information Technology
– Misr University for Science and Technology – Al-Motamayez
District 6th of October City - Egypt

Hazem M. El-Bakry
Department of Information Systems,
Faculty of Computer Science & Information Systems,
Mansoura University, EGYPT
helbakry20@yahoo.com

Torky Sultan
Faculty of Computer Science and Information Systems - Helwan
University, Helwan - Egypt

Abstract - A real-time database is a database in which both the data and the operations upon the data may have timing constraints. This paper presents the design of a replicated real-time database simulator. The Simulator architecture is discussed with a high level overview of modules and their functionality. We discuss the individual modules in a detailed manner by describing the various parameters that control their behavior as well as their interfaces with other modules. This simulator can be used as a base to study many design issues that affect the performance of real-time database such as different types of concurrency protocols, distributed commit protocols, priority assignment policies and load balancing.

Keywords - Database modeling, Distributed simulation, Real-Time databases design.

1. Introduction

Many real-time applications are inherently distributed in nature, and need to share data that are distributed among different sites. For example, military tracking, medical monitoring, naval combat control systems and factory automation etc. Such applications introduce the need for distributed real-time database systems (DRTDBs). A DRTDBS is a collection of multiple, logically interrelated databases distributed over a computer network [1]. A real-time database has two distinguishing features: the nature of its data that has a temporal constrains, and marinating a real-time constraints on transactions. Transactions in a real-time database are classified into three types, hard, soft and firm. A general model of distributed real-time systems was presented by Kopetz & Verissimo (1994) [2]. This model based on the interaction between real-time entities which is an element of the environment whose state is relevant to the DRTS such as temperature and pressure. A DRTS observes or modifies the states of RT entities; for example, based on an observation of the fluid level in a tank, the system could modify the position of a valve that affects the fluid drain. A DRTS interacts with the environment via sensors (hardware that samples the state of RT entities, such as temperature and motion sensors).

Many issues affecting the design of A DRTDBS to maintain its requirements; Data Consistency and Scalability are the main issues that are considered in this paper. All of those critical systems need data to be obtained and updated in a timely fashion, but sometimes data that is required at a particular location is not available when it is needed and getting it from remote site may take too long before which the data may become invalid, this potentially leads to large number of transactions miss their deadline and violating the timing constraints of the requesting transaction. One of the solutions for the above-mentioned problem is replication of data in real-time databases. By replicating temporal data items, instead of asking for remote data access requests, transactions that need to read remote data can now access the locally available copies which help transactions meet their time and data freshness requirements. Replication in DRTDBs also, is used to remove unpredictability of network delays or network partitioning, that the database is fully replicated to all nodes. It also improves fault tolerance for the main-memory resident data. In order to suite the different needs of the distributed real-time systems such as different data workloads and database specifications, multiple ways to handle the replication control and different replication schemes are proposed. However, such a database has another scalability problem since an update to an object of a fully replicated database needs to be sent to all other nodes.

2. Architecture

Simulation is an experimental method, instead of experimenting with the real system the experiments are performed with the simulation model (whose design is thus the key point of simulation studies). Based on time model on which events occur or their temporal behavior, simulation models can be classified into either continuous or discrete time models [3]. A model is a continuous time model when time flows smoothly and continuously. A model is a discrete time model if time flows in jumps of some specified time unit [4], [5].

The proposed simulator is based on Object Oriented Simulation (OOS) which is a special case of Object Oriented Programming (OOP)[6]. Some principles of OOP like existence of a varying number of instances of interfering objects have been in standard use in the simulation environment. Worth to say that Simula language (used to be called Simula 67) [7] is the first true object oriented language that Being more than 30 years old, it still has most (and all important) mechanisms and principles of OOP. Some things like classes, inheritance, virtual methods, etc. have been defined in Simula long time before they were rediscovered by the OOP boom in last year's.
Central to our simulation model for RTDBS is a single-site memory resident database system; the proposed framework that was illustrated in [8] is used. So the database contains a collection of objects belonging to different classes. Each object has a certain number of predefined methods. The object can be accessed only through these methods. The methods of the parent object may invoke the methods of the sub objects which can be either read or write.

The high level design of the simulator is illustrated in Figure 1; this design can be considered as an extension to [9] a simulator designed to model active real-time database. In the simulator each site is represented by an operation class, which is a general class for representing a node. For each node, an instance of the operation class is generated and there is a global Controller to manage communication/interaction between the operations instances. Each operation is consists of a set of components of its entire simulator, figure 2 shows this components, the figure also presents the communication-control structure of the simulator. The simulator for each operation consists of nine major components. Generator, Admission Controller, Replication Manager, Transaction Manager, Scheduler, Object Manager, Concurrency Controller, and Recourse Manager.

3. Simulation Time

The execution of the simulation events around the nodes could be represented by an acyclic directed graph (see Figure. 3), a sequential execution of processes at nodes is considered. Each vertex of the graph corresponds to an event occurring in the simulated node. Precedence constraints exist among the events, to model the chronological order of events. These precedence constraints are modeled by the arcs of the graph: an arc from vertex to vertex means that event cannot occur (or be executed) before event is processed. Two types of arcs are distinguished: intra-process arcs and inter-process arcs. Intra-process arcs are precedence constraints between events that occur within the same node (e.g., arc between vertex EA and EC in Figure. 3). The intra-process arc denotes an independent unit of sequential work inside a node. While inter-process arcs represent the precedence constraints between events that occur in different nodes (e.g., arc between vertex EB and EC). These inter-process arcs represent synchronized interactions between different intended nodes achieved by network communication.

For parallel execution, it was assumed that there are one processor as hardware for the simulation to represent both inter and intra- process dependant events. (i.e., one process to one exclusive processor), all inter and intra-process dependent events occur at the same exclusive processor. Although, there are one processor for the whole simulation, there are different time phase for each node where events are synchronized by a one global clock.

The global clock implements a virtual time system for a distributed system executing in coordination with an imaginary virtual clock that ticks virtual time. Table 1 shows the time span and the equivalence values of seconds that represent the tick. Virtual time itself is a global, one dimensional, temporal coordinate system imposed on a distributed computation; it is used to measure computational progress and to define synchronization. It may or may not have a connection with real time. We assume that virtual times are real values (with a positive infinite value +inf),
totally ordered as usual by the relation <. From a programmer’s semantic point of view, the global virtual clock always progresses forward (or at least never backward) at an unpredictable rate with respect to real time. But, from the implementer’s point of view, there are many local virtual clocks, loosely synchronized, and while all of the virtual clocks tend to go forward toward higher virtual times, they occasionally jump backward. We envision systems of many (maybe thousands) of small processes all executing concurrently on a computer with many processors.

Based on the above paradigm, the simulator being modeled is viewed as being composed of some number of physical processes (nodes) that interact at various points in simulated time. The simulation is constructed as a set of logical processes, one per physical process. All interactions between physical nodes are modeled by time stamped event processes, one per physical process. All interactions between events, and then only to the value in the timestamp processes (nodes) that interact at various points in simulated time. Ideally, the execution of a node is simply by receiving messages and executes events in increasing virtual time order. This ideal execution proceeds as long as no message ever arrives with a virtual receive time in the “past.”

Table 1: time span for ticks from MSDN of Microsoft

<table>
<thead>
<tr>
<th>TimeSpan(1)</th>
<th>00:00:00.0000001</th>
</tr>
</thead>
<tbody>
<tr>
<td>Days</td>
<td>TotalDays</td>
</tr>
<tr>
<td>Hours</td>
<td>TotalHours</td>
</tr>
<tr>
<td>Minutes</td>
<td>TotalMinutes</td>
</tr>
<tr>
<td>Seconds</td>
<td>TotalSeconds</td>
</tr>
<tr>
<td>Milliseconds</td>
<td>TotalMilliseconds</td>
</tr>
<tr>
<td>Ticks</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: the virtual time management by the global time controller

Time at each node could be changed only by two actions; global time control or the termination of an operation execution. While the global time itself is changing each tick. Figure 4 illustrates how the global time controller controls the time at each node. Each tick the controller scans all nodes one by one to check if there is any process arrived at node. As mentioned above each node could have a different time value according to its process time. When the controller checks the status of a node there are three different actions that could be taken according to the node status:

- The node time is less than the global time and there are no process running at that node. So, the time of the node is jumped to equal the global time.
- The node time is less than the global time and there is an operation arrived at the node, in this case the process is started and set the node time to the end time of the process.
- The node time is greater than the global time as a result for process execution, in this case the node just sleep and takes no action.

As illustrated in figure 4, At time 1 (tick1) the controller scans node 1 which satisfy the first case, then the node time is jumped to equal the global time. For node 2 there is an operation arrived at tick1 leading to setting the node time to the end time of the operation. At tick 3 the scanning of node 3 resulting in taking no action and move to the nest node, because the virtual time of node 3 is equal to 5 which is greater than the global time.
4. Distributed Classes

In a simulation it is often necessary to specify the distribution functions of various events (e.g., the rate of arrivals of transaction, or the time between failures for a node). As such we have created a set of classes which provide access to various useful distribution functions. These include: RandomStream, UniformStream, ExponentialStream, and HyperExponentialStream. By creating instances of these classes, the simulation processes can gain access to the appropriate distribution function. To illustrate how the distribution functions are derived and to show how further functions could be built we model a replicated database by an open queuing network in which each of the n identical sites is represented by an M/Ht/1 system. Transaction arrivals to the distributed system from outside are modeled by n identical Poisson processes with parameter λ (one arrival stream per site). Poisson streams are known to be a good approximation for jobs submitted independently by a large number of users, and are used in nearly all performance models of distributed database. As declared earlier, there are t different types of transaction streams (sensor – user- propagation) generated by the Generator. The t different types of transactions are numbered 1, 2, . . . , t . Hence, the service time for the combined arrival process of all t transaction types follows a t - phase hyper-exponential distribution.

5. Simulator Components

A. Generator

The generator component simulates the incoming transactions into the system. It actually simulates the application or the environment in which the Real-time Database is used. The transaction generator can be changed depending on the application we are modeling. We can study the transaction characteristics of the application of our choice and design the generator to model it as closely as possible. It generates transactions with timing constraints with a particular arrival distribution. It can generate both periodic and aperiodic transaction streams. We refer to the transactions generated in the source as external transactions. In the simulator, the generator is represented as a class that generate different stream of transactions. There are three instances of generator class running in the simulator:

1. Sensor Transaction generator, it is an instance of the generator class to generate a periodic sensor transactions for each real-time data object according to its BUF.

2. User Transactions generator, an instance of the generator class to generate aperiodic transactions to simulate both external environment events signaled from the applications and the local transactions generated locally at each site. The arrival of hose aperiodic transactions is based on a Poisson arrival pattern.

3. On-line Transaction Generator; this class instance generate transactions when it is needed (for example the propagation transactions, when an object is updated it calls the generate method in this instance to generate propagation transactions for the participated nodes).

The transaction generator class can generate either hard real-time or soft real-time transactions depending on the timing constrains properties (e.g. Relation between deadline violation and the value obtained from transaction execution). The transactions generated by the different class of generator have the following parameters:

- MAXTRANSCCLASS: This specifies the maximum number of transaction classes in the system.
- NoInstance: the actual number of generator class used to generate transactions with different streams.
- MAXGENTRANS: The maximum number of generated transactions during the whole simulation.
- SensorTransRatio: The ratio of number of sensor transactions to the total number of transactions.
- UserTransRatio: The ratio of number of user transactions to the total number of transactions.
- UserWriteProb: The probability that the operation in the user transaction is write.
- UserRemotePercent: The percentage of the operations that are executed remotely at other site.
- UserLocalPercent: The percentage of the user transactions that are executed locally at its initiated site.
- SensorRemotePercent: The percentage of the sensor transactions that are executed remotely at other site. It was assumed that all the sensor data items are updated locally.
- SensorLocalPercent: The percentage of the sensor transactions that are executed locally. Sensor data object can be updated only by its primary site.
- StreamType: Per class Instance, This parameter specifies whether this stream of transactions generated by this instance is aperiodic stream or a periodic stream or online transactions stream.
- InterArrival: This specifies the inter-arrival time between two transactions of aperiodic transactions.
- IsHardFirmSoft: This parameter specifies whether this stream generates hard real-time, firm real-time or soft real-time transactions. According to the value obtained from completing its execution by its deadline time.
- ClassPriority: This per class parameter gives the priority of that class of transactions.
- ArrivalDistribution: This parameter gives what arrival distribution the generator instance follows in generating transactions.
- Deadline: a value used to represent the deadline value.
- MinSlack: A per class parameter that gives the minimum slack values of that transaction class.
- MaxSlack: A per class parameter that gives the maximum slack values of that transaction class.
- MaxTransClassPercentage: A per class instance parameter that gives the maximum number of transaction generated for that class.

There are some parameters related specifically to each transaction as illustrated in [8]:
- TransId: a unique identifier for each generated transaction.
- RSet: a set of objects that are read by the transaction.
- WSet: a set of objects that will be updated by the transaction.
- ExecutionTime: a value assigned randomly to the transaction according to its type class:
  - For sensor transactions (it is uniformly distributed).
  - For user transactions (hyper-exponential distribution is used).
- LocalSite: the local site where the transaction is originated.
- RemoteSite: the remote site to where the transaction will be sent. Note that to generate transaction for a specific site N, the LocalSite equal the current site while RemoteSite equal (1-N). If LocalSite equal RemoteSite, then the transaction is a local transaction, otherwise it is a remote transaction.
- InitialTime: a value assigned to the transaction to represent its initiation time at which it was generated from the generator.
- ArrivalTime: a value assigned to the transaction according to its class type as previously illustrated.
- StartTime: time at which the transaction start execution.

B. Object Manager

Object Manager is the module that models the data in the simulator. The object manager use the framework presented in chapter 4 to represent data objects. As it was illustrated, the data is modeled as having certain number of object classes and each object class have a certain number of instances. Each class has certain number of methods defined which are used to access the object. Each object instance in the database is mapped to a certain number of pages in the secondary storage. The number of pages that each object instance in a class needs is given as a parameter. All object instances of a particular class are assumed to be stored in a single file. So, now we have a collection of files of known size. In the simulator we use class for temporal data objects, both sensor and derived data objects.

A data object in a distributed database system is considered the smallest accessible unit of data, for example a data object O could be a table, relation, an attribute or even a field. The level of granularity is maintained by the database designer and this level is what is subjected to different data manipulation operations, such as replication.

- MAXOBJCLASSES: This is the maximum number of object classes that is in the database.
- NumberOfClasses: This parameter defines the actual number of object classes to which the objects in the database belong.
- NumInstances: This is a per class parameter that gives the number of object instances in each class in the database. The sum of all class instances gives the total size of the Database.
- FreshReq: Freshness requirements, this per object parameter controls the minimum required degree of consistency.

For each data object, there are the following sets of attributes:
- Id: is the unique identifier for the object on his primary site.
- Type: whether it is local or shard data object.
- PsiteId: the object's primary site id where the object was originates, this attribute give an indication of whether the object is a primary data object or it is a replica e.g., if PsiteId = local site, this object is a primary object originated at this site, otherwise it is a replica for a remote data object.
- Value: is used to store the final attribute value captured by the related last update method.
- TS: is used to store the latest time at which the attribute's value was updated.
- VI: denotes object's absolute validity interval i.e., the length of the time interval following timestamp during which the object is considered to have absolute validity.
- BUF: is the Basic Update Frequency, for each temporal data object it is updated periodically at a given update frequency received from its primary.
- FR: if that object has a predefined freshness requirements to maintain the consistency level between different replicas scattered over all sites for the same data object.
- VN: is the version number that reflects the last update number of that object.
- IsLock: a Boolean variable to specify whither the data object is locked or not.

The Object Manager, OM is also responsible for accepting the access requests from both the Transaction Manager and Concurrency Controller and satisfying the access requests. The access requests just specify the object that is to be accessed and the method to be executed on the object. The OM checks if the object is accessible by checking the lock variable, and if it is accessible, executes the method and sends back a success or failure message to the Transaction Manager (TM), If the object is not accessible it sends back
C. Transaction Manager

Transaction manager receives the transactions from the source generator. It executes the submitted transactions by calling object manager to execute the methods of various objects accessed. It handles the various transaction events like begin, commit and abort.

Generally, a transaction is an element of the system that brings information from applications to Object Manager and returns information from Object Manager to applications. A transaction is associated with requirements policies for executions and importance in addition to usual method invocations with required parameters. A transaction is modeled by < ID, StatCode, P, RQ, SES > in which ID is a unique identifier of the transaction, StatCode is the statement or execution code, P is the importance or assigned priority, RQ is a set of requirements, and SES are the static execution semantics. The set of requirements RQ consists of requirements for each property real-time, QoS, and optional PreCond and PostCond. Pre Cond represents preconditions that must be met before the transaction can be executed and PostCond represents post conditions that must be satisfied upon completion of the transaction. For example, it may be appropriate for a transaction to execute only if some specified event has occurred, such as dependability of the related transactions. The real-time requirements of a transaction are specified in terms of a deadline, start time its period if it requires periodic execution the transaction type (hard firm, or soft). The SES consists of the following types of information about transactions resource needs for memory, bandwidth, IO execution time, and the read and write sets. SES represents the facts about the needs of a transaction while RQ represents the requirements that a user or the system imposes on an instance of the transaction. The SES for each transaction is determined by pre analysis and is stored as part of the database.

As distribution requirements, the transaction manager is responsible for transactions commitment by implementing a Strict Two Phase Commit protocol (S2PC). In a common model of a distributed system, for each transaction there is one process, called as coordinator, which is executed at the site where the transaction is submitted, and a set of sub processes, called Cohorts, which executes on behalf of the transaction at these various sites that are accessed by the transaction. In the simulator, it was assumed that only one cohort could be initiated for each transaction. As future work it will extended to enable more than one cohort to be initiated in parallel for a specific transaction.

In the 2PC protocol, there are two phase for transaction commitment, the voting and decision phase. When a transaction is initiated, the data items that will be used are chosen by the Generator, and the coordinator is then loaded at its originating site and initiates the first phase of the protocol by sending PREPARE (to commit) messages in parallel to all the cohorts. Each cohort that is ready to commit, first force writes a prepared log record to its local stable storage and then sends a YES vote to the master. At this stage, the cohort has entered a prepared state wherein it cannot unilaterally commit or abort the transaction but has to wait for final decision from the coordinator. On other hand, each cohort that decides to abort force-writes an abort log record and sends a NO vote to the master. Since a NO vote acts like a veto, cohort is permitted unilaterally abort the transaction without waiting for a response from the coordinator. After the master receives the votes from all the cohorts, it initiates the second phase of the protocol. If all the votes are YES, it moves to a committing state by force-writing a commit log record and sending COMMIT messages to all the cohorts. Each cohort after receiving a COMMIT message moves to the committing state, force-writes a commit log record, and sends an acknowledgement (ACK) message to the master. If the master receives even one NO vote, it moves to the aborting state by force writing an abort log record and sends ABORT messages to those cohorts that are in the prepared state. These cohorts, after receiving the ABORT message, move to aborting state, force-write an abort log record and send an ACK message to the master. Finally, the master, after receiving acknowledgement from all the prepared cohorts, writes an end log record and then forgets about the transaction.

- MaxActiveTrans: The maximum number of transactions that can be active in the transaction manager.
- ActiveTransId: The number of the transaction that is running in the Transaction Manager.
- CoordinatorLoadCPU: The cpu time needed to load the coordinator in its originated.
- CohortLoadCPU: the cpu time needed to load the cohort.
- MaxCohort: The maximum number of cohort that can be initiated by the transaction manager.
- DelayRestart: This parameter specifies if there will be delay in the restarting of transactions, this is neglected in the simulator.
- logFlushTime: Time to flush the log.
- logTimeOut: Time in between log flushes

D. Scheduler

A scheduler determines the transaction to be executed at a particular moment. There are multiple of scheduling algorithms such as preemptive and priority driven ones. In
these algorithms, whenever there is a request for a task that is of higher priority than the one currently being executed, the running task is immediately interrupted and the newly requested task is started. Thus the specification of such algorithms amounts to the specification of the method of assigning priorities to tasks. A scheduling algorithm is said to be static if priorities are assigned to tasks once and for all. A static scheduling algorithm is also called a fixed priority scheduling algorithm. A scheduling algorithm is said to be dynamic if priorities of tasks might change from request to request.

The main task of a scheduler is to minimize the number of late transactions. In the simulator Earliest Deadline First (EDF) policy is used. A common observations and studies have showing that assigning priorities to transactions according to an Earliest Deadline policy minimizes the number of late transactions in systems operating under low or moderate levels of resource and data contention. This is due to the EDF policy is giving the highest priority to transactions that have the least remaining time in which to complete. These studies have also observed however, that the performance of Earliest Deadline steeply degrades in an overloaded system. This is because, under heavy loading, transactions gain high priority only when they are close to their deadlines. Using EDF and considering that ATr, DLr, and Pr are used to denote the arrival time, deadline, and priority of transaction r respectively. The priority assignments of all the mappings are such that smaller Pr values reflect higher system priority. The Earliest Deadline mapping assigns higher priority to transactions with earlier deadlines, and the transaction priority assignment is Pr = DL. The parameters of the scheduler are:
- RejectionPolicy: This specifies the rejection policy in the transaction manager (e.g. all eligible, not tardy, feasible deadlines).

E. Admission controller

In the proposed simulator, both periodic and a periodic transactions are managed. It is known that the transaction arrival rate is useful to model a RTDB in which transaction execution times are usually not known a priori. We have performed a simulation in which the utilization is measured for the increasing transaction arrival rate. In Figure 1, the utilization increases and saturates at 100% as the average arrival rate, indicated by tps (transactions per second), increases. Notably, the success ratio sharply decreases as the arrival rate further increases after the utilization is saturated due to the severe data/resource contention. Thus, it is necessary to avoid the saturation by controlling the arrival rate. Data contention can also increase the load due to many transaction aborts and restarts. In the next section, it will be illustrated how this module could be used to increase the propagation transaction sent to other nodes. There are two types of transactions: sensor transactions periodically update the sensor data in the RTDB to reflect the current real-world status. User transactions arrive aperiodically and they can read sensor data and read/write non-sensor data as discussed before. User transactions also include both local propagation transactions sent to other sites and remote propagation transaction received from other site to achieve the desired level of consistency between nodes.

As described in Figure 5, the QoS Manager and Admission Controller adapt the workload according to the control signal. Note that QoS degradation is applied before admission control to improve the success ratio. When the system is overloaded due to data/resource contention, in this case, a subset of transactions in the system can be degraded. The QoS degradation is conceptually equivalent to decreasing the arrival rate. Data/resource contention can be decreased in this way, because a degraded transaction will access a reduced number of data doing less arithmetic/logical computation.

F. Resource Manager

The resource manager simulates the CPUs and Disks. While the simulated database is a memory resident, the resource manager simulates only the CPU. The Transaction manager requests the resource manager for the necessary CPU time. It always assumed that pages are resident in the memory buffer and there is no need to bring it into memory. The measure of overall performance is average response time for completed transactions.

G. Recovery Manager

The recovery manager module is responsible mainly for maintaining the ACID property for the transactions, as it is responsible for rollbacks and recovery of transactions. The transaction manager informs the recovery manager of the transaction events. Transactions that are interrupted due to a failure of a replicated object are restarted from the beginning. The nodes fail according to their Mean Time To Fail (MTTF) and recover according to their Mean Time To
Recover (MTTR). MTTF and MTTR are the reliability values which are assigned when the nodes are “created” and never change during a single simulation run. In our experiments, for each node a backup node is specified and we assumed that both a specific node and its backup cannot be failed at the same time. Also, nodes failures are assumed to be independent.

The Recovery Manager module has the following parameters:
- MTTF: this parameter is Per node, and it indicates the Mean time to fail for a node.
- MTTR: Also, per node, to indicate the Mean Time to recover.
- Restart time: This parameter specifies if there will be delay in the restarting of transactions.

H. Replication Manager

The replication Manager is responsible for implementing the proposed replica control algorithm for calculating both the number of replicas (RD) created for each object and nodes to which it will be propagated (RAS), as part of DoMORE module. Figure 6 depicts the inputs and outputs of a Replica Control Algorithm implemented in the Replication Manager. Entire site work load is one of the factors affecting the output values of the Replication Manager; the effect of the work load is described in [8]. Another factor such as freshness requirements and user defined degree are also could be considered in calculating the Replication Degree. The Replication Manager parameters are:
- WorkLoad: the entire work load for node N.

Data in a node directory is updated incrementally by add() and remove() operations, received from database nodes that have changed their object allocations. This data is need to be replicated to all the leaders (this replication effort is treated same as any replication transaction) but will use a strict replication degree which includes all Leader Nodes. Each node's directory is periodically updated by the Leaders Directory using a pre-defined update plan. This plan includes both update for Leaders Directories and the ordinary node's directories, while requests for directory records are treated as ordinary data requests. An alternative to that Director structure would be to search for object information at all the database nodes in the system, or depend on the probability of finding the right property for an object or a node. However, with a directory service at a few nodes, the lookup cost is much lower for replica establishment communicates with one node, than a search approach needs to communicate with all other database nodes which is considered a worst case scenario.

J. Network Manager

The network manager encapsulates the model of the communication network. In the simulator it just act as a switch for routing messages from site to site. The main task for the network Manager is to control the exchange of messages between different nodes. When a site has to send a message to some other site, it hands over the message to the communication medium, which delivers it to the destination site in finite time. We assume that, for any pair of sites Si and Sj, the communication medium always delivers the messages to Sj in the same order in which they were handed to the medium by Si. In the simulator, we assumed that for each remote transaction (for each cohort) there are 4 messages; one as a request which in turn wait for a reply with another message, and two message for committing the transaction. There three types of messages:
- A request message that carry a remote transaction to be executed at other node.
- A response message, for a specific transaction at two different nodes.
- An acknowledgment messages such as messages that are exchanged during the commit protocol.
The sender site save a unique identifier for each sent message called (TrackingId) in the tracking-list, when it receives a new message, it checks the received message with the TrackingIds saved in its tracking-list. When one of the coming IDs matches one of the saved IDs, the message is processed and is deleted from the TrackingList. The defined parameters for the Network Manager:

- MaxMessage: maximum number of messages in the tracking list (in the current work there are no restrictions in the messages number).

6. Conclusion

A general simulator to model a replicated real-time database has been designed for small and medium scale distributed real-time database. This simulator can be used as a base to study many design issues that affect the performance of real-time database such as different types of concurrency protocols, distributed commit protocols, priority assignment policies and load balancing. It also can be used as a base environment to be used in any proposed models to test all the previous issues.

REFERENCES