A Replication Strategy for Distributed Real-Time Object-Oriented Databases[*]

Praveen Peddi, Lisa Cingiser DiPippo
The University of Rhode Island
Kingston, RI USA 02881
dipippo@cs.uri.edu
ppeddi@hotmail.com

Abstract

This paper describes a replication algorithm for distributed real-time object-oriented databases in a static environment. All data requirements are specified a priori, and the algorithm creates replication transactions that copy remote data to a local site in order to guarantee that every data request reads temporally valid data. The algorithm conditions are proven to be necessary and sufficient for providing this guarantee. Test results indicate that under most conditions, this replication strategy is better than total replication, which is a typical strategy used in distributed databases.

1. Introduction

Many time-critical applications require the sharing of data that may be distributed among multiple sites. For example, a critical planning collaboration among ships in a fleet may need to share sonar and radar data that is collected by the various vessels. In such applications, it is imperative that the data be available to the requesting transactions at the time it is needed. In a typical distributed database, the transaction is required to access the remote data directly, at the risk of missing its deadline. Another problem can occur in such a scenario when the requesting transaction accesses the data, but it is not temporally valid. That is, its value is "out-of-date" because the transaction did not read from the most recent update.

In this paper, we present a replication algorithm, called the *Just-In-Time Real-Time Replication* (JITRTR) algorithm, that creates replication transactions based on client's data requirements in a distributed real-time object-oriented database (DRTOODB). These replication transactions copy data objects to the site on which they are needed "just in time" for the read to occur. The algorithm carefully computes the parameters of the replication transactions so that we can guarantee that any requests that read data, in fact, read temporally valid data.

The JITRTR algorithm is designed to work in a static environment in which all object locations, and client data requirements are known a priori. Given these static system specifications, the algorithm creates a set of transactions that access the data objects. These transactions are then mapped to a well-known schedulability model so that the system can be analyzed to determine if all specified deadlines can be met. If so, then the system can be executed and all requests will read temporally valid data. If the system is not schedulable, then the system specification must be reconsidered.

The rest of this paper is organized as follows. Section 2 presents some background on the real-time schedulability model on which we have based our algorithm. It also provides a survey of related work to indicate the novelty of our algorithm. Section 3 presents the JITRTR algorithm, starting by describing the system model, and then describing how the algorithm maps the system specifications to transactions that access the data objects. Section 4 provides an analysis of the algorithm. It presents three theorems that demonstrate the goodness of the algorithm. Section 5 provides a brief description of a set of performance tests that we carried out to illustrate how our algorithm compared to other related techniques for data replication. Finally, Section 6 summarizes the results of this work, briefly describes an extension of the work that provides finer granularity of

[*] This work is partially supported by the U.S. Office of Naval Research grant N00014-00-1-0060.

data access, and gives some insight into future work that may result from this project.

2. Background and Related Work

This section presents foundational information for the underlying concepts of the JITRTR algorithm including concepts of DRTOODBs, scheduling protocols and related replication control algorithms.

2.1. Distributed Real-Time Object-Oriented Databases

A real-time database is one in which both the transactions and the data have timing constraints [1]. For instance, deadlines can be imposed upon the completion time of transactions and on the validity of a data item. The *temporal validity* of a data item is the amount of time that its value is considered valid. In a real-time object-oriented database (RTOODB years [2,3, 4, 5]), temporal validity can be expressed as a part of the object.

In a DRTOODB, real-time access to a remote object can be obtained by having transactions directly access the object, keeping copies of all objects on all sites, or providing a mechanism for making copies such that they are kept on sites where they are needed. Data replication incurs complexity because not only must the original data object be kept temporally consistent, but the copies must be as well.

2.2. Scheduling Protocols

In a DRTOODB, transactions must be scheduled so that they do not violate timing constraints. Additionally, access to shared data must be controlled in such a way as to prevent data inconsistency and to uphold the priority assignment made by the scheduling algorithm. The JITRTR algorithm creates transactions that execute in a system running deadline monotonic scheduling (DM) [6], and distributed priority ceiling protocol (DPCP) for resource access control [7].

Deadline Monotonic Scheduling. DM scheduling [6] assigns priorities to periodic tasks such that the task with the shortest relative deadline is assigned the highest priority. To determine the schedulability of a set of tasks scheduled using DM priority assignment, we compute the worst case response time for a task by considering the amount of time the task will be preempted by higher priority tasks, plus the execution time of the task itself, plus any blocking by lower priority tasks that may occur due to sharing of resources (priority inversion). If all tasks can meet their deadlines, then the system is schedulable.

Distributed Priority Ceiling Protocol. DPCP [7] is a resource access control algorithm that does not bounds priority inversion and prevents deadlock. The protocol assumes that tasks and resources have been assigned and statically bound to processors, and that priorities of all tasks are assigned in advance making it suitable for static hard real-time systems.

The DPCP scheduling model is made up of resources, and tasks that access the resources. A resource that resides on the local processor of a task is a *local resource.* A *global resource* is a resource that is accessed by at least one task that is on a different processor. A task executes in a *global critical section* if it is accessing a global resource and executes in *local critical section* if it is accessing a local resource.

The various possible uses of resources within the DPCP model define a set of blocking times that must be taken into account when analyzing the schedulability of a system. These blocking times (described fully in [7]) are added to the analysis described above for DM to provide the proper analysis.

2.3. Replication Control Algorithms

Many real-time replication control algorithms have been proposed based on concurrency control mechanisms like *majority consensus approach* [8] and *distributed two-phase locking* [9], *distributed two-phase locking, distributed optimistic concurrency control (OCC)* [10], *distributed optimistic two-phase locking (O2PL)* [11]. MIRROR (*Managing Isolation in Replicated Real-Time Object Repositories)* [12] is a concurrency control algorithm designed for real-time systems with replicated data. It augments O2PL to provide state-conscious priority blocking. All of the above mentioned concurrency control algorithms suffer from the possibility of deadlock and unbounded blocking. Thus, they are not suitable for a static real-time database system.

3. JITRT Replication Algorithm

This section describes the *Just In Time Real-Time Replication (JITRTR)* algorithm. It creates real time replication transactions in a DRTOODB based on the data requirements in a static system. Figure 1 depicts the flow of the algorithm, which has two parts. In the first part, the *Replication Manager* (RM) takes the parameters from the system specifications and creates local and replication transactions. In the second part, the transactions are mapped to an analyzable model, based on the DPCP model [7].

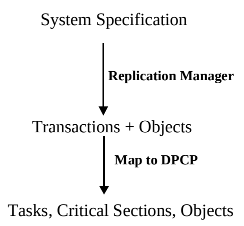


Figure 1 - JITRT Algorithm Methodology

We begin this section by defining assumptions we have made about the system, followed a description of the system model. We then explain how the RM does the core work of the algorithm, i.e. taking the parameters from the system specifications and creating the local and replication transactions. Finally, we describe how the transactions are mapped to the DPCP model.

3.1. Assumptions

The following is a list of assumptions we have made regarding the system in which JITRTR algorithm works.

1) The system is static. That is, all distributed sites and every object on each site are known a priori. All read/write requests from clients are known a priori.
2) For each object there is one update transaction that we call the "sensor update transaction". There can be more than one transaction that updates the object, but only one is called the sensor update transaction.
3) Each object has a local site, where it originates. Any other sites that require this object have a copy of it.
4) All the databases in the distributed system are homogeneous. All the sites in the system contain the same DBMS.
5) The period of the sensor update is always less than the temporal validity of the objects. That is, the object will be updated before it becomes temporally inconsistent.
6) Copies of objects are not accessible to transactions on other sites. That is, only the object on its origination site is accessible to be replicated.

3.2. System Model

The model on which the JITRTR algorithm is based is made up of *M* distributed sites, data *objects*, and periodic *requests* and *updates* that access the data objects.

Objects. Each object in the system is defined as follows:

Object = < OID, Value, Time, OV >

OID is a unique identifier of the object within the system. *Value* is the present value of the object. *Time* is the time at which the object was last updated. *OV* is the object validity, i.e. the time after which the value of the object is no longer valid.

Requests and Updates. Application requirements are specified as periodic *Requests* for data and *Updates* of data with following parameters:

Request = <OID,per,rel,dl,LSiteID>
Update = <OID per,rel,dl,LSiteID>

Requests are read-only data accesses, and updates are write-only. *OID* is the unique identifier of the requested object. *Per* is the frequency (period) at which the data is to be accessed. *Rel* is the release time at which the request/update should be started, *dl* is the relative deadline of the request/update within each period and *LSiteID* specifies the site at which the update/request was made.

3.3. Replication Manager

Given a system specified by the above model, the JITRTR algorithm creates replication transactions to ensure the availability of data. The algorithm produces a model with two types of transactions, local transactions and replication transactions. A transaction is a local transaction if all of its operations execute on the same site as the site on which the request was made, and it is a replication transaction if at least one of its operations executes on a remote site.

The following is the specification for the model of a transaction created by the Replication Manager (RM):

T_{type} *<opers(OID),per, rel, dl, exec, LSiteID>*

where *type* specifies the type of the transaction, local or replication. *opers* is a set of operations on object OID, such as read, and write. *Per, rel, dl* and *LSiteID* are as defined above. *Exec* specifies the worst-case execution time of the operations plus any other time incurred by the transaction.

The following subsections describe how the JITRTR algorithm maps systems specifications for requests and updates respectively, to local and replication transactions.

3.3.1. Requests. When mapping requests to transactions, there are two cases to be considered. The first case occurs when the site on which requested object originated is the same as the site at which request is made, and the second case is when the two sites are not the same.

Case 1: RSiteID = LSiteID:

This is the simpler of the two cases, because no copies of data need to be made. The request maps to a local transaction specified as follows:

T_{local} *(opers(OID), per, rel, dl, exec LSiteID)*

where *opers(OID)* is a *read(OID)*on local site of OID, and *exec* is the execution time of the read operation.

Case 2: RSiteID ≠ LSiteID

In this case, the request is for data that resides on a remote site. Thus, a copy of the data will be made on the local site (by a replication transaction) so that the data can be read within the specified deadline. The replication transaction must finish before the start of the local transaction so that the local transaction can read the copy.

Replication transaction: The JITRTR algorithm creates a single replication transaction for all requests made on a particular site for an object, OID. This allows for sharing of the local copy among all local transactions. The parameters for the replication transaction are:

T_{rep}*(opers(OID),per,rel,dl,execLSiteID)*

where *opers*(OID) are *read(OID)* on RSiteID and *write(OID)* on LsiteID. *Per* is equal to the period of sensor update of the object so that the transactions will read valid data (see Theorem 2). The release time *(rel)* is the start of the period, and *exec* is the total execution time of the replication transaction (exec time of read + exec time of write + network delay). The deadline (*dl)* is the crucial part of the transaction, because it must be computed to ensure that the local copy is available and valid when the data is needed.

Deadline Computation. Let *d* be the deadline that we are computing for a replication transaction on LSiteID created for a data request of object *OID*. Let *N* be the least common multiple of the periods of all requests for *OID* on LSiteID and the period of sensor update. Let *n* be the number of replication periods that should be considered for the analysis, where *n* is computed as

$$n = N/per$$

We call *N* the superperiod of the replication transaction because it represents a complete cycle of all requests for the data being replicated on LSiteID. We define OV_i to be the point in time in the i^{th} period of the replication transaction that the value of the object (from the most recent update) becomes temporally invalid. An invalid interval is an interval of time during which the object does not have a valid value associated with it, that is, the object is temporarily inconsistent (See Figure 2).

Initially we set the deadline of the replication transaction equal to its period. Then, for each of the *n* periods in the superperiod, there are 3 cases to consider in calculating the deadline.

1) If no requests are executing in the invalid interval, the deadline is unchanged because no requests will be reading invalid data.
2) If no request has started executing before the invalid interval but a new transaction enters at x_i, where $OV_i < x_i < P_i + d$, then the deadline is changed to $min(d, x_i - P_i)$.
3) If any request has started before or at OV_i and continues to execute in the invalid interval, then the deadline is changed to $OV_i - P_i$. This deadline assignment ensures that the replication transaction completes before the data becomes invalid, and thus the requests read valid data.

Note that if the deadline is changed to $OV_i - P_i$ at any point, the computation of deadline is complete as we have reached the minimum possible deadline. Otherwise we consider these three cases for each of the *n* replication transaction periods in the superperiod.

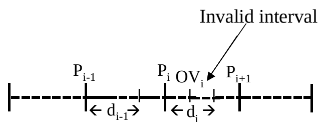


Figure 2 – Invalid Interval

Local Transaction: After the above replication transaction is created for a set of requests, a local transaction is created for each request on OID with the following parameters:

T_{local} *(opers(OID), per,rel,dl,exec,LSiteID)*

where *opers(OID)* is *read(OID)* on LSiteID and e*xec* is execution time of the read.

3.3.2. Updates. For an update, again, two cases must be considered. If the data to be updated is on the same site as the update, then it is a simple write to the object. Otherwise, after the write, the updated object copy must be written back to its originator site so that all other transactions that access the object can see this new value.

Case 1: If RSiteID = LsiteID

In this case the update maps to a local transaction as follows:

T_{local} *(opers(OID), per, rel, dl, exec, LSiteID)*

where *opers*(OID) is *write(OID)* on local site of *OID* and e*xec* is the execution time of *write(OID).*

Case 2: If RSiteID ≠ LSiteID

In this case, the update maps to a local transaction that writes to the local copy of the object, and a replication transaction that copies the updated object to its originator site. Each update maps to a separate a replication transaction, with one exception, described later in this section.

Local Transaction. The local update transaction writes to the local copy and is in the following form:

T_{local} *(opers(OID), per, rel, dl, exec, LSiteID)*

where *opers(OID)* is *write(OID)* on the LsiteID and e*xec* is the execution time of *write(OID).*

Replication Transaction. The replication transaction is a copy back transaction. It reads the local copy of the object and writes it to its originator site. This replication transaction is defined as follows:

T_{rep} *(opers(OID), per, rel, dl, exec, LSiteID)*

Here, *opers(OID)* are *read(OID)* on LSiteID and *write(OID)* on the RSiteID. *Per* is same as the period of the local transaction, *rel* is equal to the deadline of the local transaction to ensure that the local write is complete before the read begins. *Dl* is the end of the period to allow maximum time for the transaction. *Exec* is the sum of the execution times of *write(OID)* and *read(OID)* plus the network delay.

Although each local update transaction generally requires a replication transaction to copy back the data that it writes, some unnecessary replication transactions can be eliminated. The possible cases for eliminating the replication transactions are:

a) If more than one local transaction has the same release time and deadline, then only one of these local transactions needs to be copied back.
b) If more than one update has the same period and starts at the same time, only the update with the shortest deadline (highest priority) creates the replication transaction.

3.4. Schedulability Model

In order to analyze and execute the transactions created by the RM, we map the local and replication transactions to the DPCP model. There are two types of objects in the database on each site: a local object, which is local to the particular site and not replicated on any other site, and a replicated object, which has copies on multiple sites in the database.

After the JITRTR algorithm translates the requests/updates into the set of local and replication transactions, it determines whether each request/update is made on a local object or on a replicated object. It then assigns priorities to all transactions based on the deadline monotonic algorithm. Once the priorities are assigned, the algorithm maps the transactions to the DPCP model.

A local object is mapped to local resource and a local transaction is mapped to local critical section. Similarly, a replicated object is mapped to a global resource. A replication transaction accesses both a local resource (local copy of object) and a global resource (original object on remote site). Therefore, the replication transaction has both a local and a global critical section. Given this mapping, the system can be analyzed for schedulability, and executed using DPCP for resource access control if found to be schedulable.

4. Theorems

This section provides analysis of the JITRTR algorithm. We state and prove three theorems that indicate the correctness and goodness of the algorithm.

***Theorem 1*: All requests will always access temporally consistent data.**

Proof: Consider a replication transaction T_O that copies object O. Let d be the deadline of T_O as computed by the JITRTR algorithm. Let OV_i be the point in time in the i^{th} period after which the copy of the object O becomes invalid and let P be the period of T_O.

O is temporally inconsistent in the i^{th} period in the interval between OV_i and d (see Figure 2). Thus if we prove that no request executes in the invalid interval, then we have proven that all requests access temporally valid data.

Recall from the JITRTR algorithm that there are three possible cases considered when the deadline of T_O is computed. We re-examine these cases to prove that no request executes in the invalid interval.

Case 1) No requests execute in the invalid interval. Clearly, in this case no requests read invalid data.

Case 2) Some request starts at time x_i such that $OV_i < x_i \leq P_i + d$. The JITRTR algorithm changes d to $x_i - P_i$ reducing the size of the invalid interval and making the replication transaction finish before any requests read the data. Thus any such requests will read valid data in this case.

Case 3) Some request executes throughout the invalid interval starting before or at OV_i and finishing at or after the deadline expiration, $P_i + d$. Then the JITRTR algorithm computes d to be $OV_i - P_i$. Again, this deadline assignment ensures that before the object becomes invalid, a new valid value has been written. Thus, the invalid interval is removed and the request reads valid data.

In all the cases, we have proven that the requests read valid data. □

***Theorem 2*: The period of the replication transaction T_O must be equal to the period of the sensor update transaction for object O in order for all requests to read valid data using our algorithm.**

***Proof*:** To prove that the period of the replication transaction and the period of sensor update must equal, let us consider a contradictory situation. Let us assume they are not equal i.e. assume that the periods P_{To} for replication transaction T_O and for P_{SUo} for sensor update transaction SU_O are equal.

Here we consider the two cases. The first case is that $P_{To} > P_{Suo}$ and the second case is that $P_{To} < P_{Suo}$. We prove that it is not possible to construct a replication transaction with the above two cases.

Case 1) $P_{To} > P_{Suo}$

As discussed in the proof of Theorem 1, the object is invalid only in the invalid interval. Consider the calculations of deadline in each period of the replication transaction.

Initially d is set to the length of the period of the replication transaction. In each successive period, deadline is calculated based on whether there are any requests in the invalid interval. The minimum deadline in any period *i* is OV_i - P_i. So, once the deadline in any period becomes OV_i - P_i the calculation of deadline is stopped and the final deadline is taken as OV_i - P_i. It can be observed from the Figure 3 that, since $P_{To} > P_{Suo}$, as *i* increases, *d* decreases and at some point (for some *i*), *d* becomes 0 or less than 0.

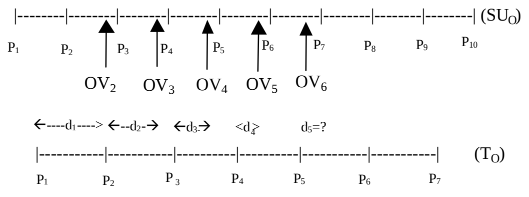


Figure 3 - Deadline Assignment - $P_{To} > P_{Suo}$

Thus we cannot guarantee that all the requests will read the valid data all the time.

Case 2) $P_{To} < P_{Suo}$

From Figure 4, it can be observed that, since the periods are not equal, there may be a case (P_4 in Figure 4) where we cannot choose a deadline that will satisfy all requests reading valid data.

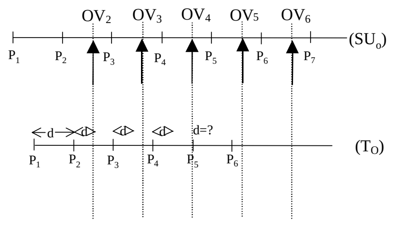


Figure 4 - Deadline Assignment - $P_{To} < P_{Suo}$

In this second case again, we cannot guarantee that a request will always read valid data.

Now we show that if $P_{To} = P_{Suo}$, we do not come across the problem of deadline becoming 0. It can be observed from Figure 5 that, between the start of every period of T_O and every period of OV_i there is always some constant time, which means they do not coincide. So, deadline can never be 0 in this case and all the requests read valid data .

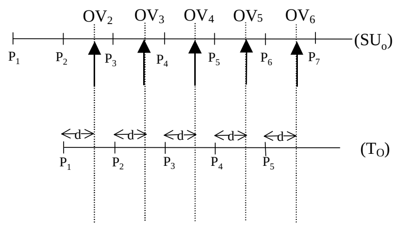


Figure 5 - Deadline Assignment 3

Thus the period of T_O must be equal to the period of sensor update transaction for object *O* in order for all requests to read valid data. □

Theorem 3: **The deadline assignment for a replication transaction from a request, made by the JITRTR algorithm, is necessary and sufficient for ensuring the temporal consistency of data.**

Proof: We first prove the sufficient condition, and then we prove the necessary condition.

Sufficient condition: Theorem 1 proves that requests always read temporally consistent data, which means that the deadline assignment is sufficient for ensuring the temporal consistency of data.

Necessary Condition: Theorem 1 considers all the three cases for computing the deadline and proves that all the requests always read the valid data. To prove that the deadline assignment of replication transaction, T_O, according to our algorithm is necessary, let us take the contradictory situation. That is, let us assume that there exists a deadline assignment *d'* of a replication transaction by some algorithm, other than JITRTR algorithm, greater than the deadline assigned by the JITRTR algorithm.

As discussed above, the object is invalid in i^{th} period only in the invalid interval. Again, we examine the three cases described in Theorem 1.

Case 1) There are no requests in the invalid interval. The JITRTR algorithm assigns the deadline to be $d=P_{i+1} – P_i$. This is the maximum deadline in the period. So *d'* cannot be greater than *d*.

Case 2) There is a request x_i such that $OV_i<x_i\leq P_i+d$. The JITRTR algorithm assigns deadline as $d=x_i-P_i$. If $d'>d$, then a request reading the object in the interval $(x_i,P_{i+1}]$ could read invalid data.

Case 3) There is a request at or before OV_i, continuing into the invalid interval. The JITRTR algorithm assigns the deadline to be $d=OV_i-P_i$. If $d'>d$, then a request in the interval $(OV_i, P_{i+1}]$ could read invalid data.

This implies that the deadline assignment by our algorithm is a necessary condition to ensure the temporal consistency of data read by the transactions.□

5. Performance Results

We tested the JITRTR algorithm to determine how it compares to two other techniques for accessing remote data in a distributed database. We compared it to full replication in which every object is fully replicated in the database, and every update is propagated to every copy. We also compared our algorithm to no replication in which data is accessed directly on the remote site.

The performance measure that we chose was percent schedulability. That is, given a random system specification, how often does the JITRTR algorithm produce a system that is schedulable, where all deadlines can be met. We also measured percentage of task schedulability to indicate how many tasks in a given system are found to be schedulable. These measures were chosen because we wanted to determine what kind of effect the additional overhead incurred by the replication transactions would have on the schedulability of the system.

To implement the tests, we created a simulation environment in which system specifications were randomly generated. The system specifications provided input to the JITRTR algorithm, and the resulting transactions were tested for schedulability using the RapidRMA [13] schedulability analysis tool. We also simulated an algorithm for creating full replication transactions and no replication transactions for comparison. All tests were averaged over 15 trials.

We performed a baseline test, and three test suites, examining the effects of the length of the period, the number of database objects, and the percentage of updates in the system. Due to lack of space, this paper describes the baseline test and one of the test suites. The results of the other test suites were very similar. For complete description of these results see [15].

5.1. Baseline Testing

Table 3 shows the parameters, and ranges of values used in the baseline test. These values were also used in the other test suites, with all parameter ranges remaining constant except for the parameter being tested.

Parameter	**Range**
Period	150 – 400
No of Objects	7 – 12
No of sites	5 – 12
No of Reqs/Object	3 – 6
percentage of Updates	50

Table 3 - Baseline Parameters

The resulting schedulability of the system for the three strategies is shown in Figure 6. It can be observed that the schedulability percentage for the *FullRep* algorithm is less than that for *JITRTR*. This is because *FullRep* consists of more transactions in the system than the other two replication strategies, for the same requests and updates.

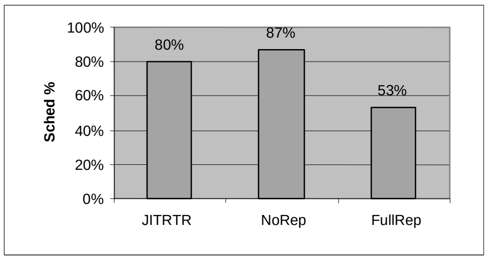


Figure 6 – System Schedulability for Baseline Testing

The schedulability percentage for *NoRep* is slightly more than that for JITRTR replication. This is because the *NoRep* algorithm has fewer transactions than the JITRTR algorithm because JITRTR creates a replication transaction for each remote data access. However the JITRTR algorithm guarantees that the objects read by the requests are always valid whereas *NoRep* does not make this guarantee. That is, even if the data on the remote site may be valid at the time it reads, it may become invalid while transferring it to the local site.

The task schedulability results (Figure 7) were similar to the system schedulability results. High percentage schedulability for JITRTR and *NoRep* shows that the system, even if it is not schedulable, is nearer to schedulability.

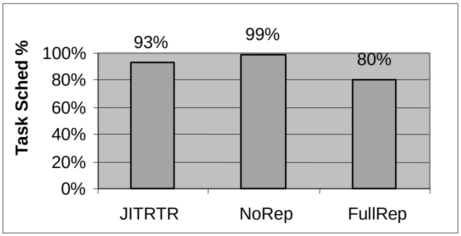


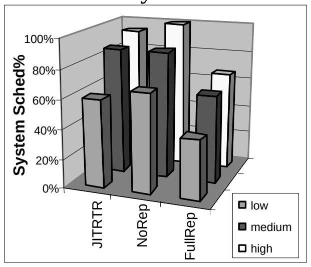
Figure 7 – Task Schedulability for Baseline Testing

5.2. Effect of Period

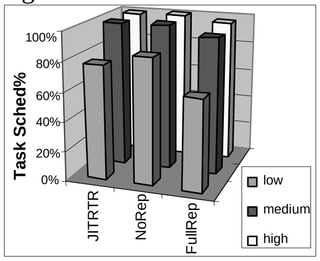
This test suite was performed to show the effect of the length of the period of a request or update on the schedulability of transactions executed by our algorithm. The three different ranges of period we chose are 100-250, 250-350, and 400-600.

System schedulability (Figure 8a) for all three strategies is found to be less than that for base case. As we expected, the figure illustrates that increase in the period increased the schedulability percentage of the system. We again see that the schedulability for JITRTR is just slightly lower than NoRep, indicating that the gain we get in the guarantee of valid data incurs only a minimal amount of extra overhead.

The task schedulability results (Figure 8b) were found to be similar to the results obtained for system schedulability percentage, but the percentage of schedulability is found to be higher.



a) System Schedulability



b) Task Schedulability

Figure 8 –Results on Effect of Period

6. Conclusions and Future Work

In this paper we have presented an algorithm for replication of data in a distributed real-time object-oriented database. The algorithm works in a static environment in which data requirements are known a priori. It analyzes the requirements of clients that will use the DRTOODB, and creates transactions that will make the data available, and that guarantee that only valid data will be read. We have proven that the algorithm uses necessary and sufficient conditions for providing valid data to all requests. The results of the performance tests were as we expected. They indicate that the benefit of guaranteed temporal validity outweighs the slightly higher overhead that is incurred over a no replication strategy.

We have also developed an extension of this algorithm designed to provide finer granularity of data access. The algorithm, called the Just-In-Time Real-Time Replication Affected Set (JITRTR-AS) algorithm, assumes that requests can access methods of objects as opposed to entire objects. It relies on a method-based extension of the DPCP (DASPCP) for schedulability analysis [14]. Due to space limitations, we have not presented this work here. For more information on this extension, see [15].

Finally, it is important to note that while this algorithm, and its method-based extension, show much promise towards providing real-time data replication, the assumptions upon which they are based can be quite limiting. In future work, we plan to explore how some or all of these assumptions can be relaxed or removed. For example, we would like to extend these algorithms to work in a dynamic system in which data requirements may not be periodic, and may not all be known a priori. In such a system, the algorithm would have to be changed to adapt to a changing environment. That is, something similar to the JITRTR algorithm could be used based on an initial specification of system requirements. Then, as these specifications change, an on-line version of the algorithm would reset the parameters of the replication transactions to reflect the new requirements.

References

[1] V. F. Wolfe, L. C. DiPippo, Real-Time Databases; chapter in *Database Systems Handbook,* P. Fortier and A. Rose, eds.; Multiscience Press; 1997.

[2] V.F. Wolfe, J.J. Prichard L.C. DiPippo and J. Black, The RTSORAC Real-Time Object-Oriented Database Model and System, chapter *in Real-Time Database Systems: Issues and Applications*, K.-J. Lin and S. Son eds., Kluwer Academic Press, 1997.

[3] L. Zhou, E. A. Rundensteiner, K. G. Shin, Schema Evolution of an Object-Oriented Real-Time Database System for Manufacturing Automation, *IEEE Transactions on Knowledge and Data Engineering*, Nov.-Dec. 1997 (Vol. 9, No. 6) pp. 956-977.

[4] J. Taina, S. H. Son, Requirements for Real-Time Object-Oriented Database Models -- How Much Is Too Much? *Proceedings of the 9th Euromicro Workshop on Real Time Systems*,1997.

[5] R. Ghaly and N. Prabhakaran: Modeling of a Real-Time Object-Oriented Database Schema, *Proceedings of the 2nd Annual Conference on Productivity through Computer Integrated Engineering & Manufacturing*, Orlando, Florida, Nov. 13-15, pp. 83-86, 1989.

[6] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, vol. 30, pp. 46-61, January 1973.

[7] J. W.-S. Liu. *Real-Time Systems*. Prentice-Hall, Fall 2000.

[8] R. Thomas, A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180-209 (1979).

[9] A. Burger, V. Kumar and M. Hines, Performance of Multiversion and Distributed Two-Phase locking Concurrency Control Mechanisms in Distributed Databases, *Information Sciences An International Journal.* Volume 1-2,1996.

[10] A. Thomasian, Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing, IEEE Transactions on Knowledge and Data Engineering, January/February 1998 (Vol. 10, No. 1) pp. 173-189.

[11] M. Carey., and M. Livny, Conflict Detection Tradeoffs for Replicated Data, *ACM Transactions on Database Systems*, Vol. 16, pp. 703-746, 1991.

[12] M. Xiong, K. Ramamritham, J. Haritsa, J. A. Stankovic, MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases, *Workshop on Advanced Issues of E-Commerce and Web/based Information Systems* (1998).

[13] TriPacific Software, Inc. www.tripacific.com.

[14] M. Squadrito, L. Esibov, L.C. DiPippo, V. F. Wolfe, G. Cooper, B.i Thurasingham, P. Krupp, M. Milligan, R. Johnston, R. Bethmangalkar, The Affected Set Priority Ceiling Protocols for Concurrency Control in Real-Time Object-Oriented Systems, *The International Journal of Computer Systems Science and Engineering*; vol. 14, no. 4., July 1999.

[15] P. Peddi, *A Replication Strategy for Distributed Real-Time Object-Oriented Databases*, University of Rhode Island Technical Report TR01-282, May 2001.