International Journal of Contemporary Research in Computer Science and Technology (IJCRCST) e-ISSN: 2395-5325
Volume1, Issue 5 (August '2015)

STUDY ON CONCURRENCY AND TRANSACTION CONTROLS

N.Poonguzhali,
M.Phil Research Scholar,
Department of Computer Science,
Siri PSG College of Arts and Science for Women,
Sankari, Tamilnadu, India.

K.Sumathi,
Assistant Professor,
Department of Computer Science,
Siri PSG College of Arts and Science for Women,
Sankari, Tamilnadu, India.

Abstract: In modern day's concurrency computing systems are discussed in two-fold areas, first, distributed system; concurrency is caused by the fact that the individual components are active. They evolve independently, and sometimes they communicate with each other in order to synchronize or to exchange data. Second, View all object-oriented systems as inherently concurrent, since objects themselves are "naturally concurrent" entities. In reality, concurrency adds a new dimension to system structure and design. Concurrent systems are extremely difficult to understand, design, analyze and modify. Transaction processing, concurrency control and recovery issues have played a major role in conventional databases, and hence have been an important area of research for many decades. However, with the increasing use of advanced database applications such as CAD/CAM, large software design projects, object-oriented databases both in centralized and distributed environments, there is a vital need for better algorithms for handling the new applications more efficiently. In this paper, we present a study on the concurrency control and transaction controls. We have discussed the work process in the area of transaction modeling, its applications in object-oriented and databases.

Keywords: Concurrency control, Transaction, Distributed System, Object-Oriented System

I.INTRODUCTION

In recent programming present features that allow a programmer to put across concurrency in an application by using active objects, i.e. objects with their own thread of control, and distribution. Concurrent systems can be classified into cooperative systems, where individual components collaborate, share results and work for a common goal, and competitive systems, where the individual components are not aware of each other and compete for shared resources. Programming languages address collaboration and competition by providing means for communication and synchronization among active objects [1]. The realization of complex object-oriented systems often needs sophisticated and elaborate concurrency features which may go beyond the traditional concurrency control associated with separate method calls. A transaction groups together a sequence of actions, and can therefore encapsulate complex behavior and embrace groups of objects and method calls. Transactions structure the dynamic system execution as opposed to the static structuring based on objects. Because of the ACID properties, transactions are able to hide the effects of concurrency and at the same time act as firewalls for errors, making them appropriate building blocks for structuring reliable distributed systems.

This paper discussed transaction models are reviewed and their suitability for concurrent programming languages. The analysis of existing models of multithreaded transactions shows that they either give too much freedom to threads and do not control their participation in transactions, or unnecessarily restrict the computational model by assuming that only one thread can enter a transaction. Hence, a significant part of this thesis is devoted to the establishment of a new transaction model named Open Multithreaded Transactions, providing features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also threads taking part in transactions. The model allows several threads to enter the same transaction in order to perform a joint activity. It provides a flexible way of manipulating threads executing inside a transaction by allowing them to be forked and terminated, but it restricts their behavior in order to guarantee correctness of transaction nesting and isolation among transactions. The open multithreaded transaction model incorporates disciplined exception handling adapted to nested transactions. It allows individual threads to perform forward error recovery by handling an abnormal situation locally, and promotes a defensive approach for developing transactional objects, so that errors are detected early and dealt with inside the transaction. If local handling fails, the transaction support applies backward error recovery and reverses the system to its "initial" state [1].

II. TRANSACTION MODELS:

Transactions [28] are a classic software structure for managing concurrent accesses to global data and for maintaining data consistency in the presence of failures. The notion of transaction has first been introduced in database systems in order to correctly handle concurrent updates of data and to provide fault tolerance with respect to hardware failures [2]. A transaction groups an arbitrary number of operations on data objects (from now on called transactional objects) together, making the whole appear indivisible as far as the application is concerned and with respect to other concurrent transactions. By using transactions, updates involving multiple transactional objects can be executed as if they happened in a sequential world. Complex systems often need more elaborate concurrency features than the ones offered by concurrent object-oriented programming languages. The existing single method approaches do not scale well, since they deal with each single operation separately [3]. There is a need for structuring units that encapsulate complex behavior and embrace groups of objects and method calls. These units should represent dynamic system execution as opposed to the static declaration of objects inside objects. System understanding, verification and modification are facilitated if program execution is recursively structured using such units. Examples of applications which require such structuring units are banking systems and e-commerce systems in general, computer supported cooperative work systems (CSCW systems), complex workflow systems, computer assisted design systems (CAD systems), control of modern production lines and cells, etc.

Another concern which makes it necessary to extend the single-object view of system structuring is provision of fault tolerance: in many situations one cannot guarantee that erroneous state is confined inside an object. In that case, the application programmer has to deal with very complex error containment domains consisting of several interconnected objects. An error in a server can for example affect several client objects. In order to continue program execution, it is not sufficient to recover only the server or a client. Correct error recovery must recover the system as a whole[4].

The transaction scheme relies on three standard operations: begin, commit and abort, which mark the boundaries of a transaction. After beginning a new transaction, all update operations on transactional objects are done on behalf of that transaction. At any time during the execution of the transaction it can abort, which means that the state of the system is restored to the state at the beginning of the transaction (also called roll back). Once a transaction has completed successfully (is committed), the effects become permanent and visible to the outside. This approach focuses on preserving and guaranteeing important properties of the data objects (sometimes called resources) accessed during a transaction. These properties are referred to as the ACID properties: Atomicity, Consistency, Isolation and Durability [4,5,6,7].

Atomicity

From the perspective of the caller of a transaction, the execution of the transaction appears to jump from the initial state to the result state, without any observable intermediate state —or, if the transaction cannot be completed for some reason, it appears as though it had never left the initial state. Atomicity is a general, unconditional property of transactions. It holds whether the transaction, the entire application, the operating system, or any other components function normally, function abnormally, or crash. For a transaction to be atomic, it must behave atomically to any outside observer. Under no circumstances may a transaction produce a result or a message that later disappears if the transaction rolls back. Atomicity is a vital property for proper system structuring and providing fault tolerance.

Consistency

A transaction produces consistent results only; otherwise it aborts. A result is consistent if the new state of the application fulfills all the validity constraints of the application according to the applications specification1. Unfortunately, this requirement is very hard or even impossible to verify. The state of an application tends to be very complex, and the number of possible consistency constraints among data items is huge. In order to still guarantee consistency, current transaction systems rely on the application programmer to only commit a transaction if the application state has been updated in a consistent way. A transaction must be written to preserve consistency. That is, each transaction expects a consistent state when it starts, and recreates that consistency after making its modifications, provided it runs to completion. Note that the intermediate states produced by a transaction during execution of its individual operations need not necessarily be consistent. The transaction system guarantees only that the execution of a transaction will not erroneously corrupt the application state.

Isolation

Multiple transactions may execute concurrently. The isolation property states that transactions that execute concurrently do not affect each other, and that the recovery of any of them is separated from the execution of the others. Therefore concurrent transactions produce the same results as if they had been executed sequentially in some order. This does not mean that transactions cannot share objects. It only implies that all modifications that a transaction has made to transactional objects during its execution cannot be based on data computed by a yet-to-be-committed transaction.

Durability

Durability requires that the results of a transaction having completed successfully remain available in the future. The system, once it has acknowledged the execution of a transaction, must be able to reestablish its results after any type of subsequent failure. It also implies that there is no automatic function for revoking a completed transaction. The only way to get rid of what a completed transaction had done is to execute another transaction with a counter algorithm.

III. CONCURRENCY CONTROL

Participants of an open multithreaded transaction collaborate loosely by accessing the same transactional objects. They are allowed to communicate directly, but this form of communication and synchronization is not supported by the model. Hence, concurrency control in open multithreaded transactions concentrates on the synchronization of accesses to transactional objects by participants: Dealing with *cooperative concurrency* means ensuring data *consistency* despite concurrent accesses to transactional objects made by participants of the same transaction. Handling *competitive concurrency* comes down to guaranteeing the *isolation* property for each transaction. Transactions running concurrently are not allowed to interfere with each other; participants of a transaction access transactional objects as if they were the only threads executing in the system. The isolation property guarantees that the abort of a transaction does not cause other transactions to abort. *Cascading aborts* are prevented[8,9,10].

Pessimistic Concurrency Control

The principle underlying pessimistic concurrency control schemes is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so. Typically, a concurrency control manager is associated with each transactional object. Before allowing a transaction to execute an operation on the object, the concurrency manager checks if the transaction performing that particular operation would create a conflict with any other uncommitted operation executed on the object on behalf of other transactions If a transaction invokes an operation that causes a conflict, the transaction is blocked or aborted. The duration of blocking and the number of times blocking or aborting occurs can be reduced by exploiting operation and object semantics.

Optimistic Concurrency Control

In optimistic concurrency control schemes [11], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, the transactions are validated to ensure that they preserve serializability. If a transaction is validated, it means that it has not executed operations that conflict with the operations of other concurrent transactions. It can then commit safely. A distinction can be made between optimistic concurrency control schemes based on *forward validation* or *backward validation*, depending on the manner in which conflicts are determined.

Forward validation checks to ensure that a committing transaction does not conflict with any still active transaction and, consequently, that the committing transaction's effects will not invalidate any active transaction's results.

Backward validation checks to ensure that a committing transaction has not been invalidated by the recent commit of another transaction.

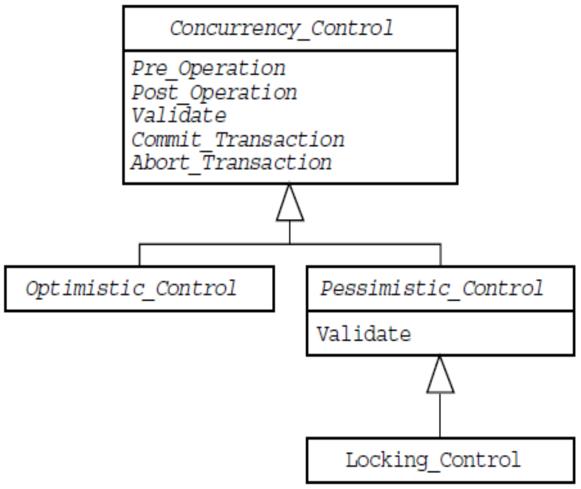

Figure 1.1: The Concurrency Control Hierarchy

4.1.4 Concurrency Control Information for Operations

In order to correctly handle cooperative concurrency, the concurrency manager must be able to determine for each operation of a transactional object if it is an *observer* or a *modifier*. To deal with competitive concurrency, optimistic and pessimistic concurrency control schemes must be able to decide if there are conflicts between operations invoked by different transactions that would compromise the serializability of these transactions. This information must be associated with each operation of a transactional object. The following sections introduce strict concurrency control and semantic-based concurrency control for operations[12].

Strict Concurrency Control

The simplest form of concurrency control among operations of a transactional object is *strict concurrency control*. In locking based concurrency control schemes this technique is also referred to as *read / write locking*.

It is simple, for strict concurrency control only distinguishes observer and modifier operations. Reading a value from a data structure does not modify its contents, writing a value to the data structure does. In this case, cooperative and competitive concurrency control is based on the same criteria.The compatibility table of read and writes operations are shown in table 1.1.

	Read(x)	Write(x)
Read(x)	Yes	No
Write(x)	No	No

Table 1.1: Compatibility Table of *Read* and *Write* Operations

Semantic-Based Concurrency Control

Inter-transaction concurrency can be increased if one knows more about the semantics of the operations of a transactional object. Exploiting this knowledge can drastically increase the performance of an application that uses transactions. According to [13], the concurrency semantics of a transactional object depend on the following:

• Semantics of the operations,
• Operation input and output values,
• Organization of the object, and
• Object usage.

Commutatively

Let's consider an abstract data type representing a set. A set is a non-ordered collection of elements without duplicates, meaning that for a given element there can only be one instance in the set at a given time. A set provides three operations, Insert (Set, Element) to insert an element into the set, Remove (Set, Element) to remove an element from a set, and Is In (Set, Element), an operation that tests if a certain element is part of a given set or not.

	Insert (y)	Remove(y)	Is_In(y)
Insert (x)	X Not Equal to Y	X Not Equal to Y	X Not Equal to Y
Remove(x)	X Not Equal to Y	X Not Equal to Y	X Not Equal to Y
Is_In(x)	X Not Equal to Y	X Not Equal to Y	Yes

Table 1.2: Backward Commutatively Table for the *Set*

Depending on the update strategy used for transactional objects, two slightly different forms of commutatively must be provided. Backward commutatively is used in combination with immediate update of data objects. In this scheme, each operation is immediately executed on the transactional object, possibly modifying its state see in table 1.2. The ordering in which two operations A and B are executed on a transactional object is important in this case, since the operation executed second "sees" the results of the execution of the first one. B commutes with A, if A followed by B has the same effects as executing A, then B and then undoing A, irrespective of the initial state of the transactional object. In particular, the return values of B must be the same in both cases.

Forward commutatively is used in combination with deferred update of data objects. In this scheme, each operation on a transactional object is executed on a separate copy of the state of the object. The ordering of the operations A and B is not important in this case, since they both "see" the same state of the object. B commutes with A, if B's return values do not dependent on the modifications that A applies to the state of the transactional object [14,15].

CONCLUSION

In this paper discussed about cooperative and competitive concurrency, the classic single threaded transaction model must be extended. An ideal model must allow multiple threads to be associated with the same transaction context, and still enforce the ACID properties. An analysis of existing transaction models has shown that they either give too much freedom to threads and do not control their participation in transactions, or unnecessarily restrict the computational model by assuming that only one thread can enter a transaction. This paper also discussed various concurrency control models.

REFERENCES

[1] "Open Multithreaded Transactions A Transaction Model for Concurrent Object-Oriented Programming" in proceeding of the Graduate Software Engineer ETH Born in Hofstetten-Flüh (SO) Lausanne, EPFL April 2001

[2] Badal, D. Z.: "Correctness of Concurrency Control and Implications for Distributed Databases", in Proceedings of the IEEE International Computer Software and Application Conference – COMPSAC 79, Chicago, USA, November 1979.

[3] Bernstein, P. A.; Goodman, N.: "Concurrency Control in Distributed Database Systems", ACM Computing Surveys **13**(2), June 1981, pp. 185 – 221.

[4] Briot, J.-P.; Guerraoui, R.; Lohr, K.-P.: "Concurrency and Distribution in Object-Oriented Programming", ACM Computing Surveys **30**(3), September 1998, pp. 291 – 329.

[5] Bernstein, P. A.; Hadzilacos, V.; Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.

[6] Birrell, A. D.; Nelson, B. J.: "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems **2**(1), 1984, pp. 39 – 59.

[7] Chrysanthis, P. K.; Ramamritham, K.: "ACTA. A Framework for Specifying and Reasoning about Transaction Structure and Behavior", SIGMOD Record (ACM Special Interest Group on Management of Data) **19**(2), June 1990, pp. 194 – 203.

[8] Cristian, F.: "Understanding Fault–Tolerant Distributed Systems", Communications of the ACM **34**(2), February 1991, pp. 56 – 78.

[9] Daynès, L.: "Extensible Transaction Management in PJava", in Proceedings of the First International Workshop on Persistence and Java, University of Glasgow, UK, September 1996.

[10] Menascé, D. A.; Nakanishi, T.: "Optimistic Versus Pessimistic Concurrency Control Mechanisms in Database Management Systems", Information Systems **7**(1), 1982, pp. 13 – 27.

[11] Papadimitriou, C. H.; Kanellakis, P. C.: "On Concurrency Control by Multiple Versions", ACM Transactions on Database Systems **9**(1), March 1984, pp. 89 – 99.

[12] Spector, A. Z.; Pausch, R. F.; Bruell, G.: "Camelot: A Flexible, Distributed Transaction Processing System", in Proceedings of the 33rd IEEE Computer Society International Conference (Spring COMPCON 88), pp. 432 – 437, San Francisco CA, USA, March 1988, IEEE Computer Society Press.

[13] Elmagarmid, A. K. (Ed.): Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1993.

[14] Eppinger, J. L.; Mummert, L. B.; Spector, A. Z.: Camelot and Avalon – A Distributed Transaction Facility. Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[15] Kung, H. T.; Robinson, J. T.: "On Optimistic Methods for Concurrency Control", ACM Transactions on Database Systems **6**(2), June 1981, pp. 213 – 226.