Concurrency Control for Distributed Real-Time Databases

Lui Sha, Department of CS and SEI
Ragunathan Rajkumar, Department of ECE
John P. Lehoczky, Department of Statistics
Carnegie Mellon University

Abstract

The concurrency control of transactions in a real-time database must satisfy not only the consistency constraints of the database but also the timing constraints of individual transactions. In this paper, we present a real-time concurrency control protocol that can be used in a distributed and decomposable real-time database. The protocol is based on the integration of a modular concurrency control theory with a real-time scheduling protocol called the priority ceiling protocol. This protocol supports the replication of data objects and avoids the formation of deadlocks. Finally, an analysis of the performance of this protocol is presented.

1. Introduction

1.1. Background

Real-time databases are becoming increasingly important in a wide range of applications such as aircraft tracking and the monitoring and control of modern manufacturing facilities. In a real-time database context, concurrency control protocols must not only maintain the consistency constraints of the database but also satisfy the timing requirements of the transactions accessing the database. In standard database applications, such as banking, one can lock data objects and prevent other transactions from accessing them in order to maintain consistency. In a real-time application such as tracking, the consistency is still important, but it is also critical that the values of the data objects are as timely as possible. If an object being tracked is moving fast, very stringent timing requirements will be placed on transactions that update the object location. Failure to meet these timing requirements will render the tracking exercise a failure, because the data will be out-of-date.

Typically, a concurrency control protocol designer does not consider the timing requirements of transactions. Rather, one designs a protocol which maximizes concurrency subject to maintaining the consistency of the database. A widely used concurrency control protocol is the two phase lock protocol [1], which was shown to be optimal in the sense of offering maximal concurrency when only transaction syntax information is utilized [3]. Each concurrency control protocol has a set of associated schedules: the interleavings of transaction steps that are permitted by the protocol. Protocol **X** is said to be more concurrent than protocol **Y** if the schedules associated with **X** contain those associated with **Y**. For example, the two phase lock protocol with read and write semantics is more concurrent than a two phase protocol using only write (exclusive lock) semantics.

On the other hand, a real-time scheduling protocol designer is typically not concerned about the issue of database consistency. Rather, one seeks to prioritize the execution of tasks in a way that maximizes the schedulability, which is the worst-case resource utilization bound below which the timing constraints of a set of periodic tasks can be guaranteed. For example, the rate monotonic scheduling algorithm was shown to be the optimal static priority

This work was sponsored in part by the Office of Naval Research under contract N00014-84-K-0734, in part by Naval Ocean System Center under contract N66001-87-C-0155, and in part by the Federal Systems Division of IBM Corporation under University Agreement YA-278067.

scheduling algorithm for periodic tasks [7]. The worst-case utilization bound for any task set was shown to be $ln\ 2$ (0.693) [7], while the worst-case utilization bound for a randomly chosen task set is likely to be no less than 0.88 [5]. These results are, however, derived under the assumption of perfect preemption: when a high priority task becomes ready to execute, it can immediately preempt the lower priority task which is currently executing. That is, it is assumed that tasks do not share data.

A real-time concurrency control designer, however, must address both consistency and schedulability. Intuitively, it seems that a higher degree of concurrency is always helpful. However, concurrency is just a measure of permissible interleavings of transaction steps. In real-time scheduling, it is important to have task preemptability. We wish to select those schedules that result in transactions meeting their timing requirements and avoid the forms of interleavings that lead to prolonged blocking. Maximizing concurrency without providing a means of selecting from among the permissible schedules can lead to very poor real-time performance. As we shall see in Section 3.3, providing read and write locks can lead to poorer schedulability than using exclusive locks alone if the writer has higher priority than the readers.

To maintain the database consistency and to provide a high degree of schedulability, we should use concurrency control protocols that minimize the duration of worst case blocking. In addition, we must properly manage the task priorities when blocking occurs. There are two important aspects in our approach. First, preemptability is promoted by the decomposition of the database and the transactions using the theory of modular concurrency control [12]. Secondly, priority during blocking is managed using the priority inheritance approach [10]. Since the theory of modular concurrency control [12] has already appeared, we will focus on the real-time scheduling aspects of concurrency control. In Section 2, we review the basic concepts of this approach. In Section 3, we formalize these concepts and in Section 4 we present our conclusion.

2. A Conceptual Framework

2.1. Concurrency Control

The concurrency offered by serializability theory is limited, and this has motivated research in non-serializable concurrency control methods [2, 8, 9, 12]. From a real-time scheduling point of view, the decomposition approach used in [12] promotes preemptability by reducing the duration of blocking caused by concurrency control. This approach has three aspects:

1. The decomposition of the database (shared data objects) into *atomic data sets* (ADS). The consistency of each atomic data set can be maintained independent of the other atomic data sets. In addition, the conjunction of all the ADS consistency constraints is equivalent to the consistency constraints of the database, and the union of atomic data sets is also an atomic data set.

2. The decomposition of each transaction (referred to as a *Compound Transaction* in [12]) into a partially ordered set of *elementary transactions*. Each elementary transaction maintains the consistency of the database and satisfies its own post-conditions when executing alone. In addition, the conjunction of the post-conditions of the elementary transactions of a compound transaction in any execution path is equivalent to the post-conditions of the compound transaction.

3. The use of a locking protocol to ensure that elementary transactions are run serializably with respect to each of the atomic data sets. When a set of transactions are executed setwise serializably, the corresponding schedule is called a *setwise serializable* schedule. When compound transactions are decomposed into elementary transactions and only the elementary transactions are scheduled setwise serializably, the corresponding schedule is called a *generalized setwise serializable schedule*. It was shown in [12] that the generalized setwise serializable schedules form a superset of sets of serializable schedules. In addition, generalized setwise serializable schedules possess the properties of consistency, correctness and modularity [12].

A real-time database can often be decomposed into disjoint sets of database objects that can be modeled as *atomic data sets*. For example, suppose that an airplane is being tracked by two radar stations and that the collection of data objects O_1 and O_2 represent the local views of these two stations. These data objects might include the current location, velocity, and identification of the airplane as seen by the particular station. Each of these two data objects forms an atomic data set. This is because the consistency constraints associated with each track can be checked and validated locally. Each new scan creates a new version of the data object, and in the course of time, the values of O_1 and O_2 form two correlated multivariate time series. The correlation can be used to create a global data object O_3. Together, the union $G = \{O_1, O_2, O_3\}$ represents the global and local views of the airplane under surveillance. That is, we have a tree structured global atomic data set, G, with global track O_3 as the root and two local tracks O_1 and O_2 as the two leaves.

The notion of atomic data sets is especially useful for tracking multiple targets. Before the formation of the global atomic data sets, we need to correlate all the local tracks near each other to find out which tracks are associated with which targets. Once a global ADS associated with a particular target is formed, the information at the root level can be referenced as a global context to aid the local operations. In addition, the knowledge of a set of local tracks belonging to the same global ADS helps to reduce the number of correlation operations. When data from new scans are used to update the local tracks of a global ADS, we will correlate them first. If the correlation is successful, then the hypothesis that all the ADSes in G are tracking the same target is considered to be valid and there is no need to further correlate the data with the tracks from other ADSes. This can substantially reduce the computation time required in tracking.

Finally, it is important to emphasize that while the ADSes in G are linked by correlation, the degree of correlation is not a consistency constraint. Consistency constraints are relationships between data objects that must be maintained by all the transactions. On the other hand, transactions do not have any responsibility to maintain the correlation between the ADSes in G. In fact, if the initial correlation is due to some signal processing error, then the global ADS will become invalid and must be eliminated, the sooner the better. Conceptually, the birth (the creation of a global track), growth (adding/deleting branches and leaves) and death (the global track is invalidated by new findings) of global ADSes model the dynamics of a tracking database.

To illustrate the syntax of a compound transaction, consider a simple distributed tracking database consisting of a global database GD and a set of local databases LD_1, ..., LD_n. Each local database is associated with a particular sensor, and the data from the sensor is processed to create local tracks. Compound transaction Global_View reads the local tracks of a global ADS, correlates them and then updates the global track if the correlation is successful. Figure 2-1 is the pseudo-code of this transaction.

This example illustrates the following characteristics of our approach. The database is decomposed, and the compound transaction models a generalized task which has both database and non-database operations. The basic units of database operations are elementary transactions operating on atomic data sets. Originally, the theory of modular concurrency control [12] did not explicitly address the issue of non-database operations. However, a non-database operation such as the data processing task in this example can be viewed as a special elementary transaction which locks some dummy data objects that no other transaction will either read or write. Hence, no special treatment is needed from a concurrency control point of view. However, from a real-time scheduling point of view, it is important to distinguish them, because one can cause blocking while the other cannot. From here on, we will refer to an elementary transaction which is a non-database operation as an elementary task and the phrase, "elementary transaction", will be reserved for those elementary transactions operating upon the database. Finally, we will use the terms "tasks" and "compound transactions" interchangeably.

A simple locking protocol for elementary transactions is the setwise two phase lock protocol: an elementary

```
Compound_Transaction Global_View;
AtomicVariable obj_new_vector: Track_Vector;
BeginSerial
    BeginParallel
            Elementary_Transaction Read_Local_ADS_1
            BeginSerial
                Lock Local_Track_1;
                obj_new_vector := Local_Track_1;
                Commit and Unlock Local_Track_1
            EndSerial;

            ...

            Elementary_Transaction Read_Local_ADS_n
            BeginSerial

                ...

            EndSerial

    EndParallel;

    Elementary_Task
    BeginSerial
     Correlate data stored in obj_new_vectors;
    EndSerial;

    If correlation_successful Then
    Begin
      Elementary_Transaction Global_View
      BeginSerial
        Lock  Global_Track;
        Update Global_Track;
        Commit and Unlock Global_Track;
      EndSerial;
    End;
    Else Begin
       Correlate the data with nearby tracks from other ADSes ..
    End;
EndSerial;
```

Figure 2-1: Compound Transaction *Global_View*

transaction cannot release any lock on any atomic data set until it has obtained all the locks on that atomic data set. Once it has released a lock on any atomic data set, it cannot obtain a new lock on that atomic data set. It can, however, obtain new locks on different atomic data sets. Elementary transactions pass the results of their computation by using atomic variables, which are local variables of the compound transaction shared by all its elementary transactions. Although the theory of modular concurrency control permits an elementary transaction to hold locks across atomic data sets, holding locks across atomic data sets prolongs the duration of locking and degrades preemptability. In this paper, we assume that elementary transactions do not hold locks across atomic data sets.

Having reviewed the basic concepts of the modular concurrency control theory [12], we now turn to the issues of distributed databases. In a distributed database environment, an atomic data set is a logical unit of data objects for distribution. An elementary transaction operating upon an ADS represents an atomic unit of operation for concurrent execution. For example, a global ADS G can be the union of many local track ADSes and a global track ADS. Each ADS in the union can reside on a different computer. To address the problem of reliability, we can (partially)

replicate an atomic data set on another processor. It may seem that the replicated ADS and the original ADS are part of a single atomic data set; however, they are *two* atomic data sets if one is allowed to be a historical version of the other [12]. For example, suppose that we have two ADSes: $A_1 = \{O_1\}$ and its copy $A_2 = \{O_2\}$. If we insist that A_1 and A_2 must be identical with respect to all references, i.e. $O_1 = O_2$, then these data objects will be part of a single ADS. The updates to them must appear to be an instantaneous event. This can be accomplished by using the setwise two phase lock to perform synchronous updates to O_1 and O_2. However, to satisfy the requirement of one being a historical copy of the other, A_1 and A_2 can be modeled as two ADSes and be updated asynchronously. The following is the pseudo-code of the compound transaction Update_Both that maintains a historical relationship between A_1 and A_2. Note that there is no attempt to ensure that other transactions will read identical versions of O_1 and O_2.

```
Compound_Transaction Update_Both;
BeginParallel
  Elementary_Transaction Update_Local_Copy_A1
  BeginSerial
    Lock O1
    O1 = 1;
    Unlock O1
  EndSerial;

  Elementary_Transaction Update_Remote_Copy_A2
  BeginSerial
    Lock O2
    O2 = 1;
    Unlock O2
  EndSerial;
EndParallel;
```

It might appear that allowing a copy to be a historical version of the data object sacrifices performance. However, the opposite is often true. Owing to communication delay, it is impossible to make the remote copies as up-to-date as the local copy. A concurrency control protocol can only guarantee that the local and remote copies are equally out-of-date. Making all the copies as out-of-date as the maximally delayed update of a remote copy is not helpful for dynamic applications such as tracking.

There are, of course, applications where a delayed but consistent view is better than just the latest information that can be obtained at each site. In an application like tracking, a local track would be updated periodically in conjunction with repetitive scans. Hence, in order to provide a delayed but consistent view in a distributed environment, we can utilize the periodicity of the writer as a time stamp concurrency control mechanism. For example, given an ADS and its replication, if for each data object there is only a single periodic writer and if the deadlines of this single periodic writer can be guaranteed on all the processors, then all these data objects will be updated by the end of the writer's period. That is, on each processor, during period n the versions of period $(n-1)$ are consistent. It is, of course, difficult to observe identical deadlines on local and remote processors because of the network communication delay. Typically, the versions of the data objects at remote sites will lag behind the local site. Thus, to ensure a network-wide consistent view, the concurrency control problem becomes a network-level real-time scheduling problem in which the time lags in the distributed versions are bounded. Once the lags are bounded, distributed tasks can read the proper versions of the distributed data objects and ensure that their decisions are based upon consistent data. In this paper, we will show that when all the tasks in every scheduling element in the network, e.g. processor and communication media, are scheduled by the rate-monotonic algorithm, the version lag between two replicated data objects in a network is bounded by the number of scheduling elements on its update path.

2.2. Priority Management

We have mentioned that in a real-time database system, a task typically has both database and non-database operations. In fact, the duration of non-database operations such as signal processing and correlation can be often much longer than that spent on reading and writing the database. Hence, in the investigation of real-time concurrency control protocol, we cannot overlook the scheduling of non-database operations . We can assign priorities to tasks (compound transactions) according to some real-time scheduling algorithm and let their elementary tasks and transactions use the assigned priorities. However, this is not sufficient. When tasks are not independent of one another, say, due to the sharing of database elements, *priority inversion* can exist. Priority inversion is said to occur when a high priority task is blocked by lower priority tasks. Priority inversion is inevitable in transaction systems. However, to achieve a high degree of schedulability in real-time applications, we must overcome the problem of uncontrolled priority inversion, where the priority inversion occurs over an indefinite period of time.

Example 1: Suppose τ_1, τ_2 and τ_3 are three tasks arranged in descending order of priority with τ_1 having the highest priority. Assume that elementary transactions $\mathbf{T}_1$ of task τ_1 and $\mathbf{T}_3$ of τ_3 access the same data object **O**. Suppose that at time t_1 transaction $\mathbf{T}_3$ locks **O**. During the execution of $\mathbf{T}_3$, the high priority task τ_1 arrives, preempts $\mathbf{T}_3$ and later attempts to execute $\mathbf{T}_1$ to access the object **O**. Task τ_1 will be blocked since **O** is already locked. We would expect that τ_1, being the highest priority task, will be blocked no longer than the time for transaction $\mathbf{T}_3$ to complete and unlock **O**. However, the duration of blocking may, in fact, be unpredictable. This is because transaction $\mathbf{T}_3$ can be preempted by the intermediate priority task τ_2 that does not need to access **O**. The blocking of $\mathbf{T}_3$, and hence that of τ_1, will continue until τ_2 and any other pending intermediate priority level tasks are completed.

The blocking duration in Example 1 can be arbitrarily long. This situation can be partially remedied if transactions are not allowed to be preempted; however, this solution is only appropriate for very short transactions, because it creates unnecessary blocking. For instance, once a long low priority transaction starts execution, a high priority transaction not requiring access to the same set of data objects may be needlessly blocked. An objective of this paper is to design an appropriate priority management protocol for a given concurrency control protocol so that deadlocks can be avoided and the duration of blocking is tightly bounded.

The use of *priority inheritance* is one approach to bound the arbitrary delays caused by a locking protocol. The basic idea of priority inheritance is that when a task τ's transaction **T** blocks higher priority tasks, the elementary transaction **T** is executed at the highest priority of all the transactions blocked by **T**. To illustrate this idea, let us apply this protocol to Example 1. Suppose that task τ_1's transaction $\mathbf{T}_1$ is blocked by task τ_3's transaction $\mathbf{T}_3$. The priority inheritance protocol stipulates that transaction $\mathbf{T}_3$ execute its transaction at task τ_1's priority. As a result, task τ_2 will be unable to preempt transaction $\mathbf{T}_3$ and will itself be blocked. In other words, the higher priority task τ_2 must wait for the elementary transaction of lower priority task τ_3 to be executed because transaction $\mathbf{T}_3$ "inherits" the priority of task τ_1. Otherwise, $\mathbf{T}_1$ will be forced to wait for task τ_2 to complete. When task τ_3's elementary transaction $\mathbf{T}_3$ completes, τ_3 returns to its assigned lowest priority and awakens transaction $\mathbf{T}_1$ waiting for the lock on shared data object **O**. $\mathbf{T}_1$, having the highest priority, immediately preempts τ_3 and runs to completion. This enables τ_2 and τ_3 to resume in succession and run to completion.

As we can see, this simple priority inheritance idea reduces the blocking time of a higher priority task from the entire execution times of lower priority tasks to only the duration of lower priority tasks' elementary transactions. However, this simple idea is not good enough for two reasons. First, the problem of deadlock has not been solved. Second, the blocking duration for a task, though bounded, can still be substantial, because a *chain* of blockings can be formed. For instance, suppose that task τ_1 needs to sequentially access objects $\mathbf{O}_1$ and $\mathbf{O}_2$. Also suppose that τ_2 preempts τ_3 which has already locked $\mathbf{O}_2$. Then, τ_2 locks object $\mathbf{O}_1$. Transaction τ_1 arrives at this instant and finds that the objects $\mathbf{O}_1$ and $\mathbf{O}_2$ have been respectively locked by the lower priority transactions τ_2 and τ_3. As a result, τ_1

would be blocked for the duration of two elementary transactions, once to wait for τ_2 to release O_1 and again to wait for τ_3 to release O_2. Thus, a chain of blockings can be formed.

The above two reasons motivate us to develop the *priority ceiling protocol*, which not only minimizes the blocking time of a task τ to at most one elementary transaction but also prevents the formation of deadlocks. The underlying idea of this protocol is to ensure that when a transaction T preempts another transaction, the priority at which this new transaction will execute must be guaranteed to be strictly higher than the priorities of all the preempted transactions, taking the priority inheritance protocol into consideration. If this condition cannot be satisfied, transaction T is suspended and the transaction that blocks T inherits T's priority. Example 2 illustrates this idea and the deadlock avoidance property of this protocol, while Example 3 illustrates the avoidance of chained blockings.

Example 2: Suppose that we have three tasks τ_0, τ_1 and τ_2 arranged in descending order of priority. In addition, there are two data objects O_1 and O_2 belonging to the same ADS A. We define the *priority ceiling* of a data object as the priority of the highest priority transaction that may lock this object. Suppose the sequences of processing steps for the transactions embedded in the three tasks are as follows:

$$T_0 = \{ \cdots, Lock(O_0), \cdots, Unlock(O_0), \cdots \}$$

$$T_1 = \{ \cdots, Lock(O_1), \cdots, Lock(O_2), \cdots, Unlock(O_2), \cdots, Unlock(O_1), \cdots \}$$

$$T_2 = \{ \cdots, Lock(O_2), \cdots, Lock(O_1), \cdots, Unlock(O_1), \cdots, Unlock(O_2), \cdots \}$$

The sequence of events described below is depicted in Figure 2-2. A line at a low level indicates that the corresponding task is blocked or has been preempted by a higher priority task. A line raised to a higher level indicates that the task is executing. The absence of a line indicates that the task has not yet arrived or has completed. Shaded portions indicate execution of elementary transactions.

Recall that the priority of transaction T_1 is assumed to be higher than that of transaction T_2. Thus, the priority ceilings of both objects O_1 and O_2 are equal to the priority of transaction T_1. Suppose that at time t_1, transaction T_2 has executed Lock(O_2). At this instant, task τ_1 is initiated and preempts transaction T_2. However, when task τ_1 tries to enter its elementary transaction at time t_2 by making an indivisible system call to execute Lock(O_1), the scheduler will find that task τ_1's priority is *not* higher than the priority ceiling of *locked* data object O_2. Hence, the scheduler suspends transaction τ_1 without locking O_1. Note that τ_1 is blocked outside its elementary transaction. Transaction T_2 now *inherits* the priority of task τ_1 and resumes execution. Since τ_1 is denied the lock on O_1 and suspended instead, a potential deadlock between T_1 and T_2 is prevented. If τ_1 were granted the lock on O_1, then τ_1 would later wait for τ_2 to release the lock on O_2, while τ_2 would wait for τ_1 to release the lock on O_1.

On the other hand, suppose that at time t_3, while T_2 is still in its transaction, the highest priority task τ_0 arrives and attempts to lock data object O_0. Since the priority of τ_0 is higher than the priority ceiling of locked data object O_2, task τ_0's transaction T_0 will be granted the lock on the data object O_0. Task τ_0 will therefore continue and execute its transaction, thereby effectively preempting T_2 in its transaction and not encountering any blocking. At time t_4, T_0 completes execution and T_2 is awakened for T_1 is blocked by T_2. T_2 continues execution and locks O_1. At time t_5, T_2 releases O_1. At time t_6, when T_2 releases O_2, task τ_2 resumes its assigned priority. Now, T_1 is signaled. Having a higher priority, it preempts T_2 and completes execution. Finally, T_2 resumes and completes.

Note that in the above example, τ_0 is never blocked. τ_1 was blocked by the lower priority task τ_2 during the intervals $[t_2, t_3]$ and $[t_4, t_6]$[2]. However, these two intervals correspond to the duration that T_2 needs to lock the two

[2]The interval $[t_3, t_4]$ is not considered blocking for τ_1 since it was only preempted by the higher priority task τ_0.

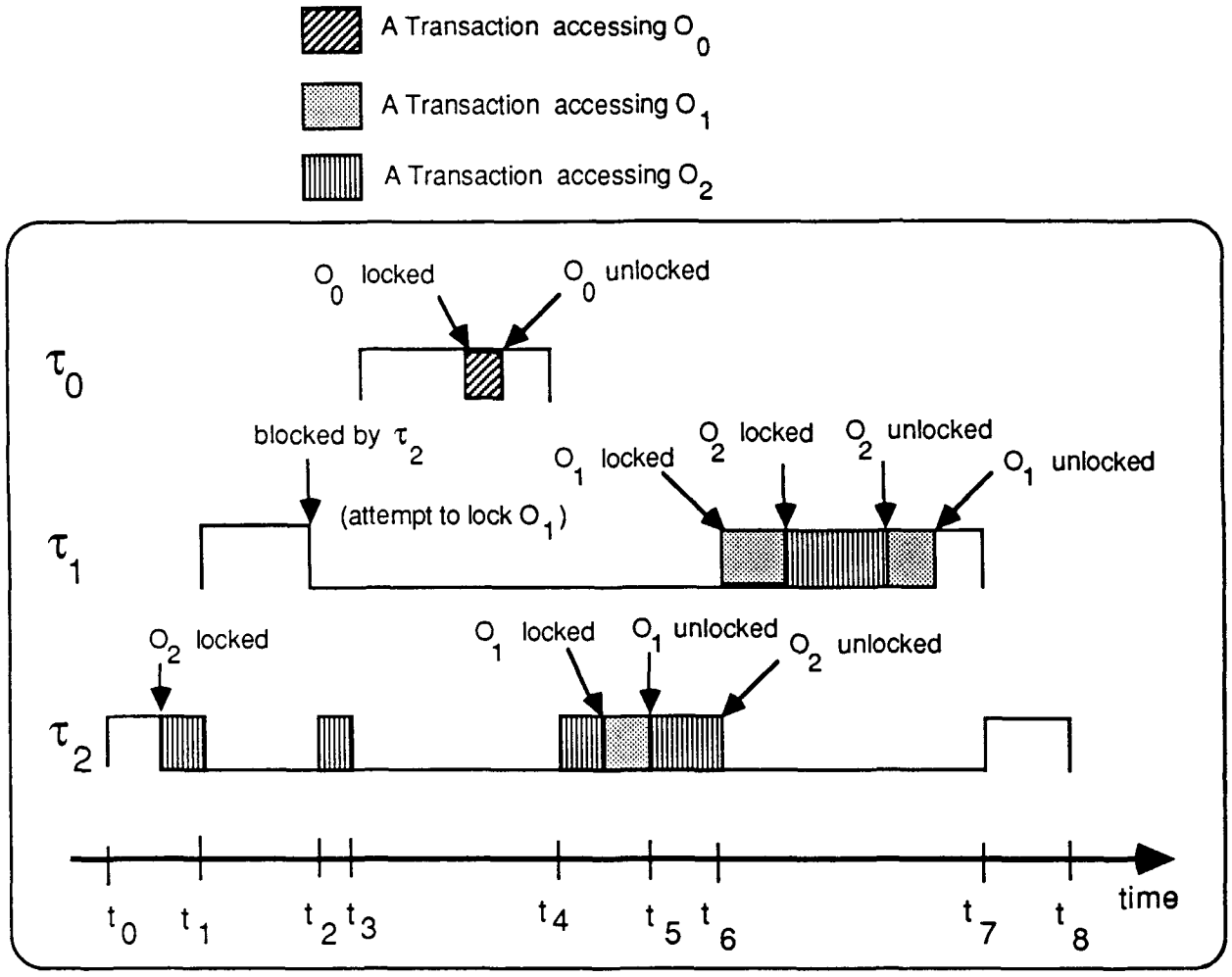

Figure 2-2: Sequence of Events described in Example 2.

data objects in $\mathbf{A}_2$. Thus, the blocking duration of τ_1 is equal to the duration of a single elementary transaction of a lower priority transaction $\mathbf{T}_2$, even though the actual blocking occurs over disjoint time intervals. It is, indeed, a property of this protocol that any task can be blocked by at most one lower priority elementary transaction until it suspends itself or completes. This property is further illustrated by the following example.

Example 3: Suppose that $\mathbf{A}_1 = \{\mathbf{O}_1\}$ and $\mathbf{A}_2 = \{\mathbf{O}_2\}$. Consider the example where a chain of blockings can be formed. We assumed that task τ_1 needs to access data objects $\mathbf{O}_1$ and $\mathbf{O}_2$ sequentially while τ_2 accesses $\mathbf{O}_2$ and $\mathbf{T}_3$ accesses $\mathbf{O}_1$. Hence, the priority ceilings of data objects $\mathbf{O}_1$ and $\mathbf{O}_2$ are equal to P_1, the priority of τ_1. As before, let τ_3 lock $\mathbf{O}_1$ at time t_0. At time t_1, task τ_2 arrives and preempts $\mathbf{t}_3$. However, at time t_2, when task τ_2 attempts to enter its transaction $\mathbf{T}_2$ and lock $\mathbf{O}_2$, the run-time system finds that the priority of τ_2 is *not* higher than the priority ceiling P_1 of the *locked* object $\mathbf{O}_1$. Hence, τ_2 is denied the lock on $\mathbf{O}_2$ and is blocked. Transaction $\mathbf{T}_3$ resumes execution at τ_2's priority. At time t_3, when $\mathbf{T}_3$ is still executing, τ_1 arrives and preempts $\mathbf{T}_3$. At time t_4, τ_1 attempts to enter its transaction $\mathbf{T}_1$ to lock $\mathbf{O}_1$ and is blocked by τ_3 which holds the lock on $\mathbf{O}_1$. Hence, τ_3 inherits the priority of τ_1. At time t_5, task τ_3 exits its transaction, resumes its original priority and awakens τ_1. Task τ_1, having the highest priority, preempts τ_3 and runs to completion. Next, τ_2 which is no longer blocked completes its execution and is

followed by τ_3. Again, note that τ_1 is blocked by τ_3 during the interval $[t_4, t_5]$ which corresponds to the single elementary transaction. Also, task τ_2 is blocked by τ_3 during the disjoint intervals $[t_2, t_3]$ and $[t_4, t_5]$ which also correspond to the duration of task τ_3's elementary transaction.

3. Theoretical Development

Having reviewed the basic concepts of our approach, we now formalize our approach. Due to page limitations, we omit the proofs of the lemmas that are intuitively clear. The proofs can be found in [13]. Before we begin the technical investigation, we first list our assumptions and state the notation used.

3.1. Assumptions and Notation

We assume that we have a set of loosely coupled uni-processors. In each processor there is a set of statically allocated periodic tasks. Note that a task can execute parallelly on more than one processor. For example, task τ can update an ADS and its replication on processors X and Y. Each stream of aperiodic tasks, if any, will be converted to periodic tasks via a periodic server task. For example, to handle random requests from an operator, we can buffer his input and use a periodic server task to respond to his requests periodically[3]. Since surveillance operations consist of both signal processing and database accessing, we assume that each instance of a periodic task is an interleaving of data processing code and database operations, which are modeled as elementary tasks and elementary transactions of a compound transaction respectively. We assume that the database is decomposed into atomic data sets and the setwise two phase lock protocol is used by elementary transactions for concurrency control. We assume that the rate-monotonic algorithm is used to assign a priority to each task. This algorithm assigns higher priorities to tasks with shorter periods and is an optimal static priority algorithm for periodic tasks [7]. If two tasks are ready to run, the higher priority task will run. Equal priority tasks are run in a FCFS order. We also assume that a transaction does not attempt to lock an object that it has already locked and thus deadlock with itself. In addition, we assume that in a uni-processor the runtime system will serialize the execution of syntactically parallel elementary tasks and transactions. For example, in a uni-processor if a compound transaction has the construction { .,. **BeginParallel** Elementary_Transaction_1; Elementary_Transaction_2; **EndParallel** ...}, then either Elementary_Transaction_1 completes before the start of Elementary_transaction_2, or vice versa. These two elementary transactions will, of course, execute in parallel, should they execute on different processors. A task can suspend itself during its execution of non-database operations, e.g. waiting for I/O. However, self suspension is not permitted when it holds locks on database objects. We also assume that the ownership of locks is mutually exclusive, that is, at any instant of time, at most one task can hold the lock on the data object **O**. This means that the *read* semantic of a lock cannot be used to allow several readers to hold the lock on the data object **O**. The *read* and *write* semantics of a lock is not utilized in this context, because it can increase the worst-case blocking. We will illustrate this after we prove that until a task completes or suspends itself, it can be blocked at most by one lower priority elementary transaction when exclusive locking is used.

Finally, the terms "task" and "compound transaction" will be used inter-changeably. In addition, the term "task" will refer to an instance of a periodic task unless explicitly stated otherwise. The term "blocking duration" refers to the time during which a higher priority task waits for lower priority tasks. The time that a task waits for higher priority tasks or equal priority tasks that have arrived earlier is *not* considered to be blocking.

Notation: We use the notation $\{\mathbf{A}_1, \cdots, \mathbf{A}_k\}$ to denote the atomic data sets of database **D**.

Notation: We denote the given tasks as an ordered set $\{\tau_1, \cdots, \tau_n\}$ where the tasks are listed in descending order

[3]For an advanced treatment of aperiodic tasks, readers are referred to [5].

of priority, with τ_1 having the highest priority.

Notation: We use $\mathbf{T}_{i,j}$ to denote an elementary transaction of task τ_i that accesses ADS $\mathbf{A}_j$. We will also use the simplified notation $\mathbf{T}_i$ when the identity of the ADS is not important.

Notation: We use the notation P_i to denote the priority of task τ_i.

Definition: The *priority ceiling* of a data object is defined as the priority of the highest priority transaction that may lock this object.

3.2. Definition of the Priority Ceiling Protocol

Having presented the basic idea of the priority ceiling protocol for tasks executing in each processor, we now present its definition.

1. Suppose that task τ has the highest priority among the tasks ready to run and is assigned the processor and let $\mathbf{O}^*$ be the data object with the highest priority ceiling of all data objects currently locked by transactions other than those of τ. Before task τ enters its elementary transaction **T**, it must first obtain the locks on the data objects. Task τ will be blocked and the lock on an object **O** will be denied, if the priority of task τ is not higher than the priority ceiling of data object $\mathbf{O}^*$[4]. In this case, task τ is said to be blocked by the task whose transaction holds the lock on $\mathbf{O}^*$. Otherwise, task τ's transaction **T** will obtain the lock on data object **O**. When a task τ exits its elementary transaction, the data objects associated with the transaction will be unlocked and the highest priority task, if any, blocked by transaction τ will be awakened.
2. A task τ's transaction **T** uses its assigned priority, unless it is in its transaction and blocks higher priority tasks. If transaction **T** blocks higher priority tasks, **T** *inherits* P_H, the highest priority of the tasks blocked by **T**. When task τ exits its transaction, it resumes its original priority. Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of original priority must be indivisible.
3. A task τ, when it does not attempt to enter an elementary transaction can preempt another task τ_L if τ's priority is higher than the priority, inherited or assigned, at which task τ_L or its transaction is executing.

We shall illustrate the priority ceiling protocol using an example.

Example 4: We assume that the priority of task τ_i is higher than that of transaction τ_{i+1}. We also assume that $\mathbf{A}_0 = \{\mathbf{O}_0\}$ and $\mathbf{A}_1 = \{\mathbf{O}_1, \mathbf{O}_2\}$. The processing steps in each transaction are as follows:

Task τ_0 has the elementary transactions $\mathbf{T}_{0,0}$ and $\mathbf{T}_{0,1}$ and performs the steps

$$\{\cdots, \mathit{Lock}(\mathbf{O}_0), \cdots, \mathit{Unlock}(\mathbf{O}_0), \cdots, \mathit{Lock}(\mathbf{O}_1), \cdots, \mathit{Unlock}(\mathbf{O}_1), \cdots\},$$

Task τ_1 contains the elementary transaction $\mathbf{T}_{1,1}$ by executing

$$\{\cdots, \mathit{Lock}(\mathbf{O}_2), \cdots, \mathit{Unlock}(\mathbf{O}_2), \cdots\},$$

and task τ's contains the elementary transaction $\mathbf{T}_{2,1}$ by executing

$$\{\cdots, \mathit{Lock}(\mathbf{O}_2), \cdots, \mathit{Lock}(\mathbf{O}_1), \cdots, \mathit{Unlock}(\mathbf{O}_1), \cdots, \mathit{Unlock}(\mathbf{O}_2), \cdots\}.$$

---

[4]Note that if **O** has been already locked, the priority ceiling of **O** will be at least equal to the priority of τ. Because task τ's priority is not higher than the priority ceiling of the data object **O** locked by another transaction, τ will be blocked. Hence, this rule implies that if a task τ's transaction **T** attempts to lock a data object **O** that has been already locked, τ will be denied the lock and blocked instead.

Note that the priority ceilings of objects $\mathbf{O}_0$ and $\mathbf{O}_1$ are equal to P_0, and the priority ceiling of object $\mathbf{O}_2$ is P_1. Figure 3-1 depicts the sequence of events described below. Figure 3-1 follows the same conventions as in Figure 2-2.

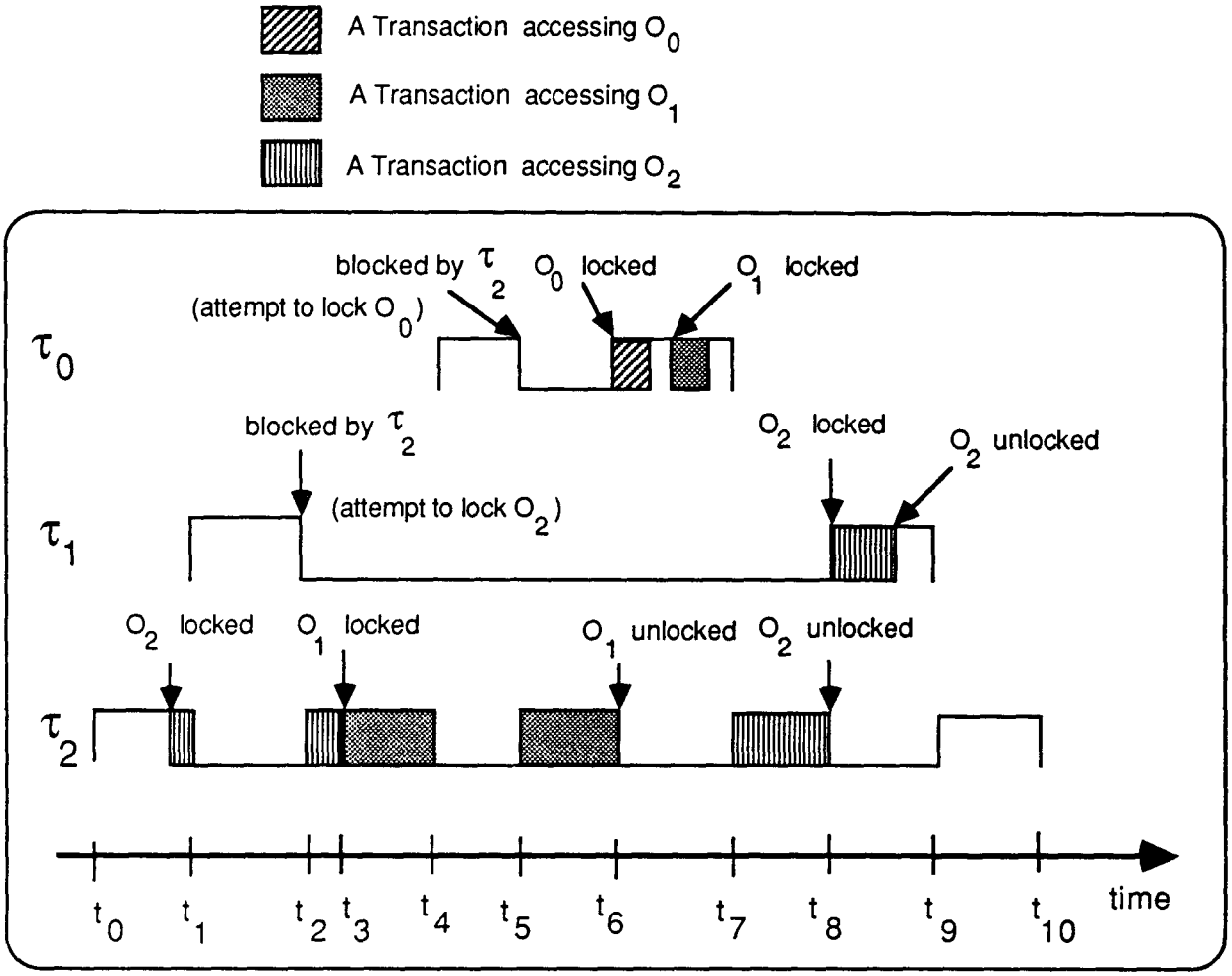

Figure 3-1: Sequence of Events described in Example 4.

Suppose that

- At time t_0, task τ_2's transaction begins execution and locks $\mathbf{O}_2$.
- At time t_1, task τ_1 arrives, preempts $\mathbf{T}_2$ and begins execution.
- At time t_2, when task τ_1 attempts to enter its transaction and access $\mathbf{O}_2$, task τ_1 becomes blocked. Task τ_2 now resumes the execution of its transaction $\mathbf{T}_{2,1}$ at its inherited priority of task τ_1, namely P_1.
- At time t_3, transaction $\mathbf{T}_{2,1}$ successfully locks $\mathbf{O}_1$, since there exist no objects locked by other transactions. Task τ_2's transaction still executes at the priority of P_1, because this is the priority of the highest priority transaction currently blocked by τ_2.
- At time t_4, task τ_2 is still executing $\mathbf{T}_{2,1}$ but the highest priority task τ_0 arrives. Task τ_0 preempts τ_2 and executes its own non-transaction code, because P_0, the priority of $\mathbf{T}_0$, is higher than P_1, the priority level at which $\mathbf{T}_{2,1}$ was being executed.

- At time t_5, task τ_0 attempts to enter its transaction and execute its transaction $\mathbf{T}_{0,0}$ by locking $\mathbf{O}_0$, which is not locked by any transaction. However, since the priority of transaction $\mathbf{T}_0$ is not higher than the priority ceiling P_0 of the locked object $\mathbf{O}_1$, task τ_0 is blocked by task τ_2's elementary transaction which holds the lock on $\mathbf{O}_1$. This is an example of *ceiling blocking* introduced by the priority ceiling protocol. At this point, task τ_2 resumes its execution of $\mathbf{T}_{2,1}$ at the newly inherited priority level of P_0.
- At time t_6, task τ_2's transaction completes its processing on the objects $\mathbf{O}_1$ and $\mathbf{O}_2$. Object $\mathbf{O}_1$ is now unlocked, task τ_2 returns to the previously inherited priority of P_1, and task τ_0 is awakened. At this point, τ_0 preempts task τ_2. Task τ_0 will be granted the lock on $\mathbf{O}_0$ and will execute its elementary transaction $\mathbf{T}_{0,0}$. This happens because τ_0's priority P_0 is higher than the priority ceiling P_1 of locked object $\mathbf{O}_2$.
- At time t_7, task τ_0 completes its execution (after locking, unlocking $\mathbf{O}_1$, and completing its non-transaction code), and task τ_2 resumes its execution of $\mathbf{T}_{2,1}$ at its inherited priority P_1.
- At time t_8, task τ_2 exits $\mathbf{T}_{2,1}$, object $\mathbf{O}_2$ is unlocked, task τ_2 returns to its own priority P_2 and task τ_1 is awakened. At this point, task τ_1 preempts task τ_2.
- Finally, task τ_1 completes its execution, and task τ_2 resumes its execution, until it also completes.

It is helpful to summarize that under the priority ceiling protocol, a high priority transaction can be blocked by a low priority transaction in one of three situations. First, there is *direct* blocking, a situation in which a higher priority transaction attempts to lock a locked object. Direct blocking is caused by the requirement for the consistency of shared data. Second, a medium priority task τ_1 can be blocked by a low priority task τ_2, which has inherited the priority of a higher priority task τ_0. We refer to this form of blocking as *inheritance* blocking. It is necessary in that it avoids having a high priority task τ_0 being indirectly blocked by a medium priority task τ_1. The third type of blocking is *ceiling* blocking, which occurs when a task cannot start the execution of a transaction because its priority is not higher than the priority ceiling of the data objects locked by transactions other than its own. An instance of this type of blocking occurs at time t_5 in the above example. Ceiling blocking is necessary to avoid deadlock and chained blocking. This avoidance approach belongs to the class of pessimistic protocols which sometime create unnecessary blocking. Although the priority ceiling protocol introduces unnecessary blocking, the worst case blocking for any task is reduced to the duration of at most one low priority elementary transaction until the task completes or suspends itself.

3.3. Properties of The Priority Ceiling Protocol

The priority ceiling protocol has two important properties. First, under this protocol, transactions cannot be deadlocked when objects within the same ADS are accessed concurrently. Second, under this protocol a task can be blocked for the duration of at most *one* lower priority elementary transaction until it completes or suspends itself. We prove both of these properties in this section.

Lemma 1: A task τ can be blocked by a lower priority task τ_L, only if τ_L is within its elementary transaction when τ arrives [13].

Lemma 2: A task τ can be blocked by a lower priority task τ_L, only if the priority of task τ is no higher than the highest priority ceiling of all the objects that are locked by task τ_L when τ arrives [13].

Lemma 3: Suppose that an elementary transaction of τ_j has locked data object $\mathbf{O}_m$ and is preempted by task τ_i and τ_i's transaction $\mathbf{T}_i$ locks $\mathbf{O}_n$, $n \neq m$. Under the priority ceiling protocol, task τ_j cannot inherit a priority level which is higher than or equal to that of task τ_i until task τ_i completes or suspends itself [13].

Lemma 4: The priority ceiling protocol prevents transitive blockings.

Proof: Suppose that transitive blocking is possible. Let T_n block transaction T_{n-1}, ..., transaction T_3 block transaction T_2, and transaction T_2 block transaction T_1. By Lemma 1, to block transaction T_2, task τ_3 must remain within its transaction, when T_2 arrives. By the same reasoning, to block T_1, task τ_2 must remain within its transaction T_2 when T_1 arrives. By the transitivity of the protocol, transaction T_3 will inherit the priority of T_1 which is assumed to be higher than that of transaction T_2. This contradicts Lemma 3, which stipulates that T_3 cannot inherit a priority that is higher than or equal to that of transaction T_2. The Lemma follows.

Theorem 5: The priority ceiling protocol prevents deadlocks.

Proof: First, by assumption, a transaction cannot deadlock with itself. Thus, a deadlock can only be formed by a cycle of transactions waiting for each other. Let the n transactions involved in the blocking cycle be $\{T_1, \cdots, T_n\}$. Note that each of these n tasks must be in their transactions, because a task that does not hold any lock on objects cannot contribute to the deadlock. By Lemma 4, the number of transactions in the blocking cycle can only be two, i.e., $n = 2$. Suppose that transaction T_2 was preempted by transaction T_1, which then locks the object O. By Lemma 3, transaction T_2 can never inherit a priority which is higher than or equal to that of transaction T_1 before transaction T_1 completes. However, if a blocking cycle (deadlock) is formed, then by the transitivity of priority inheritance, transaction T_2 will inherit the priority of transaction T_1. This contradicts Lemma 3 and hence the Theorem.

Remark: Suppose that the run-time system supports the priority ceiling protocol. The above theorem leads to the useful result that under the ceiling protocol, the setwise locking protocol becomes deadlock free. As long as each transaction does not deadlock with itself, there will be no deadlock in the system.

We now proceed to prove that under the priority ceiling protocol, a higher priority task τ can be blocked for at most a single lower priority elementary transaction until τ completes or suspends itself.

Lemma 6: Let task τ_L be a task with a lower priority than that of task τ. Suppose that τ does not suspend itself during execution. Task τ can be blocked for at most the duration of a single elementary transaction of τ_L.

Proof: Suppose that task τ's transaction T is blocked by a lower priority transaction T_L of τ_L. By Theorem 5, there is no deadlock and hence task τ_L will complete its current transaction at some instant t_1. Once task τ_L exits its transaction T_L at time t_1, transaction T_L is preempted by the transaction T of task τ. By the definition of the priority ceiling protocol and by Lemma 1, task τ_L cannot enter another elementary transaction until task τ has completed its execution. The Lemma follows.

Theorem 7: Suppose that a task does not suspend itself during execution. A task τ can be blocked for at most the duration of one lower priority elementary transaction.

Proof: Suppose that task τ can be blocked for the duration of n lower priority elementary transactions where $n > 1$. By Lemma 6, the only possibility is that transaction T is blocked by the transactions of n different lower priority tasks. Suppose that the first two lower priority tasks that block task τ are τ_1 and τ_2. By Lemma 1, in order for both these tasks to block τ, both of them must be in their transactions when task τ becomes ready for execution. Let the lowest priority task τ_2 enter its transaction first. And let the highest priority ceiling of all the objects locked by the transaction of τ_2 be ρ_2. Under the priority ceiling protocol, in order for transaction T_1 of τ_1 to lock any object when T_2 has already locked some data objects, the priority of task τ_1 must be higher than priority ceiling ρ_2. Since we assume that task τ can be blocked by transaction T_2, by Lemma 2 the priority of τ cannot be higher than priority ceiling ρ_2. Thus, the priority of task τ_1 is higher than ρ_2 and the priority of task τ is no higher than ρ_2. This means that task τ_1's priority must be higher than the priority of task τ. This contradicts the assumption that the priority of task τ is higher than that of both τ_1 and τ_2. Thus, it is impossible for task τ to have priority higher than both tasks τ_1 and τ_2 and to be blocked by both of them under the priority ceiling protocol. The Theorem

follows immediately.

Remark: While a detailed investigation of locking semantics in real-time systems is beyond the scope of this paper, it is still worthwhile to comment on this issue. We have shown that in a processor if only exclusive locking is used, then under the priority ceiling protocol a higher priority task will be blocked for at most one elementary transaction by lower priority tasks. However, a direct application of the read and write semantic can lead to poor performance. For example, suppose that we have ten transactions concurrently holding read locks on data object **O**. When a higher priority task arrives later and attempts to write **O**, it has to wait for all ten of these transactions to complete. Thus, if writing is frequent (as in tracking) and hence has higher priority than the readers, exclusive locking leads to better schedulability. On the other hand, if the writer has priority lower than the readers, then the use of read and write semantics improves schedulability. This issue becomes rather complicated when the writer has intermediate priority.

3.4. Schedulability Analysis

In this section, we develop a set of *sufficient* conditions under which a set of periodic tasks with hard deadlines at the end of the periods can be scheduled by the rate-monotonic algorithm [7] when the priority ceiling protocol is used. To this end, we will use a simplified scheduling model. First, we assume that all the tasks are periodic. An aperiodic task can be converted to a periodic task by the use of a periodic server task and buffering the arriving aperiodic tasks. Secondly, we assume each task has deterministic execution times for both its non-transaction code and transactions. Readers interested in more general scheduling issues, such as the reduction of aperiodic response times and the effect of stochastic execution times are referred to [6, 11]. In the present context, we assume that periodic tasks are scheduled by the rate-monotonic algorith, the optimal static priority algorithm [7].

We quote the following theorem also due to Liu and Layland [7] which was proved under the assumption of independent tasks, i.e. when there is no blocking due to data sharing and synchronization.

Theorem 8: A set of n periodic tasks scheduled by the rate-monotonic algorithm can always meet their deadlines if

$$\frac{C_1}{T_1} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n}-1)$$

where C_i and T_i are the execution time and period of task τ_i respectively.

Theorem 8 offers a sufficient (worst-case) condition that characterizes the rate-monotonic schedulability of a given periodic task set. An exact characterization of rate-monotonic schedulability can be found in [5].

When tasks are independent of one another and do not access shared data, Theorem 8 provides us with the condition under which a set of n periodic tasks can be scheduled by the rate-monotonic algorithm.[5] Although this theorem has taken into account the effect of a task being preempted by higher priority tasks, it has not considered the effect of blocking caused by the need to access shared data from a database. We now consider the effect of blocking.

Let Z be the set of elementary transactions that could block task τ. That is, each element in Z is an elementary transaction that is accessed by a lower priority task and that accesses a data object whose priority ceiling is higher than or equal to the priority of task τ. By Lemma 2 and Theorem 7, task τ can be blocked for at most the duration of a single element in Z if it does not suspend itself. Hence the worst-case blocking time for τ is z_i, the duration of the longest elementary transaction in Z when τ does not suspend itself. We denote this worst-case blocking time of task

[5]That is, the conditions under which all the instances of all the n tasks will meet their deadlines.

τ_i as B_i. Note that given a set of n periodic tasks, $B_n = 0$, since there is no lower priority task to block τ_n.

Theorem 8 can now be generalized in a straightforward fashion. In order to test the schedulability of τ_i, we need to consider both the preemptions caused by higher priority tasks and blocking by lower priority tasks along with its own utilization. The blocking of any instance of τ_i can be in the form of direct blocking, inheritance blocking or ceiling blocking but does not exceed B_i. Thus, Theorem 8 becomes

Theorem 9: Suppose that a task does not suspend itself. A set of n periodic tasks using the priority ceiling protocol can be scheduled by the rate monotonic algorithm if the following conditions are satisfied:

$$\forall\, i,\ 1 \le i \le n,\quad \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \le i(2^{1/i} - 1).$$

Proof: Suppose that for each task τ_i the equation is satisfied. It follows that the equation of Theorem 8 will also be satisfied with $n = i$ and C_i replaced by $C_i^* = (C_i + B_i)$. That is, in the absence of blocking, any instance of task τ_i will still meet its deadline even if it executes for $(C_i + B_i)$ units of time. It follows that task τ_i, if it executes for only C_i units of time, can be delayed by B_i units of time and still meet its deadline. Hence the theorem follows.

Corollary 10: If a task τ_i suspends itself at most k times, then the above theorem holds with $\mathbf{B_i} = kz_i$.

Remark: The first i terms in the above inequality constitute the effect of preemptions from all higher priority tasks and τ_i's own execution time, while B_i of the last term represents the worst case blocking time due to *all* lower priority tasks for one instance of task τ_i.

For example, suppose that we have three tasks. For task τ_1, we first check if equation "$C_1/T_1 + B_1/T_1 \le 1$" holds. Next, we check if equation "$C_1/T_1 + C_2/T_2 + B_2/T_2 \le 2(2^{1/2} - 1)$" holds for task t_2. Finally, we check if equation "$C_1/T_1 + C_2/T_2 + C_3/T_3 + B_3/T_3 \le 3(2^{1/3} - 1)$" holds for task τ_3. If all three equations hold, then the tasks can meet all their deadlines. The conditions, however, are not necessary.

Having discussed the scheduling issues within a uniprocessor, we now address the issues in a distributed environment.

Lemma 11: Let $\mathbf{A}_i[v]$ denote the version v of an ADS residing in processor i. Suppose that ADS $\mathbf{A}_1$ and its replications, $\{\mathbf{A}_1, ..., \mathbf{A}_n\}$, are distributed in n processors. Assume that a real-time scheduling algorithm guarantees that $max(v_i, v_j) \le k$, $1 \le i, j \le n$. That is, the maximal lag between the versions of these ADSes at different processors is bounded by k. At period n, we have $\{\mathbf{A}_1[n-k] = \mathbf{A}_2[n-k] = ... = \mathbf{A}_m[n-k],\ n \ge k\}$ available at all the n processors.

Proof: It directly follows from our single writer assumption and our assumption that the maximal lag is k.

Remark: Since we do not have deadlock within each processor and locks are not allowed to be held across processor boundaries, we do not have the problem of distributed deadlocks.

To calculate the version lags, each scheduling element on the path is counted as a node. Thus, if two replicated data objects reside in two processors connected by a communication medium, we consider that there are three nodes on the path: the local processor, the communication medium and the remote processor.

Theorem 12: When all the tasks in every element of the network are schedulable by the rate-monotonic algorithm, the version lag between two replicated data objects in a network is bounded by the number of nodes on its update path.

Proof: Suppose that replicated data objects $\mathbf{O}_1$ and $\mathbf{O}_2$ can be updated within the same period. The version lag is 1, because data object $\mathbf{O}_1$ can be updated before $\mathbf{O}_2$. But both of them will be updated by the

end of the same period. Thus, the lag between the versions of these two data objects is at most 1. By introducing each additional full period delay between the updates of the two data objects, the version lag increases by 1. That is, when the updates are separated by n periods, the version lag is at most $(n + 1)$. When there are k nodes on the path, O_1 will be updated by the end of the first period and O_2 will be updated by the end of the k^{th} period. That is, they are separated by at most $(k-1)$ periods. It follows that the version lag between them is at most $(k-1) + 1 = k$.

For example, suppose that we have a set of processors connected by a communication bus and that all the processors and communication bus are scheduled by the rate monotonic algorithm[6]. Let data objects O_1 and O_2 reside in two different processors and O_1 be at the home site. The rate monotonic algorithm guarantees that O_1 can be updated by the end of the first period. That is, we can initiate the *send* operation no later than the starting time of the second period. The rate monotonic algorithm on the communication medium ensures that the message will be delivered to the receiving processor by the end of the second period. It follows that the "update O_2" request is ready at the initiation time of the third period and can be carried out by the end of the third period.

4. Conclusions

Distributed real-time databases are becoming an increasingly important area of research with applications ranging from surveillance to manufacturing and production control. In this paper, we have integrated a modular concurrency control theory with a real-time scheduling protocol to create a real-time database concurrency control protocol, the setwise two phase lock with priority ceiling. We have shown that this integrated approach is free from deadlocks and bounds the blocking encountered by a task at each processor to at most one elementary transaction until it suspends itself or completes. This approach also supports the replication of data objects. We have also derived a bound for the lag between the versions of replicated data objects at different sites. Finally, we have provided a schedulability analysis for a set of periodic tasks with embedded transactions. This study also provides the interesting observation that the use of read and write locks can lead to worse performance in terms of schedulability than the use of exclusive locks.

[6]Readers who are interested in the scheduling issues of communication media are referred to [4].

References

[1] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notion of Consistency and Predicate Lock in a Database System.
CACM, Vol. 19, No 11 , Nov. 1976.

[2] Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2 , June, 1983.

[3] Kung, H. T. and C. H. Papadimitriou.
An Optimal Theory of Concurrency Control for Databases.
In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 116-126. ACM, 1979.

[4] Lehoczky, J. P. and Sha, L.
Performance of Real-Time Bus Scheduling Algorithms.
ACM Performance Evaluation Review, Special Issue Vol. 14, No. 1 , May, 1986.

[5] Lehoczky, J. P., Sha, L. and Ding, Y.
The Rate Monotonic Scheduling Algorithm --- Exact Charaterization and Average Case Behavior.
Technical Report, Department of Statistics, Carnegie-Mellon University, 1987.

[6] Lehoczky, J. P., Sha L and Strosnider, J.
Enhancing Aperiodic Responsiveness in A Hard Real-Time Environment.
IEEE Real-Time System Symposium , 1987.

[7] Liu, C. L. and Layland J. W.
Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment.
JACM 20 (1):46 - 61, 1973.

[8] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4 , December, 1983.

[9] Schwarz, P.
Transactions on Typed Objects.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1984.

[10] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
Technical Report (CMU-CS-87-181), Department of Computer Science, CMU , 1987.

[11] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Task Scheduling in Distributed Real-Time Systems.
Proceedings of IEEE Industrial Electronics Conference , 1987.

[12] Sha, L., Lehoczky, J. P. and Jensen E. D.
Modular Concurrency Control and Failure Recovery.
IEEE Transactions on Computers, Vol. 37, No. 2 , 1988.

[13] Sha, L., Rajkumar, R. and Lehoczky, J. P.
Concurrency Control for Real-Time Databases.
Technical Report, Department of Computer Science , 1988.