

A Transaction Manager Component for Cooperative Transaction Models

George T. Heineman

Columbia University
Department of Computer Science
500 W. 120th Street
New York, NY 10027
Tel: (212) 939-7085, Fax: 666-0140

Abstract

The database community has produced extensive research on the concurrency control problem in the context of traditional databases. However, this traditional model is not suitable for some applications, such as software development environments and CAD/CAM systems. What is needed is an extended transaction model better suited for these newer applications. Unfortunately, there is no consensus as to which of the almost dozens of extended transaction models is appropriate.

This paper posits that there is *no* single model which will be applicable to every application and seeks other ways to use existing database technology. Towards this end, we present PERN, a new transaction manager component which provides the flexible and tailorable support required by advanced database applications.

1 Introduction

Software development environments (SDEs) have evolved to meet the ever increasing complexity of managing the development of software systems. As SDE researchers focus their attention on large scale software development, we see that management issues become increas-

ingly prevalent [8]. The need to prevent corruption and incompatible access becomes an absolute necessity; quite naturally, database technology is used to store and manage the necessary information. However, we shall see that the traditional solutions from the database community are not well suited to SDEs; other domains such as CAD/CAM and design environments are similarly affected.

We briefly present the concurrency control problem and one of its traditional solutions; we outline an example that reveals the limitations of the solution. We propose an approach by which a database is used to store information, and a separate transaction manager component, PERN, is used to manage the access of the data. In this fashion, the concurrency control for a particular application can be specified. We describe the architecture in which this component will be integrated and show how the component can be tailored.

1.1 Concurrency Control

Concurrency control is the task of maintaining the correctness of a database when multiple programs (or users) access the database. Instead of having the individual programs contend with concurrency issues, the database management system (DBMS) itself should mon-

itor and control the programs. A database therefore must provide a correctness criterion and a policy for enforcing it. The first step to creating a policy is to have the DBMS group all database operations performed by a program into a *Transaction* [2]; it is assumed that each program (i.e., transaction) is correct. The second step is to fix a particular behavior that the *Scheduler* of the DBMS will enforce. It is the scheduler's job to receive data requests from the transactions and produce a correct schedule of execution for the transactions. Since multiple transactions can execute concurrently in the database, the schedule is an interleaving of the operations of the active transactions. Traditional schedulers guarantee four properties for transactions, often called the ACID properties.

1. **Atomicity:** All operations of a transaction must be treated as a single unit; all of the operations are executed, or none.
2. **Consistency:** A transaction takes the database from one consistent state to another.
3. **Isolation:** Each transaction executes as if it were the only transaction in the database.
4. **Durability:** The results of a transaction are never lost if the transaction terminates normally.

With no further information about the semantics of the transactions or the integrity constraints of the data, traditional databases have no way of providing a correctness criterion; instead, they resort to guaranteeing *Serializability*.

Serializability

Serializability is based on the fact that if each transaction is correct, then any serial schedule of the transactions is correct. Hence, to maintain correctness, the DBMS need only produce a schedule for a set of transactions that is equivalent to some serial execution. There are several concurrency control protocols that guarantee serializability including two-phase locking [2], timestamp ordering [9], and optimistic validation [6].

Serializability is not appropriate, however, for such domains as design engineering and

software development environments. As these domains sought to use database technology, they were hindered by several limitations of the ACID properties. The **I**solation property requires each transaction to operate independently of any other transaction; this prevents sharing among transactions. A subtler problem results from the **C**onsistency property, which requires each transaction to *individually* leave the database in a consistent state. This prevents applications from using concerted effort among several transactions which together are required to bring the database into a consistent state.

The traditional transaction model is best suited to domains in which little or no semantic information is known about the transactions. In software development environments, however, semantic information is known and formally encoded. If this information could be used by the concurrency control protocol of the database, great benefits could be realized.

Check-Out

Early attempts at concurrency control in the software engineering community resulted in the Check-Out model, best epitomized by such systems as RCS [10]. The check-out model has users make all data modifications to private versions in a personal workspace. Other users can continue to read old versions even while multiple users are updating private versions of the same data. Once the changes are complete, the user must manually integrate these modifications into the global store; this model is actually orthogonal to serializability. It relies on the assumption that two versions of the same data item can be merged in a consistent fashion. However, this model often hinders cooperation, since the users are unable to share information, but must operate on private versions of data.

1.2 Example

Researchers have attempted to use database technology to manage the data belonging to a large-scale software project. However, the traditional mechanism of enforcing serializable access has been found to be too restrictive.

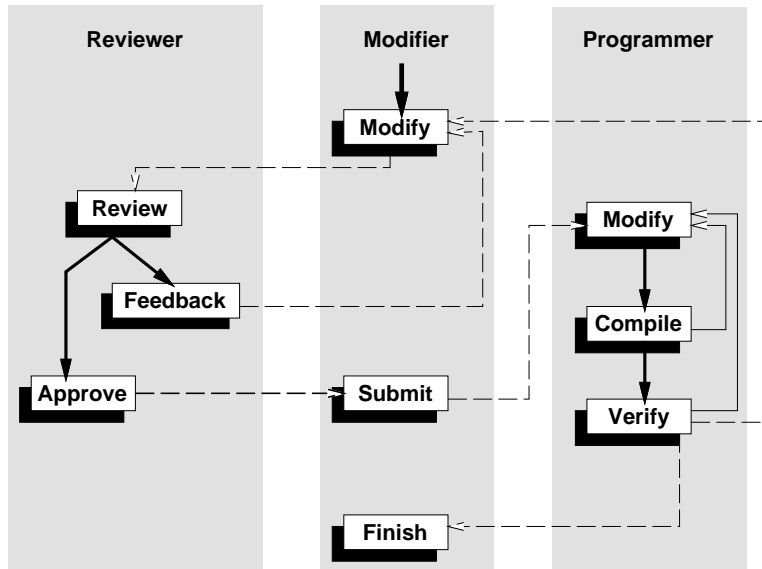


Figure 1: Sample Cooperative Process

Consider the scenario in Figure 1, extracted from the ISPW7 example problem [4]. There are three users, the Reviewer, the Designer, and the Programmer. They each have a set of tasks (in white boxes) that they must perform. The solid arrows define the sequence of tasks for an individual user, and the dashed arrows show how the users communicate with each other. The long grey vertical boxes represent the transactions encapsulating each user's actions. The work starts when the Designer submits a modified design for review. The Reviewer either approves the design or produces feedback and notifies the Designer, who either continues to modify the design or submits it to the Programmer. Once the Programmer has made the necessary modifications, the code is compiled and verified and the Designer is notified of either success or failure, in which case the design is finished, or modified, respectively.

In this example the two traditional means of handling concurrent access, namely serializability and check-out, will not work. Serializability requires that the schedule of execution be equivalent to a serial ordering of the transactions. As seen by the cyclical nature of this process, no serializable execution of the transactions will produce correct behavior. The check-out model is also inappropriate, since it

would be incorrect for the Reviewer to be reading an old version of the design that the Designer has modified in his private workspace. This scenario of collaborative work is a routine example of user-controlled interaction, a feature not well supported by database technology most suited for programmed batch-like transactions. Another limitation to standard database technology is the rigid and uniform way in which the system treats all transactions. If the database management system were extensible to allow tailorable access to its data, then an administrator could define the access patterns of transactions; this ability is called *tailoring the concurrency control policy*. Before presenting the architecture for PERN, we briefly highlight PERN's transaction model.

2 PERN Transaction Model

A transaction is composed of *data* and *management* operations. Data operations read or write a particular data item in the database. Management operations include *Begin*, *Commit* and *Abort*; each transaction is initiated by *Begin*. The *Commit* operation makes permanent all changes made by the encapsulating transaction, while the *Abort* operation rolls back all the effects of a transaction.

```

cooperation-rule
  selection-criterion:
    Under which conflicts is this CORD rule applicable
  local bindings:
    Gather information for the condition/action pairs
  body:
    ( condition-1, set-of-actions-1 )
    ( condition-2, set-of-actions-2 )
    ...
    ( condition-n, set-of-actions-n )

```

Figure 2: CORD syntax

PERN uses a locking protocol as a basis for determining consistency and allows the administrator to create and tailor a lock compatibility matrix. In this way, a conflict between locks signals a potentially inconsistent access. In response to the conflict, PERN first checks its *cooperation model* to determine if an administrator has foreseen this particular conflict scenario and provided a cooperation rule, written in CORD (see Figure 2), to resolve the conflict. If no such rule can be found, then PERN uses the default policy of the transaction manager and aborts the offending transaction. Otherwise, PERN issues a set of actions in response to the conflict. In this way, the correctness criterion of the database has been formally encoded, and the system is no longer limited to serializability.

PERN might force a data operation to block in order to maintain the consistency of the database; this is the *Suspend* management operation. PERN has a similar management operation, *Procrastinate*, which is used when the suspended transaction might wait a potentially indefinite length of time. In certain situations PERN might be ordered to preempt and abort a transaction to give another transaction priority. One can imagine a scheduler that preempts a transaction and, instead of aborting it, *modifies* it in some special way, to allow another transaction to continue. To do this we need management operations that treat transactions as divisible units.

Kaiser and Pu [5] present a model that restructures in-progress transactions as an approach to managing consistent concurrent access. The authors introduce two management operations for a transaction: *Split* and *Join*.

Split divides an ongoing transaction into two or more transactions that are serializable with respect to each active transaction (including each other), as if they had always been separate transactions. *Join* merges two or more transactions that are serializable with respect to each other into a single transaction, as if they had always been part of the same transaction. These operations are the cornerstone of the PERN transaction restructuring mechanism.

Locking is conventionally used to prevent certain dependencies [1] from forming between transactions. However, if PERN allows a particular locking conflict to exist, then dependencies are necessarily formed between the conflicting transactions. The management operation *Allow* is used to allow a conflict and create dependencies between transactions. There are several types of dependencies; here we list two:

1. **Commit-Dependency:** $T_i \rightsquigarrow T_j$ states that transaction T_i cannot commit until transaction T_j finishes (either commits or aborts). This does not imply that if transaction T_j aborts, then transaction T_i should abort.
2. **Abort-Dependency:** $T_i \rightarrow T_j$ states that if T_j aborts, then T_i must also abort. This implies neither that if transaction T_j commits, then transaction T_i should commit, nor that if transaction T_i aborts, then transaction T_j should abort.

The concept of an *Obligation* [7] is also part of the transaction model. A transaction T can acquire an obligation to fulfill some specific constraint C , at some future time. If T does not satisfy C , one could consider aborting T . However, obligations are intended to exist after the transaction to remind or enforce other transactions to satisfy C . Each transaction operates within a *Session* (see Section 3.1). We briefly mention here that each transaction operates within a session, and if a transaction completes without satisfying its obligation, the obligation is transferred to the session. Once a session has acquired the obligation, any transaction in the session may satisfy it, and the obligation is removed.

In PERN we introduce a concept analogous to an obligation called a *Restraint*. A restraint is,

in some sense, a negative obligation. A transaction can acquire a restraint in order to restrict its future activities. When a transaction commits, any restraints it acquired are transferred to the session in which the transaction is operating. A restraint can only be removed explicitly; this is notably different from obligations which can be silently removed as they are satisfied.

3 Architecture

We now present the design for a four-layered architecture as shown in Figure 3. We first demonstrate the need for this architecture and then describe the responsibilities of each of these layers. The layered approach is an abstraction for separating the various responsibilities of the system. The PERN Layer (PL) is responsible for all transaction management services. As a minimal requirement, PL assumes that the Data Management Layer (DML) provides an interface for accessing data items based upon a unique data item identifier (in object-oriented systems this is the object identifier). DML is an existing database management system. PL can query DML for arbitrary relationships maintained between data items. For example, if DML has composite data items (i.e., data items containing other items), PL might need to place intention locks on multiple data items to realize a particular locking protocol (consider multi-granularity locking [3]).

The PERN layer receives requests to start and end transactions. Whenever a transaction requests an access to a data item, PL acquires a lock for the specific access. PL requires that all accesses to a data item be preceded by a request (to PL) to access that item. As described, PL appears to require only one additional layer above it, so we still need to provide additional motivation for having four layers. PL assumes that it can acquire and use semantic information from a higher layer, called the Task Layer (TL). TL stores the description of the various user tasks, and how they are executed. In particular, it contains the mapping from tasks to transactions, which PL uses when it dynamically restructures transactions. There is, however, the need for a layer

above TL, which we call the Session Layer (SL). This layer is responsible for managing the various user-contexts of all users interacting with PERN. Each client issues user-commands to TL, and SL maintains the history of all TL interaction by each client. Such useful concepts as “Goals” might not be possible to encode in TL task definitions. For example, there might be no way to specify in advance what tasks need to be executed to achieve the goal of fixing a software defect. In addition, there are entities, such as obligations and restraints, which span task execution.

Administrator Interface

PERN is based on the idea that each layer has extensible features. As the architecture in Figure 3 shows, each of the Task, PERN, and Data layers are extensible. The layers can make use of administrator-provided extensions either on-line or off-line. DML allows the administrator to define the *data schema* for the data items which it stores. TL allows the administrator to provide information regarding the particular tasks. This ranges from simple parameterization to a full semantic encoding of the tasks. Finally, PL reads the *cooperation model* off-line, to get the cooperation rules (written in CORD) it will use. Once initialized, the PERN layer uses these rules as conflicts arise.

3.1 Session Layer

The Session Layer provides a context within which each user works. When a client application connects to the database server, a default session is created. There are primitives to create new sessions as needed, to end a session, and to split and join sessions. Sessions are persistent for each client, and clients can detach manually from a session (to be left in progress). If a client terminates abnormally, this is treated as if the client had detached from its session.

A session cannot end if it has an unfulfilled obligation; a client can still detach from this session, but it cannot be ended. Theoretically, one can consider rolling back a session with an unfulfilled obligation, but in practice this is extremely undesirable, as the session potentially represents a large amount of committed work.

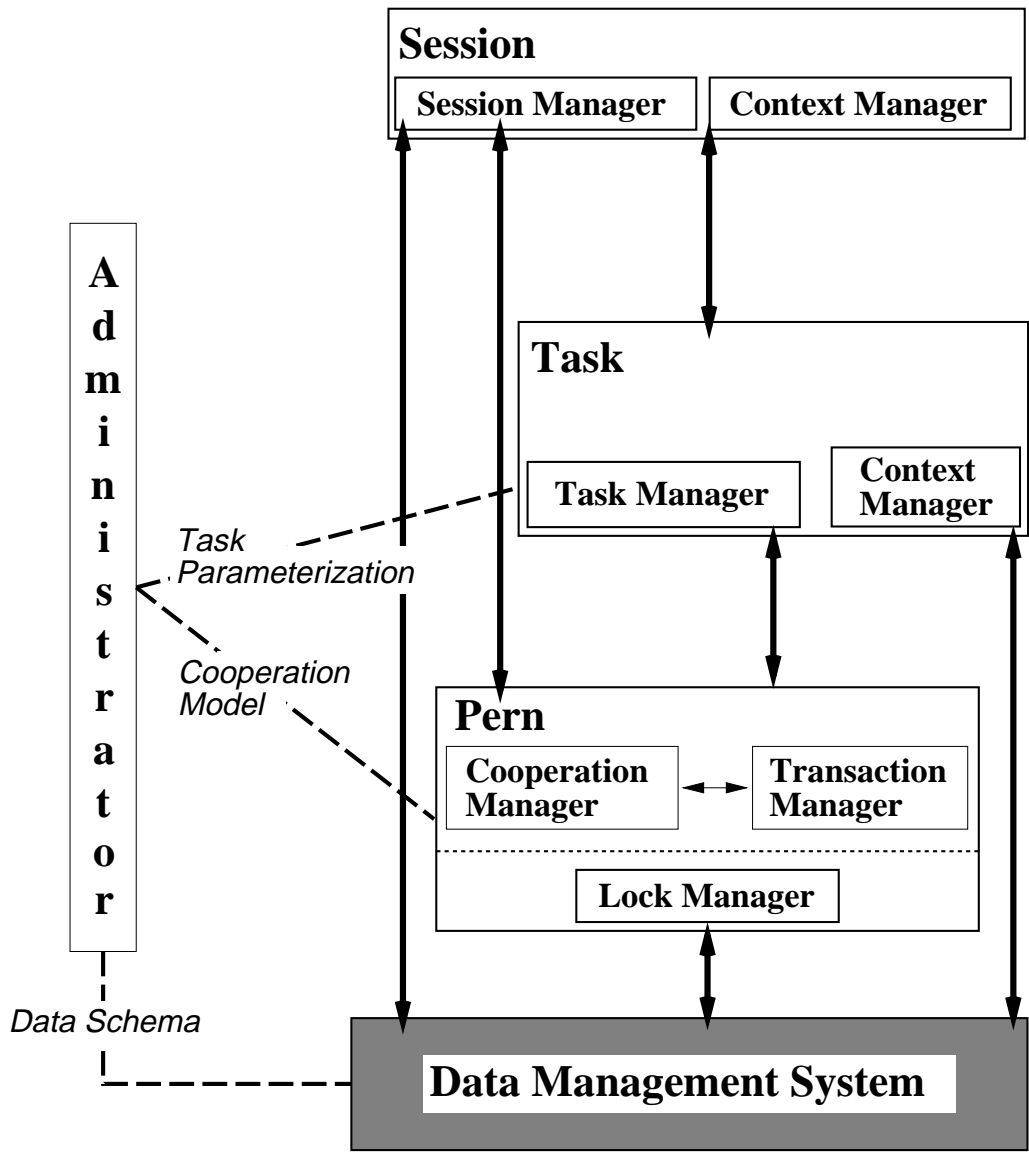


Figure 3: PERN Four-Layered Architecture

Instead, we consider the default case of splitting the session into two parts: the work associated with the unfulfilled obligation(s) and the independent work. We then end the session that has no problems and (possibly) roll back the session with obligations.

Sessions also manage restraints and obligations (see Section 2). Once associated with a session, a restraint can only be removed manually, while obligations can be removed silently by PL. There is an interface for PL (and TL) to notify SL that a restraint (or obligation) should be added to or removed from a session; SL maintains all sessions for each client accessing the database. Whenever a client communicates with SL, the appropriate session is made active by SL's context manager.

3.2 Task Layer

The Task Layer implements each user request by communicating directly with PL. TL has the responsibility of defining transaction boundaries and hiding the implementation details of fulfilling the user request. Any dependencies between transactions known in advance are set by TL. For example, if TL chooses to use a nested transaction model, it is responsible for assigning the appropriate static dependencies between parent and children transactions.

Each task performed by TL operates within a specific session, and TL can use the obligations and restraints of the session while processing the task. For example, TL could restrict certain tasks by checking to see if they would violate its session's restraints. Correspondingly, TL could suggest tasks that would satisfy any obligations maintained by the session.

TL specifically gives PL the freedom to restructure transactions according to a cooperation model. This implies that TL must alter its own data structures accordingly in response to PL's actions. Through dynamic restructuring, PERN will provide a more flexible treatment of user-controlled transactions. If TL is not resilient to restructuring notifications (from PL), then it cannot participate in advanced transaction restructuring.

To use the transaction abstraction appropriately, TL should not be concerned with the actual interleaved schedule produced at execution

time, so long as correctness is maintained; note that this correctness is no longer limited to serializability. A final example of having TL respond to PL actions occurs when PL decides to preempt and abort a transaction T_j to allow a more important transaction T_i to proceed. TL presumably has two tasks $Task_i$ and $Task_j$ that initiated these transactions, and while processing $Task_i$ it needs to update its own internal data structures to reflect the fact that the transaction for $Task_j$ has been aborted.

3.3 PERN Layer

The PERN Layer will provide a well-defined functional interface to TL. First, there are the standard *begin*, *commit*, and *abort* primitives for defining the boundaries of transactions. `commit(tid)` makes persistent all changes for transaction `tid` whereas `abort(tid)` restores the database to a state in which it would have been in had transaction `tid` not attempted to execute. The `abort()` and `commit()` actions are complicated by the presence of transaction restructuring; consider aborting a transaction that has split into several transactions.

All access to a data item must be preceded by a lock request to PL. The decision to employ a locking mechanism is central to PERN and does not preclude PERN's ability to implement such protocols as optimistic concurrency control. We introduce additional complexity into the locking protocol by separating the data from the lock manager; consider the case of composite data items. The lock manager needs to know the structure of the data to appropriately assign intention locks; PL assumes an interface to determine this information.

PL provides both dynamic and static support for TL transactions. TL can statically specify the interaction of a set of programmed transactions using PERN primitives. The dynamic nature of the transaction reconstruction occurs when PL detects a conflict and repairs it according to PERN's cooperation model; recall that the cooperation model is loaded into PL when it is initialized. PL notifies TL of any transaction restructuring it has performed.

PL considers any restraints and obligations associated with a transaction and its session when it is asked to commit a transaction. If

a transaction satisfies an obligation, PL notifies the session that the obligation has been removed. If a transaction violates a particular restraint, PL can invoke the cooperation manager to try to resolve the situation. This is different from the standard conflict situation we have been discussing throughout this paper; if the cooperation manager is unable to resolve this situation (the only way to do so would be to remove the restraint), the transaction must be aborted.

3.4 Implementation

As part of our research, we will implement a PERN component to be used in the four-layered architecture. PERN is intended to be used with a variety of DML and TL instances. We will have sample DML, TL, and SL instances, but PERN can be integrated into a system which already has its own TL and DML. The extent to which PERN succeeds in being integrated with arbitrary systems will be indicative of PERN's generality.

Even though PERN assumes the four-layered architecture, it is still able to function in the absence of either SL or TL. The PL is only able to provide cooperative transaction support if it has sufficient semantic information. In the absence of SL, for example, obligations must be limited to a specific task. In the absence of TL, PL is able to assign obligations, but is unable to perform any assistance for cooperation. Naturally, PERN is best suited for applications which can provide both SL and TL.

4 Conclusions

We are currently implementing PERN and designing a transaction model to define the semantics of the actions allowed by CORD. The model and further experimentation with the PERN component are currently being investigated as part of the author's dissertation.

References

- [1] P. K. Chrysanthis and K. Ramamritham. A unifying framework for transactions in competitive and cooperative environments. *IEEE Bulletin on Office and Knowledge Engineering*, 4(1):3-21, February 1991.
- [2] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624-632, November 1976.
- [3] J. Gray, R. Lorie, and G. Putzolu. Granularity of locks and degrees of consistency in a shared database. In *International Conference on Very Large Data Bases*, pages 428-451. Morgan Kaufmann, 1975.
- [4] Dennis Heimbigner and Marc Kellner. Software process example for ISPW-7. Anonymous ftp from ftp.cs.colorado.edu./pub/cs/techreports/ISPW7. 1991
- [5] Gail E. Kaiser and Calton Pu. Dynamic restructuring of transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 265-295, San Mateo CA, 1992. Morgan Kaufmann.
- [6] H. T. Kung and John Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213-226, June 1981.
- [7] Naftaly H. Minsky and Abe D. Lockman. Ensuring integrity by adding obligations to privileges. In *8th International Conference on Software Engineering*, pages 92-102, London, UK, August 1985.
- [8] Dewayne E. Perry and Gail E. Kaiser. Models of software development environments. In *10th International Conference on Software Engineering*, pages 60-68, Raffles City, Singapore, April 1988.
- [9] David P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3-23, February 1983.
- [10] Walter F. Tichy. RCS - a system for version control. *Software - Practice & Experience*, 15(7):637-654, July 1985.

About the Author

George Heineman is a PhD candidate in the Computer Science Department at Columbia University. His research interests are software development environments and programming support for large scale systems. He received his BA degree in computer science from Dartmouth College, and his MS degree from Columbia University. He is a member of IEEE and ACM. He can be reached at the Department of Computer Science, 450 CS Building, Columbia University, New York, NY 10027, email: heineman@cs.columbia.edu.