Looking Ahead in Open Multithreaded Transactions

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Faculté Informatique & Communications
Laboratoire de Génie Logiciel

Superviseur :
Prof. A. Strohmeier, LGL – EPFL

MASTERS THESIS 2003-2004

in collaboration with

McGill
UNIVERSITY

School of Computer Science
Software Engineering Laboratory

Responsible:
Prof. J. Kienzle, SEL – McGill University

Masters thesis realized by
Maxime Monod

Montréal, April 2004

Abstract & Acknowledgements

Complex distributed object-oriented systems often need complex concurrency features, which may go beyond the traditional concurrency control associated with separate method calls. A *transaction* groups together a sequence of operations, and can therefore encapsulate complex behavior and method calls. Transactions are able to hide the effects of concurrency and at the same time prevent the propagation of errors, making them appropriate building blocks for structuring reliable distributed systems. The o*pen multithreaded transaction* model, introduced by Prof. J. Kienzle, in his PhD thesis, 2001, provides 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 model also incorporates disciplined exception handling adapted to nested transactions.

When ending an open multithreaded transaction, synchronization on exit is needed to assure consistence and isolation of transactions. This limitation is described and a possible improvement is exposed, introducing *look-ahead threads*. Look-ahead threads are threads that speculate on the outcome of a transaction, leaving it to execute operations in advance.

This thesis proposes different models that allow looking-ahead in open multithreaded transactions, covering every concern, from typical read/write concurrency control problems to exception handling. Some models can even be used complementary, depending on the system context. Every model proposed is discussed in terms of implement complexity, execution speed, and exception handling granularity.

The research realized to write this Masters thesis was done in the Software Engineering Laboratories of the Swiss Federal Institute of Technology (EPFL) and of the School of Computer Science, McGill University.

I would like to thank Prof. A. Strohmeier for having supervised me at the EPFL, letting me go to Canada in order to obtain my Diploma.

I am grateful to Prof. J. Kienzle for having given me choices on different subjects to do my thesis on, for supporting me during these months in Montréal, giving me time for discussions and brainstorming, especially during our sessions on the blackboard, in my office. I also want to thank him for reading draft versions of my thesis, giving me the opportunity to improve its readability and for his never-ending motivation outburst.

I also want to thank Dr Alexander Romanovsky for his condensed and quick E-mail answers on the model proposals, especially for the exception handling part.

Maxime Monod

Résumé & Remerciements

Les systèmes orientés objet complexes ont besoin de mécanismes de contrôle de concurrence plus sophistiqués et plus évolués que ceux traditionnellement associés avec les appels de méthodes individuelles. Une transaction regroupe une séquence d’opérations et peut ainsi renfermer un comportement complexe et englober des groupes d’objets et d’appels aux méthodes. Les transactions permettent de cacher les problèmes liés à la concurrence et empêchent en même temps la propagation d’erreurs. Celles-ci forment ainsi des structures appropriées à l’élaboration de systèmes répartis fiables. Le modèle de transactions multitâches ouvertes (o*pen multithreaded transactions*), introduit par le Prof. J. Kienzle, dans sa thèse, en 2001, offre non seulement les mécanismes habituels qui permettent de contrôler et de structurer l’accès aux objets, mais aussi la possibilité de superviser les tâches qui participent à la transaction. Plusieurs tâches ont le droit de pénétrer dans une même transaction pour travailler en commun. Ce modèle autorise également la création de nouvelles tâches de même que leur destruction à l’intérieur ou à l’extérieur d’une transaction. Ce comportement est limité à certains endroits pour obtenir une imbrication correcte et pour garantir l’isolation entre les transactions. Le modèle intègre aussi un traitement d’exceptions structuré.

Lors de la terminaison d’une transaction multitâche ouverte, les tâches participantes sont synchronisées pour assurer la consistance et l’isolation des transactions. Cette règle limite les performances générales du modèle et peut être modifiée en introduisant des tâches prédictives (*look-ahead threads*). Ces tâches supposent que la transaction dans laquelle elles travaillaient va arriver à terme sans problème et exécutent en avance les prochaines instructions disponibles.

Ce projet de diplôme propose des modèles autorisant les tâches prédictives dans les transactions multitâches ouvertes, couvrant chaque sujet, allant des problèmes typiques de dépendances lecture/écriture au traitement d’exceptions. Certains de ces modèles peuvent être utilisés de façon complémentaire, dépendamment du système. Chaque modèle est analysé quant à la complexité de son implémentation, sa rapidité d’exécution et la granularité des exceptions traitées.

La recherche faite pour écrire cette thèse a été faite dans les laboratoires de Génie Logiciel de l’Ecole Polytechnique Fédérale de Lausanne (EPFL) et du département d’informatique – *School of Computer Science* – de l’Université McGill, à Montréal.

Je tiens à remercier le Prof. A. Strohmeier pour avoir supervisé mon travail et m’avoir laissé la possibilité de partir au Canada afin d’obtenir mon diplôme.

J’aimerais aussi remercier le Prof. J. Kienzle pour m’avoir laissé plusieurs sujets à choix, pour m’avoir appuyé durant ces mois à Montréal, me donnant du temps pour des discussions et des séances de « brainstorming », surtout pendant nos sessions sur le tableau noir de mon bureau. Je le remercie aussi pour avoir lu des versions non définitives de cette thèse, me laissant l’opportunité d’améliorer sa facilité de lecture, mais aussi et surtout pour son enthousiasme et sa motivation toujours débordante.

J’aimerais aussi remercier le Dr Alexander Romanovsky pour ses réponses E-mail rapides et condensées sur les propositions de modèles, tout particulièrement sur le traitement des exceptions.

Maxime Monod

Table of Contents

Abstract & Acknowledgements .... **i**

Résumé & Remerciements .... **iii**

Chapter 1 : Introduction **1**
1.1 Context & Objectives 1
1.2 Thesis Organization 2
1.3 Transactions 2

PART I OPEN MULTITHREADED TRANSACTIONS **5**

Chapter 2 : Main Concepts **7**
2.1 Open Multithreaded Transactions 7
2.2 Starting an Open Multithreaded Transaction 8
2.3 Joining an Open Multithreaded Transaction 8
2.4 Ending an Open Multithreaded Transaction 8
2.5 Concurrency Control in Open Multithreaded Transactions 9
2.6 Exception Handling 17

Chapter 3 : Limitation & Possible Improvement **21**
3.1 Look-Ahead in Atomic Actions 21
3.2 Objectives 23
3.3 Requirements Notes 24
3.4 Look-Ahead Thread in Open Multithreaded Transactions 25

Chapter 4 : Potential Issues **29**
4.1 Spawning Threads 29
4.2 Read/Write Dependencies 29
4.3 Multiple Objects Access - Deadlock Analysis 31
4.4 Nesting 32
4.5 Short Look-Aheads & Amount of Look-Aheads 32
4.6 Exception Handling 36

Chapter 5 : Look-Ahead & Concurrency Controls **39**
5.1 Pessimistic 39
5.2 Optimistic 41
5.3 Discussion 44

PART II PROPOSED MODELS **45**

Chapter 6 : Leave **49**
6.1 Concept 49
6.2 Concurrency Control Analysis 50
6.3 Exception Handling 51
6.4 Specific Limitations 52
6.5 Discussion 53
6.6 Variations 53

Chapter 7 : Stretch **55**
7.1 Concept 55
7.2 Concurrency Control Analysis 57
7.3 Exception Handling 58
7.4 Specific Limitations 60
7.5 Discussion 60

7.6 Variations 61

Chapter 8 : Create 63
8.1 Concept 63
8.2 New statements 64
8.3 Starting a Look-Ahead Transaction 64
8.4 Joining an Open Multithreaded Transaction 64
8.5 Ending a Look-Ahead Transaction 68
8.6 Notes on Spawning Threads 68
8.7 Concurrency Control Analysis 68
8.8 Exception Handling 69
8.9 Nesting 70
8.10 Specific Limitations 70
8.11 Discussion 70
8.12 Variations 70

Chapter 9 : Implicit 73
9.1 Concept 73
9.2 New Statements 74
9.3 Concurrency Control Analysis 76
9.4 Exception Handling 76
9.5 Nesting 76
9.6 Specific Limitations 77
9.7 Discussion 77
9.8 Variations 77

PART III BEYOND 79

Chapter 10 : Future Improvements 81
10.1 Cooperative Concurrency 81
10.2 Nested Transactions: Concurrency Control 81
10.3 Looking-Ahead Dedicated Concurrency Control 82
10.4 Asynchronous Handling 82

Chapter 11 : Conclusion 83

PART IV ANNEXES 85

Annex A : Bibliography 87

Annex B : Index 89

List of Figures

PART I OPEN MULTITHREADED TRANSACTIONS................................ 5

Chapter 2 : Main Concepts.. 7
Figure 2.1 : An Open Multithreaded Transaction 9
Figure 2.2 : Locked-Based Protocol in Open Multithreaded Transactions....................................... 11
Figure 2.3 : Deadlock Situation in Open Multithreaded Transactions... 11
Figure 2.4 : Timestamp Ordering in Open Multithreaded Transactions 12
Figure 2.5 : Multiversion Timestamping 13
Figure 2.6 : Optimistic Concurrency Control Situation 13
Figure 2.7 : Forward Validation 14
Figure 2.8 : Account and User Classes 15
Figure 2.9 : Exceptions in Open Multithreaded Transactions.. 18

Chapter 3 : Limitation & Possible Improvement 21
Figure 3.1 : An Atomic Action with Exception Handling 22
Figure 3.2 : Looking-Ahead in Atomic Actions 23
Figure 3.3 : Execution of Open Multithreaded Transactions in the Initial Model 26
Figure 3.4 : Looking-Ahead: Transactions & Lone Code.. 26

Chapter 4 : Potential Issues 29
Figure 4.1 : General Read/Write Dependency Model.. 30
Figure 4.2 : General Read/Write Dependency Model (Simple View) 30
Figure 4.3 : Simple RAR (Read-After-Read) Dependency Model 30
Figure 4.4 : Simple WAR (Write-After-Read) Dependency Model 30
Figure 4.5 : Simple RAW (Read-After-Write) Dependency Model 31
Figure 4.6 : Simple WAW (Write-After-Write) Dependency Model 31
Figure 4.7 : General Deadlock Dependency Model.. 31
Figure 4.8 : Model of a Short Look-Ahead Transaction.. 32
Figure 4.9 : Short Look-Ahead Lone Code.. 33
Figure 4.10 : Speculative Commit Statements in Short Look-Aheads... 34
Figure 4.11 : Last Former Transaction Non-Blocking.. 35
Figure 4.12 : Last Former Transaction Blocking.. 35
Figure 4.13 : Root Former Transaction Blocking... 35
Figure 4.14 : Exception Raised in Root Former Transaction... 37
Figure 4.15 : Exception Raised in Look-Ahead Context 37

Chapter 5 : Look-Ahead & Concurrency Controls 39
Figure 5.1 : Solution to WAR Dependency in Pessimistic Concurrency Control............................ 40
Figure 5.2 : Solution to RAW Dependency in Pessimistic Concurrency Control............................ 40
Figure 5.3 : Solution to WAW Dependency in Pessimistic Concurrency Control 41
Figure 5.4 : Solution to Read/Write Dependencies in Locking Protocol.. 41
Figure 5.5 : Solution to Deadlock in Pessimistic Concurrency Control 41
Figure 5.6 : WAR Dependency in Optimistic Concurrency Control 42
Figure 5.7 : RAW Dependency in Optimistic Concurrency Control 42
Figure 5.8 : WAW Dependency in Optimistic Concurrency Control 43

PART II PROPOSED MODELS...45

Chapter 6 : Leave 49
Figure 6.1 : Illustration of the "Leave" Model... 50
Figure 6.2 : Exception Handling in the "Leave" Model... 51
Figure 6.3 : Exception Handling in the “Leave” Model, with Nesting 52

Chapter 7 : Stretch 55

Figure 7.1 : Illustration of the "Stretch" Model 56
Figure 7.2 : Terminating Thread is Postponed in "Stretch" Model 56
Figure 7.3 : Cooperative Concurrency in the "Stretch" Model 57
Figure 7.4 : Cooperative Concurrency in Initial Model 58
Figure 7.5 : Exception Handling in the "Stretch" Model 59
Figure 7.6 : Exception Handling in the “Stretch” Model, with Nesting 59

Chapter 8 : Create 63
Figure 8.1 : Example of a Re-Joining Problem 65
Figure 8.2 : Self-Restrictive Join 66
Figure 8.3 : One-Way Permissive Joining Situation 67
Figure 8.4 : Two-Way Permissive Situation 67
Figure 8.5 : Retroactive Cascading Aborts 68
Figure 8.6 : Exception Handling in the "Create" Model 69

Chapter 9 : Implicit 73
Figure 9.1 : Illustration of the "Implicit" Model 74
Figure 9.2 : Terminating Threads is Postponed in "Implicit" Model 75
Figure 9.3 : Exception Handling in the "Implicit" Model 76

Chapter 1 : Introduction

1.1 *Context & Objectives*

Modern programming languages provide features that allow a programmer to express concurrency in an application by supporting active objects: objects with their own thread of control. Concurrent systems can be classified into *cooperative systems*, where individual components collaborate, share results and work for a joint goal, and *competitive systems*, where 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.

Complex object-oriented systems often need more sophisticated concurrency features. *Transactions* are program structures, which encapsulate groups of objects and of method calls, representing the dynamic execution as opposed to the static declaration of objects inside objects.

The *Open Multithreaded Transaction* model [Kie04], intended to be used in concurrent programming languages, provides 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 executions inside a transaction by allowing them to be forked and terminated, but it restricts their behavior when necessary in order to guarantee correctness of transaction nesting and isolation among transactions.

The goal of this Diploma Thesis is to relax the synchronization rules of the initial model, in order to maximize the overall performance of the system. The proposed approach has not been implemented yet, but the proposed models respect the design choices made in the *OPTIMA Framework* [Kie04]. Several models are proposed that improve the overall execution speed of the transactions allowing threads to *look-ahead* from a transaction and continue their execution instead of waiting for the other transaction participants. The entire theoretical basis to understand how transactions and particularly open multithreaded transactions work is explained in the following sections. The complete reference is [Kie04].

1.2 *Thesis Organization*

The thesis is split in three main parts. Part I describes the *open multithreaded transactions* model. Chapter 2 introduces the main concepts while chapter 3 covers one of the model limitations (synchronization on exit) and gives the objectives of a possible solution to overcome it: *looking-ahead*. In chapter 4, potential issues introducing looking-ahead theory are described. Chapter 5 describes problems that occur in concurrency controls, when looking-ahead, and provides solutions for the potential issues.

Part II exposes models introducing look-ahead threads in open multithreaded transactions. Chapters 6-9 describe each model, with their respective capabilities and limitations.

Part III is a critical view of the initial model and the models proposed, giving new guidelines for future research, design and implementation. Chapter 10 gives a list of the future improvements that could be added to open multithreaded transactions and the look-ahead models. Finally, chapter 11 gives a conclusion, summarizing the main results of this work.

Part IV contains the bibliography, with all references cited in this work, and an index.

1.3 *Transactions*

Some applications require concurrent access to distributed information shared amongst multiple components. Such applications must maintain consistency of data when trying to access multiple objects; or when multiple components try to access a single shared data.

The concept of a transaction has been invented to simplify the design of such applications by adding a new level of abstraction: transactions assure the integrity of data, being an indivisible unit of work.

A transaction groups several instructions together. The program structure relies on three standard statements: `begin`, `commit` and `abort`, defining the borders of a transaction.

Before the transaction begins, the system is in an initial state that is stable and coherent. The `begin` and `commit` statements are encapsulating the operations we want to consider as atomic. A transaction can `abort`, which results in restoring the initial state of the system (this is also called *rollback*). If the transaction succeeds and commits, all changes in the system state due to the encapsulated operations become durable and persistent. The requirements that transactions must provide are called the *ACID properties*.

1.3.1 ACID properties

Atomicity

Atomicity requires that either all the operations of a transaction are executed or none of them. The operations inside the transaction usually form a logical group. If for some reason, the system stopped after only some of them, the overall intent of the transaction would be erroneous.

Consistency

A transaction must preserve the consistent states of data, meaning that from an initial consistent state, the transaction transforms it into another consistent state of data. If, for some reason, there is a failure during the execution of a transaction, it gets aborted and thanks to atomicity, the system remains at its initial consistent state.

Isolation

Transactions can be executed concurrently, and therefore compete for resources. The isolation property assures that a single transaction behaves as if it was the only one running at this time. Multiple transactions running at the same time cannot interfere with each other and the outcome of one cannot change the behavior of any other one. The changes the transaction is doing are visible to the outside only after its commit has been done.

Durability

This property assures that once the transaction has succeeded, whatever failure or crash could have happened, the outcome of the transaction is durable and visible to the outside. This property is also called *persistence*.

Serializability

The concurrency management that allows multiple transactions to run concurrently is called *concurrency control*. It must assure that the concurrent execution of the transactions results in the same outcome as a serial execution. This is a result of the atomicity and isolation rules.

After having introduced the general idea behind transactions, the next chapter exposes the main concepts of the open multithreaded transactions.

Part I

Open Multithreaded Transactions

Chapter 2 : Main Concepts

We expose the general concepts behind *Open Multithreaded Transactions* [Kie04] in the current chapter. Only considerations on rules that we could change or transgress are exposed. Particular features or rules related specifically to naming or closing an open multithreaded transaction are not described, please refer to [Kie04] for information on these subjects.

Once an open multithreaded transaction is created (section 2.2), several threads can join it (section 2.3). The transaction then completes to its end (section 2.4) or abort in case of an exception (section 2.6). In order to collaborate, threads inside a transaction share access to *transactional objects*, which are also available from other transactions. In order to guarantee the isolation property, different concurrency control approaches can be applied (section 2.5).

2.1 *Open Multithreaded Transactions*

The open multithreaded transactions model allows tasks to be created, to run to completion, or to join an ongoing transaction at any time. There are only two elementary rules that restrict thread behavior:

- A thread created outside of an open multithreaded transaction is not allowed to terminate inside the transaction.
- A thread created inside an open multithreaded transaction must also terminate inside the transaction.

Threads working on behalf of an open multithreaded transaction are referred to as *participants*. External threads that create or join a transaction are called *joined participants*; a thread created inside a transaction by some other participant is called a *spawned participant*.

2.2 *Starting an Open Multithreaded Transaction*

- Any thread can start a transaction. This thread is the first joined participant of the transaction.
- Transactions can be nested. A participant of a transaction that starts a new transaction creates a nested transaction. Sibling transactions created by different participants execute concurrently.

2.3 *Joining an Open Multithreaded Transaction*

- A thread can join an open transaction, thus becoming one of its joined participants.
- A thread can join a top-level transaction if and only if it does not already participate in any other transaction. To join a nested transaction, a thread must be a participant of the parent transaction. A thread can participate in only one sibling transaction at a time.
- A thread spawned by a participant automatically becomes a spawned participant of the innermost transaction in which the spawning thread participates. A spawned participant can join a nested transaction, in which case it becomes a joined participant of the nested transaction.

2.4 *Ending an Open Multithreaded Transaction*

- All participants finish their work inside a transaction by voting on the transaction outcome. Possible votes are *commit* or *abort*. Voting abort is done raising an external exception (see section 2.6, page 17).
- The transaction commits if and only if all participants vote commit. In that case, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If any participant votes *abort*, the transaction aborts. In that case, all changes made to transactional objects on behalf of the transaction are undone.
- Once a spawned participant has given its vote, it *terminates* immediately.
- Joined participants are not allowed to leave a transaction, **i.e. they are blocked, until the outcome of the transaction has been determined**. This means in particular that all joined participants of a committing transaction **exit synchronously**. At the same time, but only then, the changes made to transactional objects on behalf of the transaction are made visible to the outside world. If a transaction is aborted, the joined participants may exit asynchronously, but changes made to transactional objects on behalf of the transaction are undone.
- If a participating thread “disappears” from a transaction without voting on its outcome, the transaction is aborted, as this case is treated as an error.

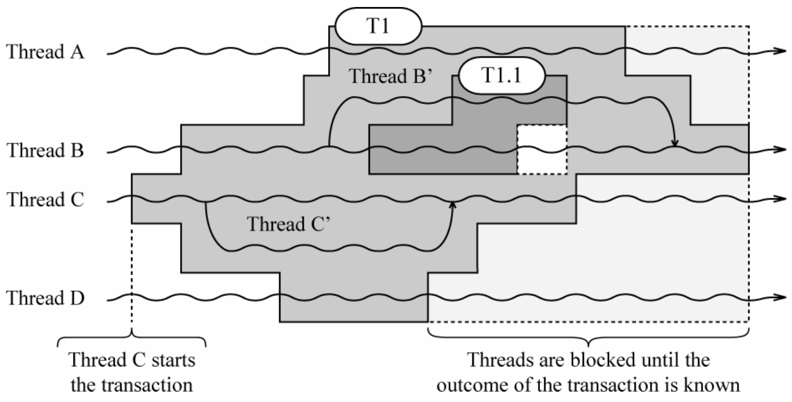

T1
T1.1
Thread A
Thread B'
Thread B
Thread C
Thread C'
Thread D
Thread C starts the transaction
Threads are blocked until the outcome of the transaction is known

Figure 2.1 : An Open Multithreaded Transaction[1]

Figure 2.1 shows two open multithreaded transactions[2]: T1 and T1.1. Thread C creates the transaction T1, threads A, B and D join it. Threads A, B, C and D are therefore *joined participants* of the transaction T1. Inside T1 thread C forks a new thread C' (a *spawned participant*), which performs some work inside the transaction and then terminates. Thread B also forks a new thread, thread B'. B and B' perform a *nested transaction* T1.1 inside of T1. B' is a spawned participant of T1, but a joined participant of T1.1. In this example, all participants of T1 vote commit. The joined participants A, C, and D are therefore *blocked* until the last participant, here thread B, has finished its work and given its vote. It is only after thread B has voted commit that the changes made to transactional objects are made persistent and that new operations can be executed by the threads.

2.5 *Concurrency Control in Open Multithreaded Transactions*

Concurrency control is needed to enforce the isolation rule of the ACID properties, assuring both cooperative and competitive handling. Its main focus is to maintain transactions isolated from each other (competitive) while letting threads inside a transaction work together loosely (cooperative). The following rules apply to open multithreaded transactions.

- Accesses to transactional objects by participants working on behalf of an open multithreaded transaction are isolated from accesses by other transactions. However, participants are allowed to make the identity of the transaction visible to the outside world. This identity can be used by threads willing to join the transaction.
- Accesses to transactional objects by participants of a child transaction are isolated from accesses by participants of the parent transaction.
- Inside a given transaction, classic consistency techniques, i.e. mutual exclusion, are used to guarantee consistency of transactional objects when accessed by several participants of the same open multithreaded transaction.

[1] Sibling transactions always appear in the same grey tone, whereas nested transactions appear darker. A blocked thread appears on a white background with a dashed outline.
[2] The dot notation always describes nested transactions (i.e. T1.1.1 is nested in T1.1, which is nested in T1).

As we are not dealing with cooperative concurrency (accesses to transactional objects made by participants of the same transaction), we focus on competitive concurrency control, guaranteeing the isolation property for each transaction. Concurrency control is commonly divided into two broad categories: pessimistic (*conservative*) and optimistic (*aggressive*). These two categories are respectively explained in section 2.5.1 and section 2.5.2, while we state what we consider as *conflicting operations* in section 2.5.3. This overview of concurrency control is very useful when analyzing the complexity of look-ahead models later on.

2.5.1 Pessimistic Concurrency Control

The fundamental concept of a pessimistic concurrency control is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so. The manager verifies that execution of operations is correct before allowing them to occur. If an operation of a transaction on the calling thread conflicts with any uncommitted operation of a different transaction, then it is blocked, or the invoking transaction is aborted.

Finding conflicts between operations can be adjusted by exploiting operation and objects semantics, which is discussed in section 2.5.3, page 15.

Lock-Based Protocols

Lock-based protocols use *locks* to implement permissions to perform operations. When trying to access a transactional object, the transaction must first get the associated lock from the concurrency manager of the transactional object. Before granting the lock, the concurrency manager must verify that this new lock does not conflict with any other lock held by other transactions in progress. If the concurrency manager determines that there would be a conflict, the thread requesting the lock is blocked[1], waiting for the release of the conflicting lock. Otherwise, the lock is granted, and the thread may proceed and execute the operation.

The order in which locks are granted to transactions imposes ordering on the transaction with respect to their conflicting operations. Two-phase locking [EGLT76] ensures serializability by not allowing transactions to acquire any lock after a lock has been released. This implies in practice that a transaction acquires locks during its execution phase (1st phase), and releases them at the end, once the outcome of the transaction has been determined (2nd phase). This mechanism is shown in Figure 2.2.

[1] In some situations, it is better to abort and restart the conflicting transaction rather than blocking it, as described in [BHG87].

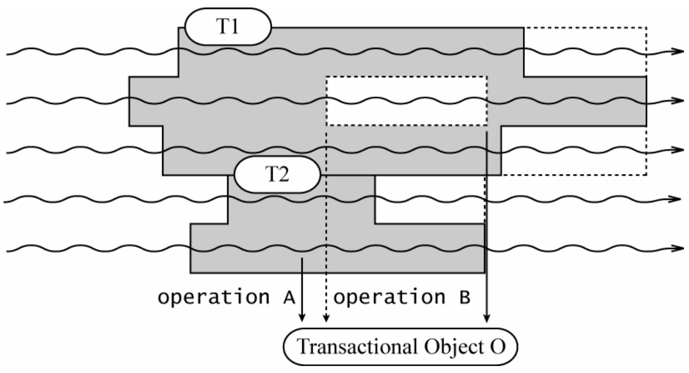


Figure 2.2 : Locked-Based Protocol in Open Multithreaded Transactions

Transaction T2, when executing operation A acquires a lock on object O that prevents T1 to invoke operation B on O[1]. Operation B has to wait until T2 releases the lock, when it commits.

When using blocking pessimistic concurrency control, deadlocks are possible. Two transactions T1 and T2 trying to acquire locks on two objects O and P can deadlock, if T2 first requests O and T1 first requests P. Now T1 is waiting for O, and T2 is waiting for P. This situation is shown in Figure 2.3. Such deadlocks can be detected and remedied by aborting one of the blocking transactions. A simple deadlock detection mechanism is waiting for a time-out, which aborts a transaction waiting for too long for a lock by guessing that it may be involved in a deadlock. Other deadlock detection mechanisms, like cycles in a wait-for graph or timestamp-based deadlock prevention, can be found in [BHG87].

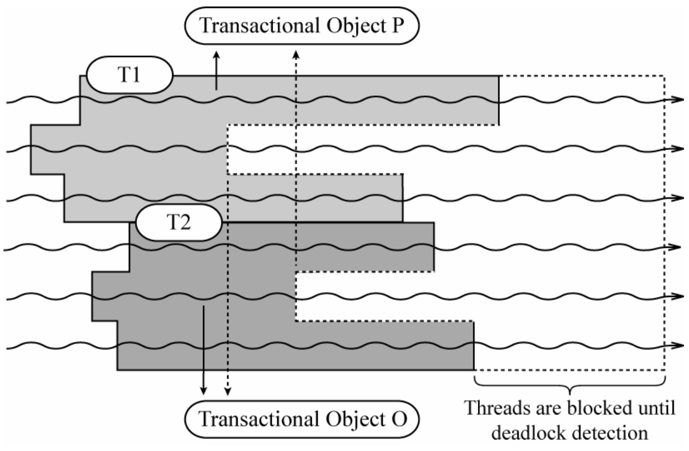


Figure 2.3 : Deadlock Situation in Open Multithreaded Transactions

Timestamp Ordering

Timestamp ordering is a pessimistic concurrency control that does not use locks. The basic idea is to generate a unique timestamp for each transaction, and associate it with all operations the transaction invokes. The concurrency control manager, also called *timestamp ordering scheduler*, ensures that conflicting operations are executed based on their timestamps. The concurrency control manager immediately schedules operations that arrive for execution unless some other

[1] We assume that in every figure, operations are conflicting.

conflicting operation with an earlier timestamp has already been invoked. In the latter case, the transaction invoking the conflicting operation is aborted. The timestamp of a transaction T1 is noted ts(T1). This situation is shown in Figure 2.4[1].

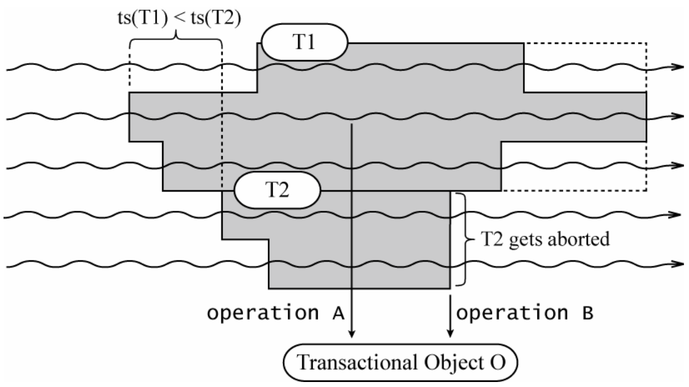


Figure 2.4 : Timestamp Ordering in Open Multithreaded Transactions

When T1 invokes A on O, it is immediately scheduled. When T2 tries to invoke a conflicting operation on the same object, permission has already been given to T1, thus T2 is aborted.

Additional Algorithms

Modifications to two-phase locking and timestamp ordering algorithms, which support multiple versions of objects, are discussed in [Sil81, BGH87]. **Multiversion Timestamping** [Ree83, BG83, BGH87] can be transposed from atomic actions to transactions associating a timestamp to every transaction, and invoking operations. When the scheduler processes an *observer*[2] operation with timestamp ts(T2), the version with the largest timestamp less or equal to ts(T2) is accessed. In case of a *modifier*[3] operation, a new version of the object – associated with the invoking transaction timestamp – is created. A modifier operation in transaction T2 is rejected if the scheduler has already processed an observer operation in transaction T3 of a version created by T1, assuming ts(T1) < ts(T2) < ts(T3). The situation is shown in Figure 2.5. T1 executes a modifier operation on the transactional object O, thus creating a new version O_1. Then T3 accesses the object O with an observer operation, the value read is the newest version less or equal to its own timestamp: O_1, because ts(T1) < ts(T3). When T2 tries to modify the object O, its action is rejected, because it would invalidate the last observer operation made.

In general, multi-versioning can be considered as an optimistic scheme in the pessimistic concurrency control: we are hoping that conflicting operations take place in the same order as the transaction were created – time stamped conflicting operation are not executed out of order.

[1] We assume that the approach of handling messages between transactions is pre-emptive – the transaction gets interrupted when receiving the abort message. Pre-emption is explained in section 2.6, page 17.

[2] An observer is an operation that reads the state of an object without modifying it.

[3] A modifier operation is an operation that updates the state of an object depending on its previous state (i.e. incrementing a counter) or that simply overwrites it.

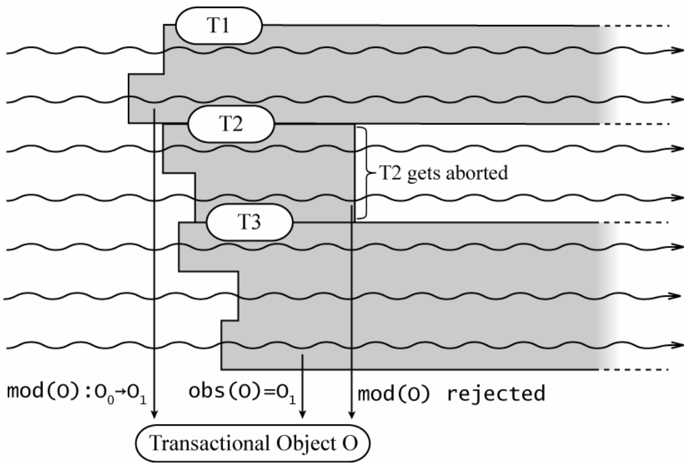


Figure 2.5 : Multiversion Timestamping

If for any reason, T1 wants to modify again the value of O later on, it gets aborted because of the observer in T3. Aborting T1 results in aborting T3 – otherwise T3 would have read an intermediate inconsistent value of O. Such *cascading aborts* problems can be avoided using **Time Warp Protocols** such as proposed in [JM86], introducing a notion of virtual time based on the transactions timestamp. The principle is that every operation inside a transaction is undoable by executing its inverse and that in case of a conflict, operations are rolled back until a consistent virtual time is found, where every action can take place without conflict – the idea is to then to try executing every conflicting operation in the timestamp order.

2.5.2 Optimistic Concurrency Control

In optimistic concurrency control [KR81, Her90], transactions are allowed to execute concurrently without checks. Transactions are first in a *working phase* where they work as if they were isolated from the other ones. They can access and make changes to objects on private copies. When a transaction is about to commit, it enters the *validation phase*. If the validation phase detects no conflict, the changes made become durable. If conflicts are found, transactions are aborted. In case of restarting every aborted transaction, the model can lead to *starvation*: a transaction would never be able to complete because of conflicts and would always restart.

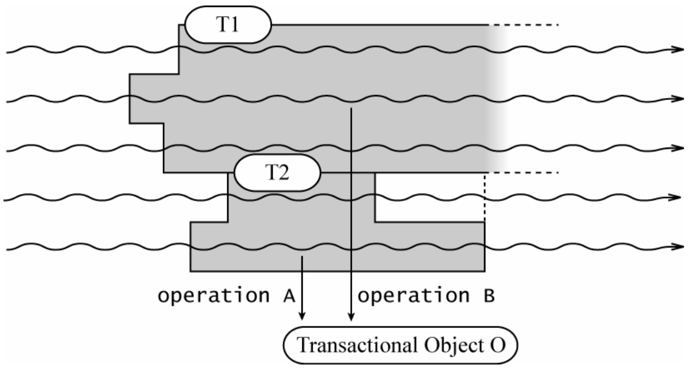


Figure 2.6 : Optimistic Concurrency Control Situation

Figure 2.6 shows a typical situation where two transactions execute concurrently. Operation A and B are conflicting. Depending on the validation scheme used, the aborting choice are different as exposed in the following sections.

Forward Validation

When a transaction comes to its validation phase, forward validation schemes look onward to find conflicts with active transactions. If a conflict is found, the validating transaction rolls back. This validation scheme can lead to *wasted aborts*: a validating transaction might abort because of transactions that get aborted later on. In Figure 2.6, when T2 tries to validate, it sees the conflict with the active transaction T1. This results in aborting T2 and maybe restart it later on, as shown in Figure 2.7[1].

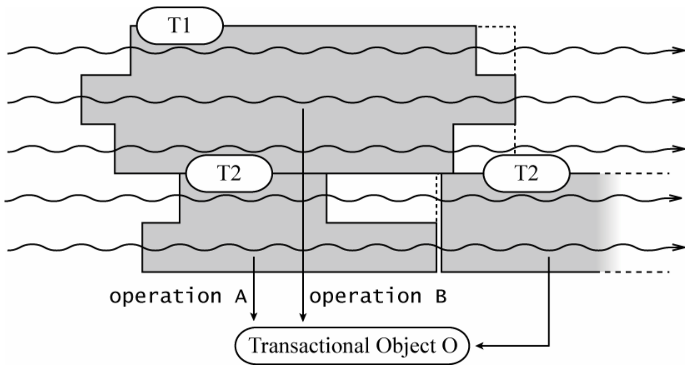


Figure 2.7 : Forward Validation

When T2 enters its validation phase, it finds a conflict with T1. It aborts and restarts. When T1 tries to validate, it finds no conflict with the active transaction T2 (operation A has not yet been executed on the private copy) and commits normally. T2 therefore continues its execution until committing.

Broadcast Commit

The broadcast commit validation scheme is a modification to the forward validation one in that, in case of a conflict, it is not the current transaction that aborts but every active conflicting one. As soon as a transaction comes to its validation point, it is assured to commit. If a conflict is found with any active transaction, a message is sent to it, informing that the current transaction has committed (broadcast commit). When the transactions receive this message, they abort and maybe restart. The wasted aborts problem is solved, but this is done at the expense of aborting every active conflicting transaction. The scheme can be described in Figure 2.6. As soon as T2 gets to its validation phase, it broadcasts its commit message to T1 that must abort and maybe restart.

[1] To be exact, when T2 tries to validate and fails in doing so, it has to rollback. This rolling back time is not shown in the figure, or we assume it is instantaneous. Another issue is that restarting T2 cannot be done synchronously by two threads: one has to start the transaction and then the second one can join it. An exception to this asynchronous joining rule is explained in [Kie04] with named transactions.

Backward Validation

In backward validation, when a transaction comes to its validation phase, it looks for conflicts with every “overlapping” already committed transaction. If the operations that the validating transaction has executed conflict with previously committed operations, then the transaction aborts. In Figure 2.6, when T2 comes to its validation phase, it commits normally. When T1 comes to its validation phase, it sees that the invoked operations in T2 conflict with its own operations and aborts.

2.5.3 Semantic-Based Concurrency Control

Assuming we have an `Account` class containing information on its owner, as shown in Figure 2.8, with respective getters and setters for every attribute, we define what conflicting operations are.

```
Class Account {
   User owner;
   float balance;
   float Withdraw(float amount);
   float Deposit(float amount);
}
```

```
Class User {
   String name;
   String lastName;
   String phoneNumber;
}
```

Figure 2.8 : Account and User Classes

Observers are methods that only read the object state without modifying it. Getters are typical observers like `Account.GetBalance()`, for instance. These operations are also called *readers*.

Modifiers are methods that modify the objects state. There are two types of modifiers: updaters are methods that change the state of an object depending on its previous state whereas over-writers do not care of the objects previous state when updating. *Updaters* first read the object value to update it (increment a counter, double a value) whereas *overwriters* only write (setters). These operations are also called *writers*.

Note: an updater can be decomposed in a read operation followed by a write operation. This means that when detecting a conflict, it can be either the read or the write operation that creates a conflict.

In basic strict locking concurrency control, when reading information from an `account` object, a read-lock on the object is granted, and subsequently only read-locks can be obtained by other operations – readers do not conflict with each other. As soon as a write-lock has been granted, no other lock can be obtained until the write-lock is released. Concurrency, in this scheme, is not optimal at all, because money withdrawal from an account[1] while updating its owner’s phone number[2] are not allowed; two operations which do no conflict because they share different data structures. The same situation would also be considered conflicting in an optimistic concurrency control, as it would be considered as two modifiers on a same object.

A solution to this problem is to adjust the conflict detection approach. In locking protocols, it would mean not to attach locks with invoking objects but with its

[1] Withdraw money from an account is a typical updater that can be decomposed into read-balance followed by write-balance.

[2] Updating a phone number is a typical overwriter: the new number does not depend on the old number value.

attributes or nested objects, adjusting the granularity of the locks. In an optimistic point of view, it would be not only to remember what kind of operation was applied on an object (observer or modifier), but what attributes or nested objects the operation was accessing. This approach restrains the conflicting domain between operations.

The idea can be described in the locking context as follows: the `withdraw` method would not acquire a write-lock on the account object, but on the `balance` attribute, so that the `owner` attribute is not locked anymore. A `setPhoneNumber()` on the `account.GetOwner()` would therefore be allowed. In an optimistic context, during the validation phase, the same idea can be applied stating that two modifier operations on the same object can be not conflicting.

We want to go further and state that two withdrawal operations can occur concurrently. If `balance` is greater than 1000, in example, `Account.withdraw(200)` and `Account.withdraw(500)` can occur at the same time without any conflict. We can even state that they can occur in any particular order. The result is still the same: in the end, `balance` is updated to 300.

The property allowing interchanging operations and still getting the same result is named *commutativity*. A way to achieve this is to provide the concurrency control a commutativity table for operations on objects. An entry in this table would state that `Account.withdraw(x)` and `Account.withdraw(y)` commute if and only if `Account.balance` is greater or equal to `x + y`, and therefore do not conflict. For a simpler example on a `Set` Class, please refer to [Kie04].

In the locking environment, locks would be associated with methods, stating that a withdraw-lock does not interfere with a setPhoneNumber-lock. Granting a withdraw-lock when one is already acquired would only be allowed if the object currently were in a specific state, i.e. when the balance attribute is greater than the withdrawal amount.

There are two types of commutativity, in which recovery techniques differ, they are explained in the following subsections. Additional information on commutativity can be found in [BR92].

Forward Commutativity

Forward commutativity is used in combination with **deferred update of objects**. Whenever executing a modifier operation, the changes are made on a separated copy of the object. In a basic scheme, the ordering of operations is not important, since both of them access the initial state of the object. In our accounting context, both withdrawal read a balance value of 1000 and therefore attempt to make their execution durable. If, in contrary, the balance is 1000 and we want to withdraw two times 800, then the execution of both operations are allowed, until validation of the second transaction, realizing that there is not enough money on the account.

The “forward” naming comes from the recovery technique used in case of aborting. The idea is that every operation until the conflict is redone until the conflicting operation, based on an initial consistent state of object.

Multi-version concurrency controls can be viewed as specialized extents to forward commutativity giving rules for accessing private copies of objects, i.e. based on timestamp.

Backward Commutativity

Backward commutativity is used in combination with **in-place update of objects**. Whenever executing an operation, the changes are made on the object itself, changing its current state. In a basic scheme, the ordering of operations is important and can change the behavior of transactions in early stage. Imagine we have an account with an initial balance of 1000, then, the execution of two withdrawals of 800 does not occur as in forward commutativity. The first withdrawal is allowed, changing the state of the object as soon as it is executed but the second one is aborted as soon as the new balance is read.

Backward commutativity is named after the backward recovery technique, which undoes every operation that should not have been done until a consistent state of the object is found. This result in the simple assumption that every operation must have an inverse, thus resulting in not executing anything (i.e. withdraw is the inverse of deposit).

Time Warp Protocols can be seen as a specific solution to in-place update of objects in an optimistic context.

2.5.4 Additional Notes

Semantics-Based Time Warp Protocols are discussed in [LA93], and information on real-time concurrency control, such as the **Two-Shadow Speculative Concurrency Control** can be found in [BBP93]. The latter one states that, as in branch prediction in processor architecture, whenever a critical operation is executed, two paths are followed: one as normal, when the execution is correct and another one, which is a copy of the actual one, speculating that the execution was wrong. Twice as much work is done, but we can state that one of the solutions is the correct one, in the end.

Concurrency control is an area of interest that is wide enough to contain transaction managers, real-time schedulers, database query languages execution and even pipelining in processors architecture. Plenty of papers were written since the early ages of computer science, but there is a lack of complete recent reference on the subject.

2.6 *Exception Handling*

This section covers a part of the exception handling mechanism developed for open multithreaded transactions [Kie00, Kie04]. Two important design decisions are:

- The model distinguishes internal and external exceptions; the latter ones are also called interface exceptions;
- Any external exception propagated from a transaction context is interpreted as an abort vote passed by the participant.

Each participant has a set of internal exceptions that must be handled inside the transaction, and a set of external exceptions, which are signaled to the outside of the transaction, when needed. If for any reason, the handler of the internal exception cannot deal with it, it can signal an external exception, resulting in aborting the transaction. The predefined external exception `Transaction_Abort` is always included in the set of external exceptions.

2.6.1 Internal Exceptions

After an internal exception is raised in a participant, the corresponding handler is called to handle it and to complete the participant's activity within the transaction. The handler can signal an external exception if it is not able to deal with the situation. In case an internal exception is not handled, the external exception `Transaction_Abort` is signaled. For a complete reference on internal exceptions, please refer to [Kie04]. What really interests us in our models is how external exceptions are handled. External exception handling in the initial model is described next.

2.6.2 External Exceptions

- External exceptions are signaled explicitly. Each participant can signal any of its external exceptions.
- Each joined participant of a transaction has a containing exception context.
- When an external exception is signaled by a joined participant, it is propagated to its containing context. If several joined participants signal an external exception, each of them propagates its own exception to its own context.
- If any participant of a transaction signals an external exception, the transaction is aborted, and the exception `Transaction_Abort` is signaled to all joined participants that vote commit.
- Because spawned participants do not outlive the transaction, they cannot signal any external exception except `Transaction_Abort`, which results in aborting the transaction.

Because the open multithreaded transaction model provides transaction nesting, the exception handling rules have to be applied "recursively" by the programmer. All external exceptions of a joined participant are internal exceptions of the calling environment.

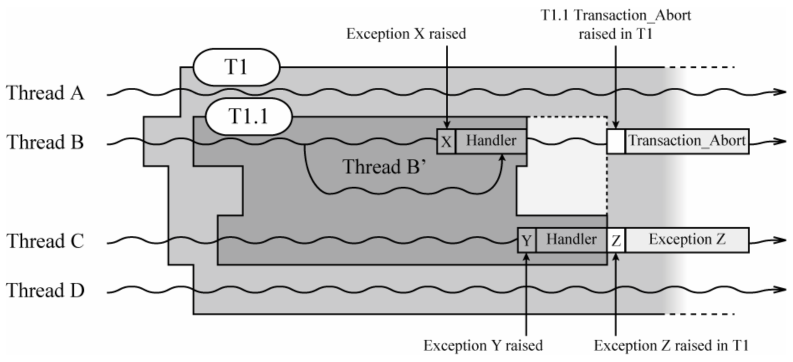


Figure 2.9 : Exceptions in Open Multithreaded Transactions

Figure 2.9 illustrates exception handling in open multithreaded transactions, especially when nesting occurs. Thread B that has started T1 also starts T1.1. Thread C joins it and then thread B spawns thread B'. Thread B' performs some work, votes commit and terminates. Thread B generates an exception while performing its work, but the exception is handled locally. It therefore does not affect the outcome of the transaction; after successful handling, thread B also votes commit. Unfortunately, thread C has generated an exception. It tries to handle it, but realizes that it cannot recover from this situation. It therefore raises an external exception, which causes T1 to abort. The exception Z is propagated to the calling environment of thread C: transaction T1. In all other joined participants, here thread B, the exception `Transaction_Abort` is raised to notify them of the transaction T1.1 abort. Now that the scope of the exception is T1.1, the previous external exception Z becomes an internal exception to T1 and can be handled or again not. The exception handling is therefore recursive in a nesting point of view.

If an interface or external exception has been raised, all participants should be informed about the abort of the transaction as soon as possible. There are two distinct approaches: non-preemptive and preemptive.

In the **non-preemptive** approach, each participant completes the transaction by voting commit or by signaling an interface exception in order to vote abort. If a participant votes abort, the other participants get to know that the transaction has aborted only once they vote commit. Non-preemption can decrease performance in applications with long running transactions. If one of the participants of a long running transaction votes abort just after the transaction has been created, all other participants would continue their now useless work until they reach their commit statement.

When using the **pre-emptive** approach, the transaction support does not wait for the participants to complete, but interrupts all participants as soon as one of them has signaled an external exception. This preemption often requires special run-time support. Its feasibility depends on the mechanisms provided by the programming language or on the underlying operating system. The model does not suffer from performance decrease for long transactions, but unfortunately introducing preemption mechanisms often results in some constant performance overhead even when these mechanisms are not used, i.e. for transactions that commit. Choosing the appropriate model, non-preemptive or preemptive, depends on the characteristics of the application.

Chapter 3 : Limitation & Possible Improvement

To preserve the isolation property, open multithreaded transactions do not allow participants to leave a transaction when they finished their work. The threads are blocked until the transaction outcome is known (synchronization on transaction exit as specified in section 2.4, page 8). Being blocked, these threads do not compute anything at all. On a single processor system, this might not really be an issue, since blocked threads do not take control of the processor anymore, but on a multiprocessor architecture or in a distributed context, this waste of computation time should be avoided as much as possible. The computational time could be used to speculatively execute the code that follows the transaction.

Therefore, we want to hypothesize on the transaction outcome and allow threads to execute the next logical operations after the transaction they were involved in, as if the transaction had committed. Being optimistic, we assume that transactions commit more often than they abort.

3.1 *Look-Ahead in Atomic Actions*

In the first following subsection, a short introduction to atomic actions is exposed, giving main differences and similarities between atomic actions and open multithreaded transactions, whereas the second subsection describes the look-ahead theory in atomic actions, as introduced in [Rom01].

3.1.1 Introduction to Atomic Actions

Atomic Actions [Lom77, Ree83] are program structures derived from *Conversations* [Ran75] introduced to build complex concurrent systems. A fixed number of threads enter an atomic action asynchronously; a recovery point is established in each of them. They freely exchange information within the atomic action, but cannot communicate with any outside thread (violation of this rule is called *information smuggling*). When all threads participating in the conversation have come to the end of the action, their acceptance tests are to be checked. If all tests have been satisfied, the threads leave the action together. Otherwise, they restore their states from the recovery points. Figure 3.1 shows the structure of an atomic action.

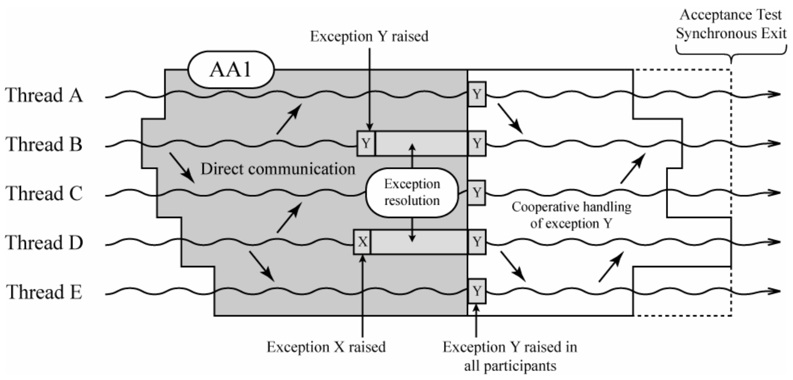


Figure 3.1 : An Atomic Action with Exception Handling

A fixed number of threads enter the atomic action and set their recovery point, then they communicate with each other until thread D raises an exception X, which is handled. Concurrently, thread B raises another exception Y. X and Y are compared (also called *exception resolution*) and the most important one is propagated to all the participants, so that it is handled cooperatively. Whenever they are all done, acceptance tests are checked to decide whether the participants can quit the action altogether or retrieve the last consistent state they were in.

The main similarities between atomic actions and open multithreaded transactions are the following:

- Open multithreaded transactions and atomic actions may be nested.
- Threads can enter atomic actions or open multithreaded transactions asynchronously.
- Threads must leave atomic actions or open multithreaded transactions synchronously.

The main differences between atomic actions and open multithreaded transactions are the following:

- The number of participants of an atomic action is fixed in advance, and hence no dynamic creation of threads is allowed; though this is a key point in open multithreaded transactions.
- Participants in open multithreaded transactions have their own separated local exception handling; whereas all participant threads recover together from an exception in atomic actions.
- Participants in open multithreaded transactions work with, and store their results in external objects; while in atomic actions, every participant threads have local states and share their results.

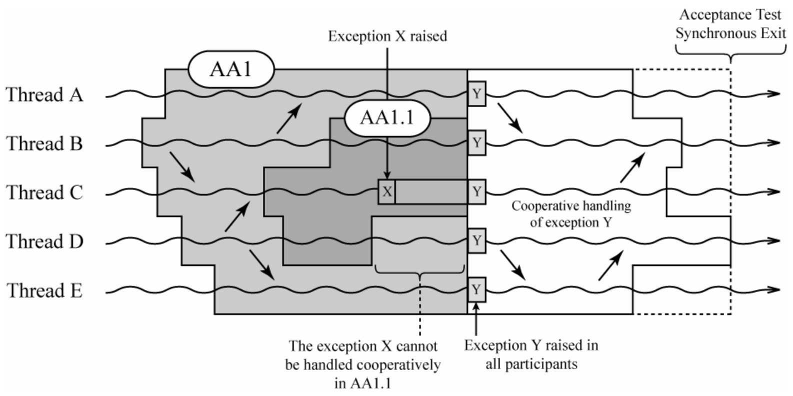
Synchronous exit, being a serious limitation to the model performance has lead to an attempt in relaxing it, as exposed in the next subsection.

3.1.2 Looking Ahead in Atomic Actions with Exception Handling

In [Rom01], a new atomic action scheme is introduced that does not impose any participant synchronization on exit. It states that letting threads leaving an action without waiting for all participants at the action exit is called *looking ahead*.

If a thread reaches the end of an action and is not aware of any exception inside it, it leaves the action and continues its execution. Such a thread is called a *look-ahead thread*. If a thread, in an action, raises an exception and this action has a participant that has looked ahead from it, then it is clearly not possible to handle this exception at the level of the action. To do this, all participant threads are needed to provide cooperative exception handling and to guarantee the absence of information smuggling.

Therefore, a look-ahead thread cannot be involved into recovery at the level of the action that it left as it has maybe entered other actions since then. The action context for exception handling is lost. The approach is then to find a containing action that includes all look-ahead threads and to involve all of its participants into cooperative handling to guarantee the consistency of recovery. The situation is shown in Figure 3.2. As mentioned previously, in open multithreaded transaction, no cooperative handling at this level is provided, thus not needing a dedicated protocol for exception handling.



Acceptance Test
Synchronous Exit
Exception X raised
AA1
AA1.1
Thread A
Thread B
Thread C
Thread D
Thread E
X
Y
Cooperative handling of exception Y
The exception X cannot be handled cooperatively in AA1.1
Exception Y raised in all participants

Figure 3.2 : Looking-Ahead in Atomic Actions

When thread C raises exception X, thread D has already looked ahead. Therefore, the exception resolution must find the containing action that includes all look-ahead threads (AA1) and makes the cooperative handling of the exception Y in this context.

3.2 *Objectives*

The objective of this diploma thesis is to apply the ideas of the atomic actions theory to the open multithreaded transactions context.

The new models we propose do not impose any participant synchronization on transaction exit. Avoiding the synchronization on exit, letting a thread leave it before its outcome is called *looking ahead*. Threads that leave a transaction without waiting for its outcome are called *look-ahead threads* and the transaction they were working in is called *former transaction*.

The models we propose still guarantee the ACID properties, isolation being the critical one. Different look-ahead models lead to different problems, solutions, and distinct exception handling.

The requirements are as follow:

- **Transparency**: we want to keep the transactions programming as simple as possible, exactly as it was in the *initial model*[1].
- **Compatibility**: programs designed for and running with the initial model should have no problem with the introduction of looking-ahead capabilities. In the worst case, we want to give the programmer the possibility of using the new models as if the transactions were executed in the initial model.
- **Robustness**: we want to provide models that enforce consistency, in any situation, providing complete and strong exception handling techniques.
- **Overall speed increase**: models should overcome the initial model main limitation providing a considerable gain in execution speed.

3.3 *Requirements Notes*

Fulfilling every requirement is not really possible when proposing models, thus, we want to give relations between them, showing that increasing one decreases another one, for example.

3.3.1 Transparency

The transparency rule states that every model proposed must be as simple as possible for the programmer, not adding too much constraint on programming, as new statement or design choices, in example. When allowing look-ahead, the user must only see a performance increase, but the global behavior of the system must be the same as in the initial model. The user, for example, must not be aware of a problem in a look-ahead if the former transaction is not completed yet. The logical time must be respected so that the programmer has always a global and traceable view of the execution.

3.3.2 Compatibility

The compatibility rule states that the programmer could be able to run his application on the new system without applying any changes. Respecting this rule can be done in two possibilities:

- A non-updated program runs in the initial model, thus not improved by allowing look-ahead, but respecting the transparency rule – a non-

[1] By initial model, we consider the model exposed in [Kie04], which is summarized in Chapter 2 : Main Concepts, page 7.

updated program behaves exactly as in the initial model, as looking-ahead is not enabled. In order to allow look-ahead, a program must be updated using new statements. A simple example is in handling the commit statement: in this case, a simple commit statement would mean to use the initial model and a new `speculative_commit` statement would mean to use the new model.

- A non-updated program is executed in a look-ahead context, thus resulting in a consistent execution (robustness requirement) but maybe not the exact same as in the initial model (see Chapter 4 : Potential Issues, page 29). In order to have the same execution, the programmer would need to update the program, with new statements (i.e. `wait-for-commit`) to recover the same synchronization on exit as in the initial model.

Following the first idea would mean that in order to allow looking-ahead, new statement must be added, at the expense of the transparency rule, but states that a non-updated application would be executed exactly the same as in the initial model, fulfilling the compatibility rule.

The second idea is that a complete compatibility is assured by adding new statements but the transparency is completely fulfilled when looking-ahead.

We are optimist and choose the second idea in the model propositions, stating that always allowing look-ahead threads would result as an improvement to the initial model execution, and that different – although consistent – behavior would be rare.

In every model proposed, both ideas can be applied, adjusting some statements. These variations, where applicable, are described in the concerned models.

3.3.3 Robustness

The robustness requirement states that executing an application in a look-ahead context must result in a consistent execution. This requirement is always completely fulfilled in the models proposed.

3.3.4 Overall Speed Increase

The overall speed increase requirement states that the execution of an application must provide significant performance speed increase. This requirement is very hard to quantify, as only an implementation can give results, but comparing the proposed models can give a good overview of which one is the best alternative. The overall speed increase is directly dependent on what information the programmer gives to the system. Thus, providing means to retrieve such information, new statement must be added, decreasing the transparency requirement wished.

3.4 *Look-Ahead Thread in Open Multithreaded Transactions*

We want now to give some definitions in a looking-ahead context. Look-ahead threads are threads that execute instructions after their commit vote, predicting the commit of the transaction they were working in. There are two main possibilities that happen when the thread leaves the transaction and becomes a look-ahead:

- The look-ahead creates or joins a transaction, called *look-ahead transaction*.
- The look-ahead executes code that is not creation or joining of a transaction, called *lone code*.

To denote a look-ahead transaction of a former transaction T1, we use the $T'^{(T1)}$ notation. In case of a second level look-ahead transaction, we can use $T'^{(T'(T1))}$ or $T''^{(T1)}$ to simplify writing. A look-ahead transaction of j^th level would be written $T^{j(T1)}$.The look-ahead number *j* is called *look-ahead amount* and characterizes the number of look-ahead that can occur while normally blocked waiting for the transaction outcome. If two threads create or join two different look-ahead transactions of T1, they are denoted $T'_1{}^{(T1)}$ and $T'_2{}^{(T1)}$.

Figure 3.3 illustrates an execution of open multithreaded transactions in the initial model. A possible execution, allowing looking-ahead, of the same system is illustrated in Figure 3.4.

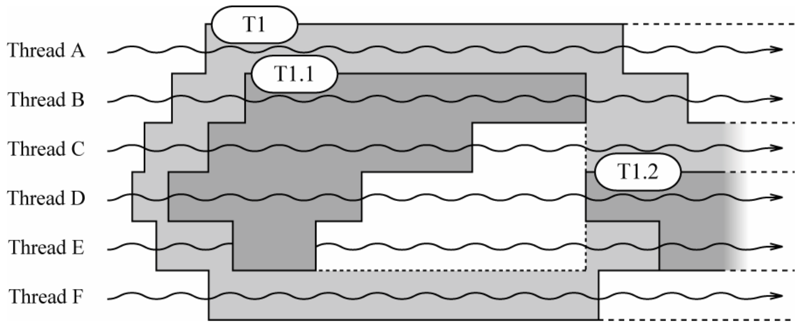


Figure 3.3 : Execution of Open Multithreaded Transactions in the Initial Model

Look-ahead transactions and lone code are shown in Figure 3.4. Thread C, as soon as it has voted commit in T1.1, looks-ahead and executes lone code until completion of T1.1 and then T1. Thread D, after voting, looks ahead and creates transaction $T'^{(T1.1)}$. Thread E looks ahead, executes some lone code and then joins the transaction created by thread D.

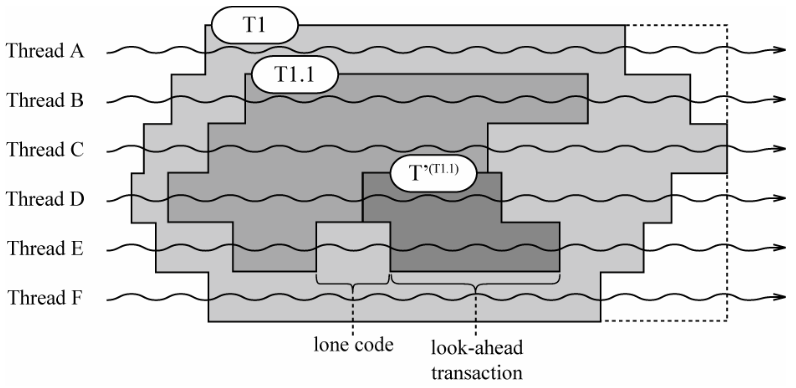


Figure 3.4 : Looking-Ahead: Transactions & Lone Code

The graphical comparison of Figure 3.3 and Figure 3.4 shows a good improvement in the overall execution speed, as essential in the requirements.

Section 3.4.1 covers the look-ahead transaction a thread may create or join whereas section 3.4.2 exposes the lone code analysis.

3.4.1 Look-Ahead: Transaction

It the transaction creates or joins a new transaction after the former transaction outcome, it is called a *look-ahead transaction*. Operations in a look-ahead transaction can be independent or dependent on the outcome of the former transaction. We introduce this classification in order to clarify our design choices made in the proposed models supporting look-ahead transactions. Please note that the dependency is semantic, and not related to the transactional objects that are accessed. A dependent look-ahead transaction might not involve accesses to the same transactional objects than the former transaction; and an independent look-ahead could. The classification is explained in the following sub-sections.

Semantic Dependency

Two transactions are called semantically dependent when the execution of the second one depends on the first one's outcome. In auctions, one might place a bid for a printer only after winning a bid for a computer (a printer is only useful if you also have the computer). In bank transactions, one might wait for a deposit to pay a bill. In other words, we only want to execute the semantically dependent transaction if the first one succeeds.

The first example is semantically dependent but independent in terms of transactional objects: we want to bid for a computer and then for a printer, which are different objects – a printer without a computer does not interest the user, whereas a computer might still be useful without a printer[1].

Whereas the second example is, in contrary, also dependent in terms of transactional objects: the user is transferring money on the same account object.

The idea behind dependent transactions is that if the former transaction T1 fails to commit after its look-ahead $T'^{(T1)}$ has started, both transactions abort – we do not want to execute $T'^{(T1)}$ if T1 aborted, resulting in a valid and consistent state where none of the two transactions are executed. Therefore, a dependency link is introduced: as soon as the former transaction aborts, its dependent look-ahead transactions also abort. When dependent transactions must be distinguished from independent transactions, they are denoted $T_D'^{(T1)}$.

Semantic Independency

Two transactions are called semantically independent if the result of the first one does not affect the second one's execution. Actually, operations could even be executed out of order. If a transaction is used to convert a multimedia flux, nested transactions could handle different sound channels, or process the image and the sound independently. In the auction system, a sportsman could bid for a tennis racket

[1] Depending on the implementation, both transactions would be linked with a `userID` defining which bidder is bidding or `balance`, in order to check if the user has enough money to bid. These examples show there is a lot of chance, that when transactions are semantically dependent, they might use same objects, being also dependent in terms of transactional objects accesses.

and for a snowboard. If the bid for the tennis racket is aborted, the user still wants his bidding for the snowboard to be valid. Independent transactions are denoted $T_I'^{(T1)}$.

The idea behind independent look-ahead transactions is that if transaction T1 fails to commit after its independent look-ahead $T_I'^{(T1)}$ has started, we abort T1, and inform $T_I'^{(T1)}$ of T1's abnormal termination. $T_I'^{(T1)}$ has different choices of behavior (exposed in the proposed models, whenever applicable). One of them is to abort itself, which results in the exact same behavior as if both transactions were dependent[1]. Note that this simple choice fulfills the transparency requirement, stating that the execution in a look-ahead model must be the same as an execution in the initial model.

3.4.2 Look-Ahead : Lone Code

Instructions executed by the look-ahead threads immediately after leaving the former transaction that are not encapsulated by another transaction are called *lone code*. Just as before, the lone code can be dependent or independent, but in both cases, there is no trivial way to undo only the lone code. Therefore, we do not have to distinguish the two cases.

[1] This affirmation is very important; because it validates the fact that independent transactions can be managed as dependent transactions, in case the implementation of independent look-ahead transaction turns out to be too complex.

Chapter 4 : Potential Issues

This section introduces potential problems that can occur when allowing threads to look-ahead from open multithreaded transactions.

4.1 *Spawning Threads*

As mentioned in Chapter 2 : Main Concepts, threads can be forked and terminated in a transaction. Rules state that both creation and termination have to occur in the same transaction. We have then to assure this rule is not transgressed when introducing looking-ahead.

4.2 *Read/Write Dependencies*

Read/Write dependencies occur when the former transaction and one of its look-ahead threads tries to access the same transactional objects in a conflicting manner. As we have seen in section 2.5.3, page 15, every operation, as complex as it could be, can be decomposed in terms of conflicting operations, into single reader or writer operations. The following figures illustrate the cases with transactions, but the situation is the same for look-ahead threads executing lone code. The following issues all refer to the situation depicted in Figure 4.1.

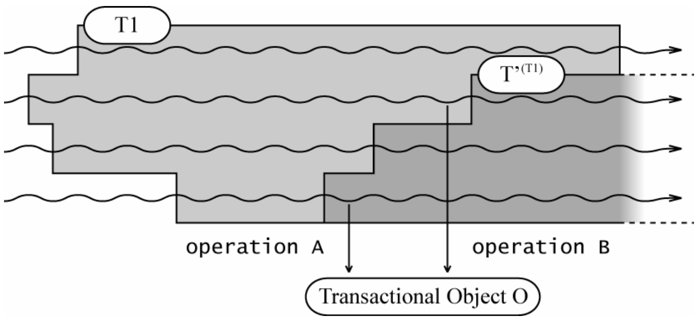


Figure 4.1 : General Read/Write Dependency Model

The figure shows two transactions: T1, the former transaction, and the look-ahead transaction T'$^{(T1)}$. Operation A, which is part of the look-ahead thread execution, takes place before operation B, executed from within T1. Operation A and B try to access an arbitrary attribute x of the transactional object O.

It the operations were executed in the initial model without look-ahead, no dependency would ever occur, since transaction T'$^{(T1)}$ would only begin as soon as T1 outcome is known.

To clarify the figure and in order to save space, we use a condensed way of representation, as shown in Figure 4.2.

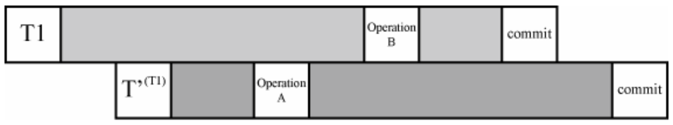


Figure 4.2 : General Read/Write Dependency Model (Simple View)

This situation results in the four separate cases that have to be analyzed.

1) RAR (Read-after-Read): operation A is a read, and operation B is a read. This is not a real dependency, as nothing is modified at all, RAR is not bringing out any conflict. Reading an attribute of the transactional object O is shown in Figure 4.3.

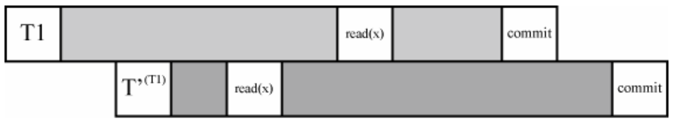


Figure 4.3 : Simple RAR (Read-After-Read) Dependency Model

2) WAR (Write-after-Read): operation A is a read, and operation B is a write. T'$^{(T1)}$ reads a value, which has not been modified yet by T1. The respective accesses to the transactional object O are presented in Figure 4.4.

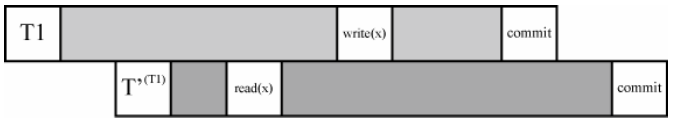


Figure 4.4 : Simple WAR (Write-After-Read) Dependency Model

3) RAW (Read-after-Write): operation A is a write, and operation B is a read. T1 tries to read a value which has already been accessed by T'$^{(T1)}$. The Read-after-Write dependency is exposed in Figure 4.5.

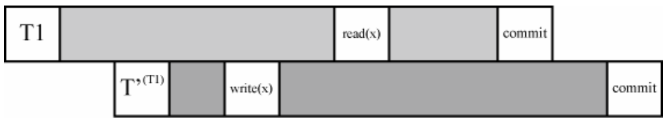


Figure 4.5 : Simple RAW (Read-After-Write) Dependency Model

4) WAW (Write-after-Write): operation A is a write, and operation B is a write. T1 tries to write in object O as T'$^{(T1)}$ has already accessed it. This case is summarized in Figure 4.6.

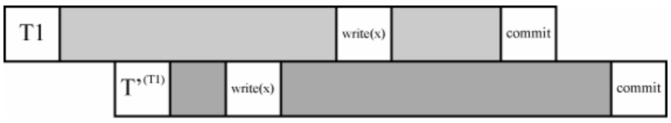


Figure 4.6 : Simple WAW (Write-After-Write) Dependency Model

In Chapter 5 : Look-Ahead & Concurrency Controls, these read/write dependencies are discussed in each concurrency controls previously exposed, and solutions are given to handle the conflicts.

4.3 *Multiple Objects Access - Deadlock Analysis*

Another problem, caused by accesses to transactional objects, can be deadlock presence or starvation issue (as described in section 2.5, page 9). This problem is again a result of look-ahead introduction. In the initial model, a former transaction T1 would have completed before its look-ahead T'$^{(T1)}$ had begun, not yielding any conflicts. Figure 4.7 illustrates this situation with transactions, but the same problem is also present with lone code.

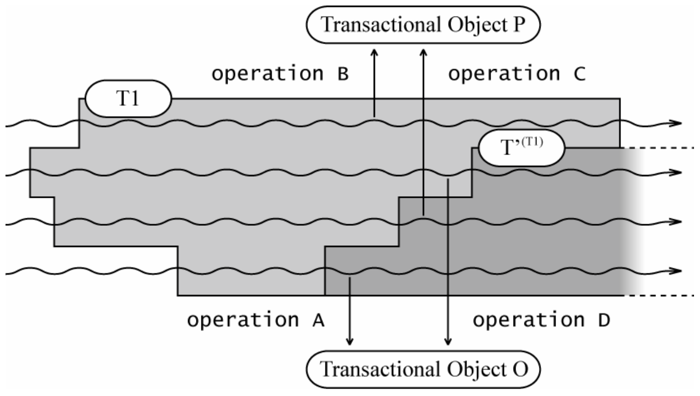


Figure 4.7 : General Deadlock Dependency Model

Operation A, which takes place in the look-ahead thread, accesses the transactional object O before T1 attempts in doing so. T1 accesses the transactional object P before T'$^{(T1)}$ tries to access it. It is obvious that in case of a lock-based protocol, deadlock occurs, if no modification is made to the current concurrency control scheme.

In case of a timestamp algorithm, it could happen that T1 gets aborted because of its look-ahead, which we also want to prevent in order to respect the transparency requirement. The same situation is also possible in case of an optimistic concurrency control (especially in forward validation).

4.4 *Nesting*

As exposed in Chapter 2 : Main Concepts, transactions can be nested. In atomic actions, a thread is allowed to look-ahead from any nested action until the top-level one, which is the last minimum action containing all possible actions that could need being aborted. It can be done in open multithreaded transactions, but in some models, it might be useful to restrict look-ahead thread behavior not letting them leave the current level of nesting.

4.5 *Short Look-Aheads & Amount of Look-Aheads*

Short look-ahead threads are of two kinds: short look-ahead transactions and short look-ahead lone code. The following sections give definitions of short look-ahead threads in both transactions and lone code.

4.5.1 Short Look-Ahead Transactions

A short look-ahead transaction is a look-ahead transaction that tries to commit before knowing its former transaction outcome. The situation is shown in Figure 4.8.

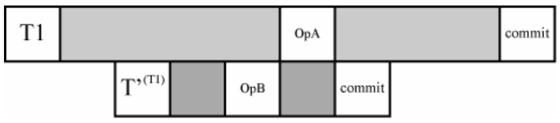


Figure 4.8 : Model of a Short Look-Ahead Transaction

If OpA and OpB conflict with each other, the problem is a read/write dependency, as exposed above, but in case they do not conflict, $T'^{(T1)}$ tries to commit or validate before its former transaction. We have now two different possibilities:

- $T'^{(T1)}$ is an independent transaction $T'^{(T1)}_I$, therefore both transactions can be executed out of order.
- $T'^{(T1)}$ is a dependent transaction $T'^{(T1)}_D$ and cannot be committed before T1 outcome is known.

In order respect the compatibility requirement (section 3.2, page 23), the look-ahead transactions must commit in the same order as if they had been executed in the initial model – in the logical time order. This means that the transactions are not dynamically rescheduled; even though this choice is not optimal in term of overall performance, i.e. independent transactions could be executed out of order.

As soon as the former transaction commits, every pending look-ahead transactions must try to commit in the logical time order.

In **pessimistic concurrency control**, as soon as a look-ahead transaction reaches its commit point, it is blocked until the former transaction is completed. Therefore, in locking protocol, the locks of the look-ahead transaction are not released at the moment. If an active transaction conflicts with the pending transaction, it is aborted and maybe restarted, as stated in section 5.1.2, page 41. In timestamp ordering, as the pending transaction is still active, it can also be aborted and restarted because of the former transaction. When the former transaction is completed, every pending transaction that was not aborted can simply commit respecting the logical time order.

In **optimistic concurrency control**, look-ahead transactions must also validate in respect with the logical time order. Every look-ahead must be prevented from validating before its former transaction. If they are blocked right before the validation phase, the former transaction still has the possibility of invalidating some operations, thus finding conflicts. As soon as the former transaction has validated, its look-ahead are allowed to validate, in the logical time order. In case of conflicts, the validating transaction must be aborted and maybe restarted, as explained in section 5.2.3, page 43. In forward validation, the former transaction might abort and remove pending look-ahead transactions, whereas in backward validation, it is absolutely mandatory that look-ahead threads validate after their former transactions.

4.5.2 Short Look-Ahead Lone Code

A short look-ahead lone code is look-ahead lone code that executes a create, join or commit operation before knowing the former transaction outcome. Figure 4.9 shows the different possibilities.

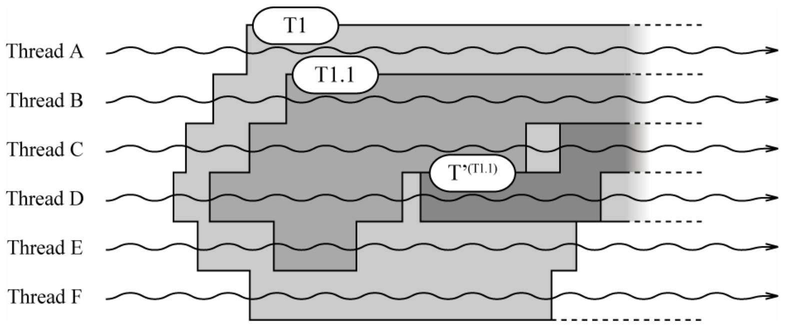


Figure 4.9 : Short Look-Ahead Lone Code

When leaving T1.1, thread C executes lone code and then joins a looking-ahead transaction, before knowing T1.1 outcome. Thread D executes lone code and creates a looking-ahead transaction, whereas thread E votes commit for the containing T1 transaction. All of these executions include short look-ahead lone code as defined above. As opposed to the short transactions, lone code cannot be blocked until committing or validating, because there is now way of undoing it, in order to recover the initial consistent state.

4.5.3 Amount of Look-Aheads

When a look-ahead thread enters several transactions sequentially, from $T'^{(T1)}$ to $T^{j(T1)}$, *j* is called *amount of look-aheads*. This number could be fixed to 0 (resulting in the initial model) or 1, for instance, or simply be unrestricted, so that look-ahead threads can freely look-ahead from any look-ahead transaction.

In look-ahead transactions, having an unrestricted amount of look-ahead is particularly useful when the former transaction is long running and when many look-ahead threads can be executed and added to the queue before the former transaction outcome is known.

As exposed previously, two design choices appear when committing a transaction. The commit statement can lead to blocking the voting thread as in the initial model, or to allowing looking-ahead.

Considering the `commit` statement as blocking (as in the initial model), a new `speculative_commit` is added to allow looking-ahead. Figure 4.10 shows a typical example of consecutive speculative commit statements.

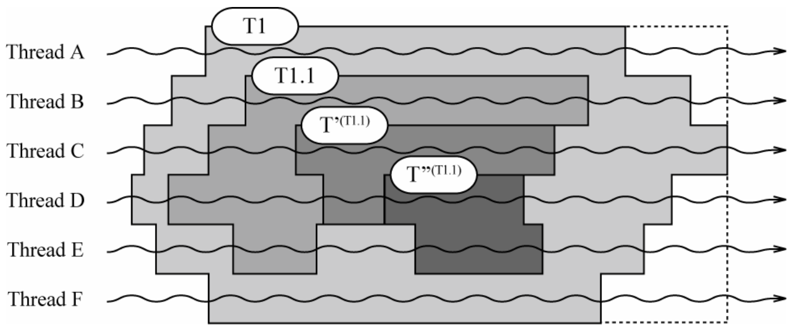


Figure 4.10 : Speculative Commit Statements in Short Look-Aheads

Thread E consecutively joins T1 and T1.1. When looking-ahead from T1.1, it executes lone code and joins a look-ahead transaction, T''$^{(T1.1)}$. This transaction has two participants, threads D and E. Both threads vote for another speculative commit statement and therefore look-ahead before the former transaction T'$^{(T1.1)}$ or the root former transaction T1.1 outcome is known.

If Thread D voted for a normal blocking commit, instead of the speculative one, the following design choices would appear:

- Participant thread is blocked until the **last former transaction** it was involved in has committed, i.e. thread D is at least blocked until thread E has voted, as shown in Figure 4.11 and Figure 4.12.
- Participant thread is blocked until the **root former transaction** has committed, i.e. thread D is blocked until thread B has voted, as shown in Figure 4.13.

On the other hand, when considering the normal commit statement as allowing looking-ahead, a new statement, allowing synchronization between threads and transactions is added: `wait-for-commit`.

Last Former Transaction Blocking

The first design choice is a way to block a look-ahead until the current level of look-ahead is completed. Its implementation would be to wait until every participant vote. Rules can be proposed like defining the last participant as a *decisive thread*. When speculatively committing, the thread is always allowed to look-ahead. Then using a normal blocking commit, it is blocked until the decisive thread of the former transaction votes. If the decisive thread votes for a speculative commit, then the threads are allowed to look-ahead, if not, threads are blocked until the decisive thread in their former transaction has voted.

In Figure 4.11, thread D votes for a normal blocking commit in T''$^{(T1.1)}$, it is blocked until the decisive thread E votes for a speculative commit, thus allowing thread D to look-ahead.

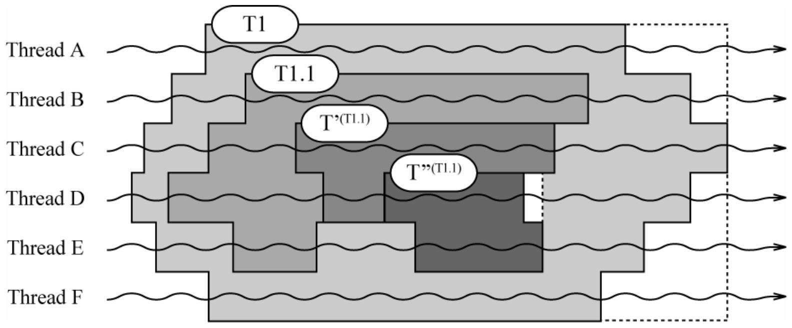


Figure 4.11 : Last Former Transaction Non-Blocking

If, in contrary, thread E also votes for a normal blocking thread, both threads E and D have to wait until the last former transaction T'$^{(T1.1)}$ is completed. If the decisive thread C votes for a speculative commit, every thread is allowed to look-ahead, as exposed in Figure 4.12. In the other case, if thread C votes for a normal commit statement, the three threads C, D and E are blocked until T1.1 outcome is known. As T1.1 is the root former transaction, this particular case would result in the root former transaction blocking design choice described in the next subsection.

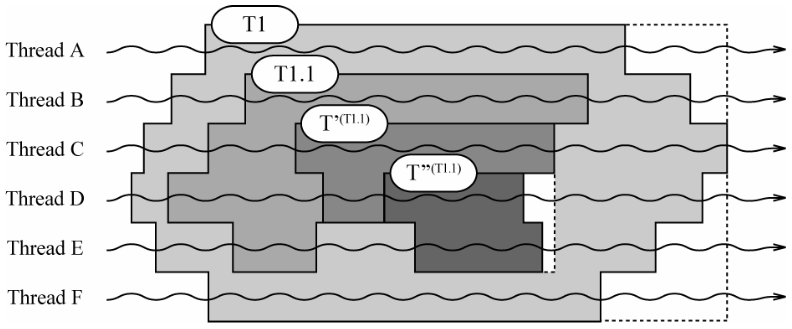


Figure 4.12 : Last Former Transaction Blocking

Root Former Transaction Blocking

The second design choice is a way to completely block a thread from looking-ahead, no matter in what level it is.

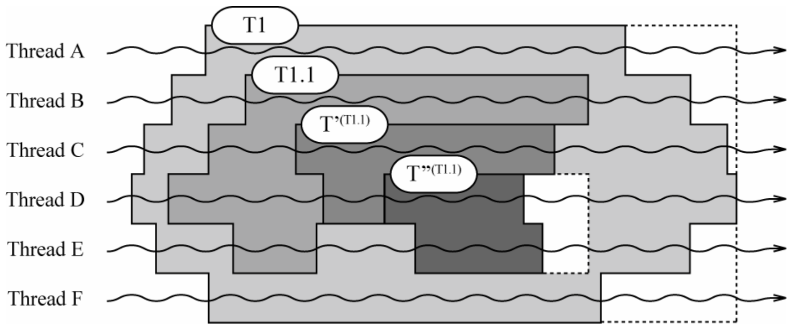


Figure 4.13 : Root Former Transaction Blocking

In Figure 4.13, when thread D and E vote for a normal commit in T''$^{(T1.1)}$, they are blocked until the root former transaction T1.1 is completed, no matter what T'$^{(T1.1)}$ decides.

Wait For Committing

If the normal commit statement is considered as allowing looking-ahead (from now on, it is the case), a means must be provided to the programmer for synchronizing threads and transactions. Adding a `wait-for-commit` statement is very convenient for the programmer, giving him means to ensure synchronization specifically.

A serious limitation in the last two design options is the random choice of the decisive thread. The decisive thread is not chosen by the programmer, but during the execution phase, in selecting the last committing thread. The `wait-for-commit` statement would allow the programmer to handle synchronization specifically in any level of look-ahead or nesting.

If the programmer does not want to allow look-ahead, a normal `commit` statement would be used and a `wait-for-commit` on the current transaction would be added. This would not allow any thread to look-ahead until the current transaction is completed: looking-ahead when the transaction outcome is known is equivalent to the initial model, without looking ahead.

4.6 *Exception Handling*

There are different problematic cases that must be solved in every proposed model:

- An exception is raised in the former transaction while one or several threads are already looking-ahead, *i.e. an internal exception that cannot be handled locally, thus propagating an external exception*.
- An exception is raised in the look-ahead context while the root former transaction is not completed.

4.6.1 Exception Raised in Former Transaction

When an internal exception is raised in the former transaction and as its handling does not involve every participant (non-cooperative), threads in the concerned former transaction do not need to be informed. Two options occur now: the exception is caught and handled – the concerned thread does not need to propagate it to the calling environment; or an external exception is raised in the containing scope and the participant threads of the concerned transaction must receive a new `LA_Transaction_Abort` exception. Doing so, they know the work they are now executing might be useless. The situation is shown in Figure 4.14. In case of pre-emptive approach, the exceptions are immediately raised in the look-ahead threads. In non-preemption, the exception would be raised only when the threads are voting.

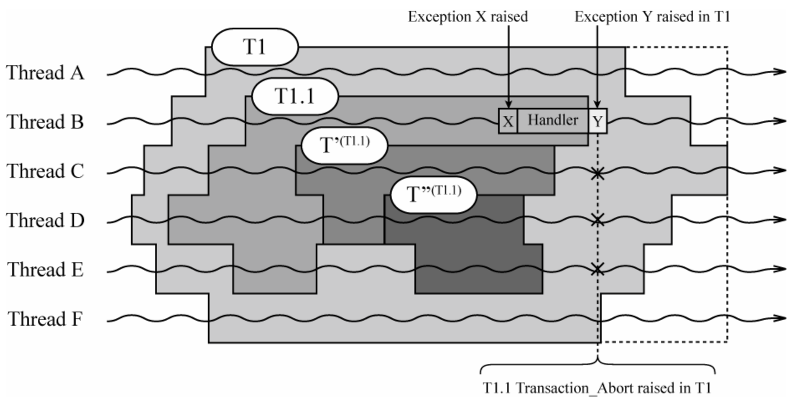


Figure 4.14 : Exception Raised in Root Former Transaction

Thread B, which is the last working thread in T1.1 catches an internal exception X and tries to handle it. Unfortunately, an external exception Y has to be raised in T1 context, thus propagating T1.1 `LA_Transaction_Abort` exceptions in every T1.1 participants (C, D and E)[1]. In order to respect the transparency rule and the fact that look-ahead might depend on the T1.1 outcome, every look-ahead has to undo whatever it has done after leaving the former transaction. The only particular case, where this statement is modified, is when the look-ahead is independent and undoable, which is covered in the models propositions.

4.6.2 Exception Raised in Look-Ahead Context

If an exception is raised during the look-ahead execution, no information must be provided to the former transaction in case of an exception in the look-ahead: giving information of the "future events" in the "past"[2] is of no use in the former transaction. The situation is shown in Figure 4.15.

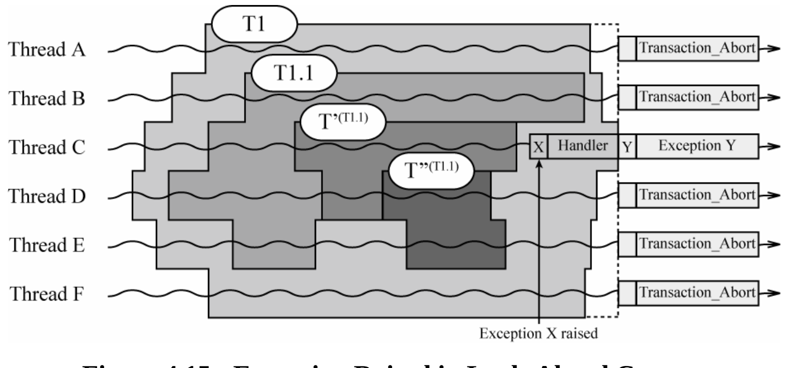


Figure 4.15 : Exception Raised in Look-Ahead Context

Thread C, when finished with T1.1 and T'$^{(T1.1)}$ executes code that raises an exception X in T1. When handled, in the worst case, it propagates an exception Y in its context, thus raising a `Transaction_Abort` in every participant thread of T1.

[1] Thread E, in the lone code preceding T''$^{(T1)}$ joining, could have done changes to an external object O, depending on the transaction outcome. If Thread F executed code because of these changes, the exception must also be raised in thread F.

[2] The time notions reference the logical time: the former transaction executes before its look-ahead in the initial model.

If executed in the initial model, the exact same result would occur, in the end, T1 must abort.

Chapter 5 : Look-Ahead & Concurrency Controls

In this chapter, we analyze how the look-ahead introduction behaves in the concurrency controls introduced in Chapter 2 : Main Concepts, and propose solutions whenever the initial concurrency control can be adapted to suit our needs. Two cases are analyzed: look-ahead transactions and lone code, but our main analysis focuses on the look-ahead transactions, as lone code can be considered as a non-undoable transaction in case of conflicts.

Being given the read/write dependencies (section 4.2, page 29), the possible solutions are analyzed in a look-ahead transaction context. The analysis is the same for semantically dependent and independent transactions. Whenever problems or solutions differ, it is clearly stated in the text.

Solutions are given when the potentially erroneous work is undoable. If the conflicting operations are executed during lone code, it is not trivially undoable. Thus, approaches trying to overcome this problem are exposed in the proposed models (Part I).

5.1 *Pessimistic*

We analyze how the pessimistic concurrency controls behave with introduction of look-ahead threads, given the read/write dependencies issues previously stated. Solutions when dealing with look-ahead transactions are confined in a dedicated subsection, for both lock-based protocols and timestamp ordering schedulers.

5.1.1 Read/Write Dependencies

In case of a **WAR (Write-after-Read):** operation A is a read, and operation B is a write, T'$^{(T1)}$ reads a value, which has not been modified yet by T1.

Operation A, which acquires a read-lock on the transactional object O does not authorize a write before the lock is released, which would be T'$^{(T1)}$ ending. This is not acceptable in respect with the serializability rule, T1 has a logical time priority over T'$^{(T1)}$. What we would have to do here is that when the write-lock is asked in T1, it has to be granted (override the permission of T'$^{(T1)}$). To do so, we have to abort T'$^{(T1)}$, which releases the read-lock, and restart it as soon as possible. T1 having the write-lock, T'$^{(T1)}$ has to wait until it is released to get the read-lock and continue its normal behavior. As the write-lock in T1 is released in its commit phase, T'$^{(T1)}$ read permission is granted when T1 is finished, as shown in Figure 5.1. Note: If the first operation of T'$^{(T1)}$ was a read, introduction of look-ahead behaves exactly as in the initial model – so to say that T'$^{(T1)}$ has to wait for its read-lock until the end of T1.

In case of timestamp ordering, the idea is similar to lock-based protocols and yields the same results: whenever the write operation tries to invalidate the read operation in T'$^{(T1)}$, it is not T1 that must be aborted but T'$^{(T1)}$.

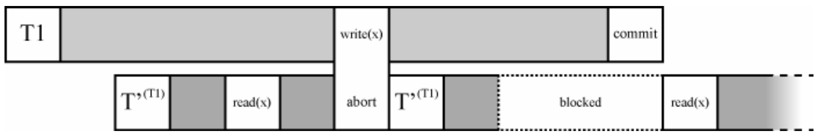


Figure 5.1 : Solution to WAR Dependency in Pessimistic Concurrency Control

In case of a **RAW (Read-after-Write):** operation A is a write, and operation B is a read, T1 tries to read a value which has already been accessed by T'$^{(T1)}$.

This is not acceptable in respect with the serializability rule, T1 occurs before its successor T'$^{(T1)}$. Any writing in T'$^{(T1)}$ should take place after T1's last reading. In fact, as T'$^{(T1)}$ obtain its writing-lock, no reading-lock is granted to T1, which blocks it until T'$^{(T1)}$ is finished. The same idea as for WAR applies here: to override any lock obtained by T'$^{(T1)}$, abort it, and restart it as soon as possible. In timestamp ordering, the idea is still to abort the look-ahead and not the former transaction. The result is visible in Figure 5.2.

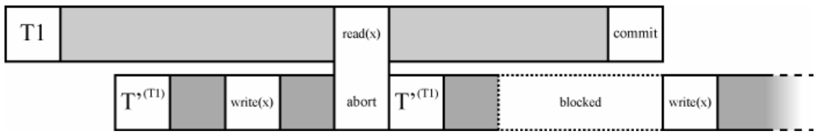


Figure 5.2 : Solution to RAW Dependency in Pessimistic Concurrency Control

In case of a **WAW (Write-after-Write):** operation A is a write, and operation B is a write, T1 tries to write in object O as T'$^{(T1)}$ has already accessed it. This is also not acceptable in respect with the serializability rule: T1 occurs before its successor and if a write-lock is granted to T'$^{(T1)}$, T1 waits until the outcome of its successor. The solution to this problem is the same as for a WAR or RAW dependency: always allow the former transaction to get lock, overriding T'$^{(T1)}$ permissions or to abort the look-ahead instead of the former transaction in timestamp ordering. Figure 5.3 shows the result.

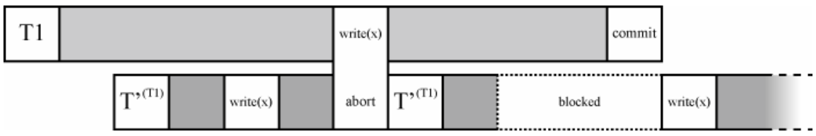


Figure 5.3 : Solution to WAW Dependency in Pessimistic Concurrency Control

5.1.2 Solution to Read/Write Dependencies

T1, preceding T'$^{(T1)}$ always has higher rights than its successor in order to fulfill the serializability rule, as shown in the following tables.

T'$^{(T1)}$	T1 has read-lock	T1 has write-lock
read(x)	yes	no
write(x)	no	no

T1	T'$^{(T1)}$ has read-lock	T'$^{(T1)}$ has write-lock
read(x)	yes	yes, restart T'$^{(T1)}$
write(x)	yes, restart T'$^{(T1)}$	yes, restart T'$^{(T1)}$

Figure 5.4 : Solution to Read/Write Dependencies in Locking Protocol

In timestamp ordering or multiversion timestamping, the following rule must be followed: in case of a conflict between a transaction and its look-ahead, the former transaction always has the higher priority, thus always aborting the look-ahead.

5.1.3 Multiple Objects Access – Deadlock Analysis

One of the pessimistic concurrency control bottleneck, beside its decrease of performance implementing locks, is the deadlock presence, which is hard to detect. We prove that deadlocks cannot appear using the solution to read/write dependencies between former transaction and their look-ahead threads. Let us analyze what happens in Figure 4.7.

When operation C tries to get object P's lock, it gets blocked. Object P's lock was granted to T1 when operation B was executed. But now, when T1 tries to get object O's lock, it does not get blocked, it takes priority over object O's lock and T'$^{(T1)}$ has to be restarted. Operation A and C, waiting for the locks are blocked until the outcome of T1, which releases the locks. By giving priority to T1 in accessing objects, we avoid deadlocks between former transactions and their look-ahead. This does not mean that deadlock does not exist anymore, it only states that the deadlock introduced in look-ahead theory can be handled easily. The result of applying our solution in a deadlock case is shown in Figure 5.5.

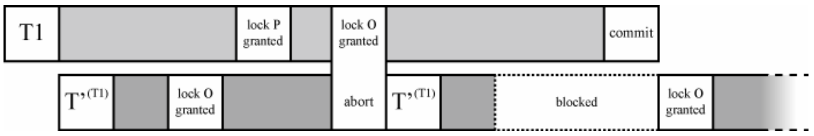


Figure 5.5 : Solution to Deadlock in Pessimistic Concurrency Control

5.2 *Optimistic*

This section exposes how the optimistic concurrency control behaves when introducing look-ahead threads. Solutions when dealing with look-ahead transactions are stated in dedicated subsections, for the different validation techniques. The figures are presented as if modifications to objects were done on private copies. Whenever an in-place update would result in a different behavior than with deferred update, comments are given.

5.2.1 Read/Write Dependencies with Forward Validation

In case of a **WAR (Write-after-Read):** operation A is a read, and operation B is a write, $T'^{(T1)}$ reads a value, which has not been modified yet by T1. This dependency is exposed in Figure 5.6.

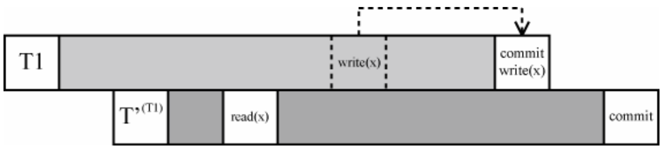


Figure 5.6 : WAR Dependency in Optimistic Concurrency Control

As soon as T1 tries to validate, it detects a conflict with an active transaction ($T'^{(T1)}$) about the transactional object O. T1 aborts, which also aborts $T'^{(T1)}$. As $T'^{(T1)}$ was the cause of the conflict we have a *wasted abort*. Introducing the transactional object dependency between T1 and $T'^{(T1)}$ results in a cyclic dependency. A solution to this problem is to force the former transaction to commit, thus aborting the look-ahead. This solution can be considered as a broadcast commit between former transactions and their look-ahead transactions. Therefore the broadcast commit protocol does not need a specific solution is this case.

In case of a **RAW (Read-after-Write):** operation A is a write, and operation B is a read, T1 reads a value which would already been accessed by $T'^{(T1)}$.

As exposed in section 2.5.2, page 13, the write operation in $T'^{(T1)}$ is only visible to other transactions after being validated. This means that T1 accesses the initial unmodified value of x. Thus, the RAW dependency is not a problem; in this case, nothing has to be modified.

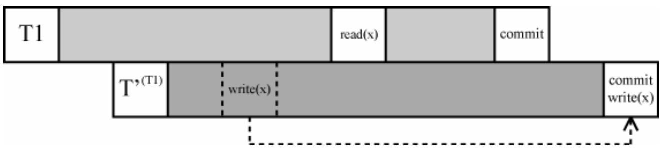


Figure 5.7 : RAW Dependency in Optimistic Concurrency Control

This situation has to be nuanced when working with in-place update of objects. The writer, taking place in the look-ahead transaction, would have an impact on the former transaction if not undone before the reader operation. Thus, when detecting the conflict (during T1 validation phase), T1 discovers a conflict with its active look-ahead, resulting in aborting itself. If both transactions can be executed out of order (i.e. semantically independent), then we are fine, but in the other case, T1 aborting results in its look-ahead abortion, which was cause of the conflict. This is another example of cyclic dependency. The solution is to undo the conflicting operation of the look-ahead before attempting to execute the one in the former transaction. This approach is similar to Time Warp Protocols, but in our interests, we want to state that in case of semantically independent transactions, this might not be a problem.

In case of a **WAW (Write-after-Write):** operation A is a write, and operation B is a write, T1 writes in object A, so does T'$^{(T1)}$.

As shown in Figure 5.8, both write operations occur during the commit phase. As there is no reading in this case, there is no conflict discovered during the validation process. WAW dependency does not introduce a cyclic aborting, because writing does not depend on the object's initial state.

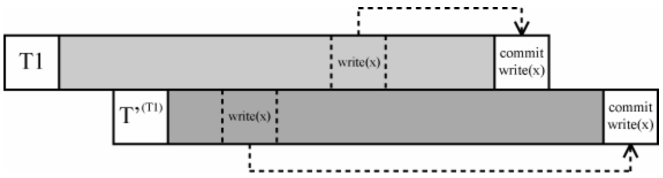


Figure 5.8 : WAW Dependency in Optimistic Concurrency Control

5.2.2 Read/Write Dependencies with Backward Validation

In backward validation, the previously exposed figures always assume the look-ahead transaction tries to validate after the former transaction. In any of these cases, backward validation, as it looks backward with overlapping already committed transaction, never aborts the former transaction because of the look-ahead.

The main problem with backward validation is in case of short look-ahead transactions as exposed in section 4.5, page 32. The solution proposed is to block the short look-ahead until the former transaction has validated. If conflicts are detected, then it is the look-ahead transaction that aborts and maybe restarts.

5.2.3 Solution to Read/Write Dependencies

Forward Validation

The validation phase has to be adapted in order to avoid cyclic dependencies between former transactions and their look-ahead. When a transaction tries to validate and finds out conflicts with an active transaction. It has to differentiate between look-ahead and normal transaction. If every conflicting transaction is of look-ahead type, then the validating transaction commits and abort-restarts every conflicting transaction. This can be considered as a broadcast commit when conflicting transactions are look-ahead threads.

Broadcast Commit

In terms of look-ahead, broadcast commit is analyzed exactly as in typical forward validation. Please note that in case of short look-ahead threads, they must validate after their former transactions. Doing so, they could be aborted because of active transactions. This is against the broadcast commit protocol idea but is necessary in order not to abort a former transaction because of any of its look-ahead threads.

Backward Validation

In backward validation, no new solution has to be provided in case of look-ahead introduction. Short look-ahead threads form the only issue in backward validation, and thus, the solutions in section 4.5, page 32, handles this situation completely.

5.2.4 Multiple Objects Access – Deadlock Analysis

Optimistic concurrency control ignores what a deadlock is, because it is a pessimistic specific problem. However, its dual problem is starvation, as mentioned in section 2.5.2, page 13, and as explained in the pessimistic multiple objects access, whenever a conflict is found priority is given to former transactions, thus always restarting look-ahead threads. This results in no new starvation between transactions when working with private copies of objects.

The situation is different with in-place update of objects. In case of conflicts between a former transaction and its look-ahead, the concurrency control has to give priority to the former transaction. Nevertheless, the problem, with in-place update, is that the conflicting operations are interleaved and already executed. This means that aborting any of them would automatically result in aborting the other invoking transaction. If restarting aborted transactions, we have a typical starvation problem. To avoid this, whenever a conflict is detected, conflicting operations must be rolled back before finding a correct possible sequence of execution that does not implicate conflicts. This is the purpose of Time Warp Protocols.

5.3 *Discussion*

As shown in previous sections, it is possible to maintain both pessimistic and optimistic concurrency controls features when introducing look-aheads in open multithreaded transactions, thus respecting the **compatibility** requirement.

In theory, the **overall speed increase** seems consequent showing that the look-ahead introduction results in a faster execution than in the initial model. In the worst case, aborting a look-ahead to restart it is done when committing the former transaction, which is the exact same situation as in the initial model: the first transaction has to commit before the second one begins. Of course, this statement is nuanced as soon as models are proposed, taking in account exception handling and lone-code execution.

Part II

Proposed Models

The next chapters present different ways of introducing look-ahead in open multithreaded transactions. Each chapter describes a model with slightly different rules for letting a thread leave the former transaction in order to perform operations speculatively.

Chapter 6 : **“Leave”**, page 49, presents the first model of look-ahead introduction. It is based on the model for atomic actions proposed in [Rom01]. It also is the most basic and most straightforward.

Chapter 7 : **“Stretch”**, page 55, describes a model based on “Leave”, trying to overcome one of its major limitations.

Chapter 8 : **“Create”**, page 63, introduces a model specifically dedicated to immediate creation or joining of following transactions.

Chapter 9 : **“Implicit”**, page 73, describes a model based on “Create”, adding capability to handle lone code as robustly as transactions.

Every model proposed contains the following sections, describing its features and limitations:

Concept

The concept section of the model proposals describes the general idea on how the model behaves. It provides information whether new statements are introduced and if any rule in the initial model is threatened. It also gives an overview on what main requirement the model targets.

Concurrency Control Analysis

The concurrency control analysis section discusses the ability of the model to manage the concurrence between the former transactions and its look-ahead threads. It states solutions on how to handle look-ahead threads whether they execute transactions or lone code.

Exception Handling

As exposed in section 4.6, page 36, the models must be able to handle exceptions in the former transaction when already executing a look-ahead. In this section, discussion is exposed in case of looking ahead from several nested transactions, changing the exception scopes.

Specific Limitations

The specific limitations section describes the model weaknesses and strengths, stating its intrinsic limitations and how they can be addressed.

Discussion

The discussion section evaluates the conformance of the model to the initial requirements, its suitability to handle conflicts between the former transaction and its look-ahead thread, and the possible exceptions. The limitations of the model and the problems found in concurrency controls are summarized, giving a good overview of the model possible usefulness.

Variations

The final section gives a list of possible modifications to the model, whether it is in order to fulfill a specific requirement more than any other or to improve the model general performance.

Chapter 6 : Leave

6.1 *Concept*

The “Leave” model is simple and straightforward. The idea is to let threads leave the transaction after their votes, and continue execution exactly as if the transaction had committed. The model is illustrated in Figure 6.1. Thread D starts T1, and then a nested transaction T1.1. Thread E joins T1, then T1.1, and finally commits instead of being blocked. Thread E can now look-ahead: it executes code outside of T1.1, but inside T1. An obvious restriction to a thread leaving a transaction is that it must be in a containing transaction – the former transaction must be nested in another one. This restriction is mandatory in term of consistence. If for any reason, the look-ahead executes code that it should not be allowed to, the erroneous execution can still be undone by rolling back the containing transaction, in order to result in a consistent state. Therefore, in the “Leave” model, **threads involved in top-level transactions cannot look-ahead**, as illustrated in the figure.

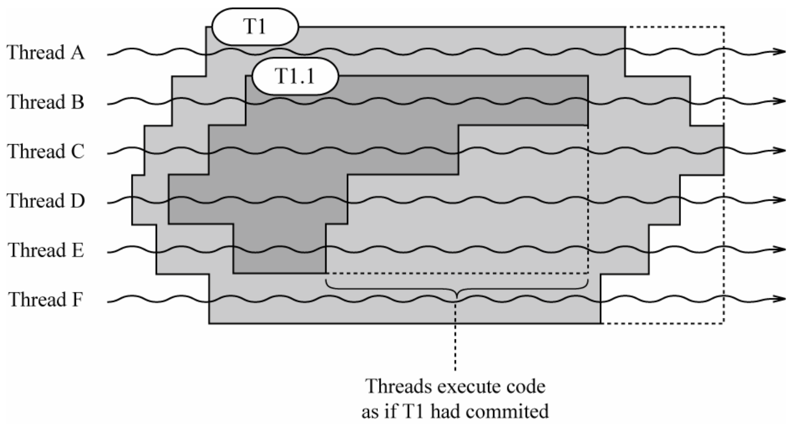


Figure 6.1 : Illustration of the "Leave" Model

6.1.1 New Statements

The `commit` statement is considered allowing look-ahead threads. Therefore, the system must provide a mean for synchronization of threads and transaction completion. This is done with the previously exposed `wait-for-commit` statement, page 36.

6.1.2 Notes on Spawning Threads

Whenever a thread is forked, it still terminates in the same transaction it was created in. Therefore, the rules defining the creation and termination of threads inside transactions, on page 7 are not bypassed.

6.2 *Concurrency Control Analysis*

We analyze now the concurrency control problems that can occur in the looking-ahead introduction, respecting the rules stated in section 2.5, page 17.

In the current model, no distinction is made between look-ahead transactions or lone code. If a conflict is found during the execution of look-ahead transactions, solutions are provided in Chapter 5 : Look-Ahead & Concurrency Controls, but there is now way, in this model, of knowing if the look-ahead transaction followed undoable lone code or not. Moreover, no information is given whether the look-ahead transaction is semantically dependent or independent. Hence, the treatment in case of read/write dependencies must be independent of the kind of look-ahead we have.

Logical time and semantic dependency state that the former transaction happens before its look-ahead. Therefore, if the look-ahead executes a conflicting operation, it has to be undone – we have no way of knowing if the lone code contains conflicting operations before they are executed. We have to wait until the former transaction executes an operation that reveals the conflict with the look-ahead (pessimistic) or during its validation phase (optimistic).

In this model, the only way of undoing the look-ahead in order to recover the system to a consistent state is to abort the *innermost containing* transaction, which obviously aborts the former transaction itself. The phenomenon of aborting the parent transaction of the former one is named *parent abortion*, and is considered as the worst case, because not only the work done in the former transaction is lost, but also the execution in the parent transaction, preceding the former transaction's execution.

Whenever a conflict occurs, the system has to abort and restart the innermost containing transaction, disabling looking-ahead for the involved transaction and all its children.

6.3 *Exception Handling*

We investigate now the exception handling in the former transaction while look-ahead threads are already executing. In case of an internal exception, the handling is done normally, as explained in section 2.6.1, page 18, and if it cannot be handled locally, an external exception is propagated. In that case, the look-ahead threads have to be informed, with a new `LA_Transaction_Abort` exception. Figure 6.2 illustrates the idea.

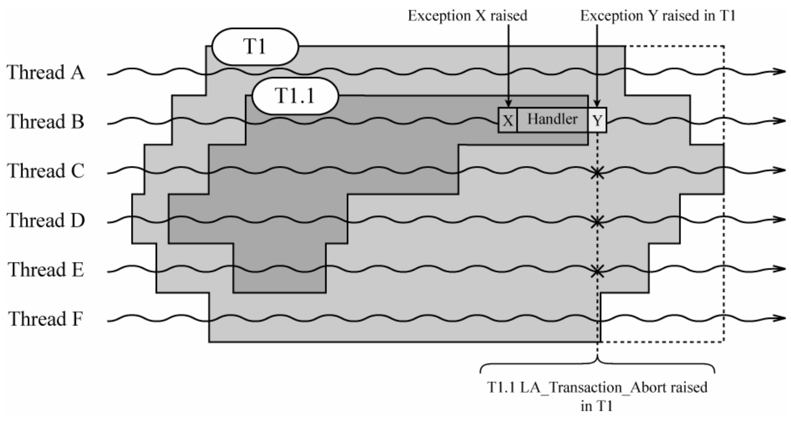


Figure 6.2 : Exception Handling in the "Leave" Model

Thread B when executing work in the former transaction raises an exception that it handles, and then propagates an exception Y to T1. Doing so, T1.1 `LA_Transaction_Abort` exceptions are raised in every former participant thread[1] of T1.1.

Two choices appear in the look-ahead threads execution:

- Transparency goal: The exception is not caught by the user and is automatically handled by the system. It results in aborting the innermost transaction containing all the concerned threads (see 2.6.1, page 18), which is T1 in Figure 6.2.
- Overall performance goal: the programmer has anticipated this possibility, and wants to handle the exception in the look-ahead context. This assumption means the programmer knows about the exception possibility, thus going against the transparency rule. This approach might lead to problems of cooperative concurrency between

[1] If thread E and F are cooperating after thread E has looked ahead, then the exception must also be raised in the context of thread F, even though it did not take part in the former transaction. This is why a new exception is raised, instead of the regular `Transaction_Abort`.

participants and outside former transaction threads and nesting issue, as exposed in the next sub-section.

This second point is put aside, because of its complexity and is explained as a modification to the model in section 6.6, page 53.

6.3.1 Nesting

As decided above, in case of conflicts or exception, the parent transaction must be aborted. If a thread inside T1.1.1 is allowed to look-ahead, continuing its execution in T1.1, and then looks ahead again, it is now executing code inside T1. Thus, in case of an exception in the root former transaction T1.1.1 the innermost containing transaction is T1, and now T1 must be aborted.

If a thread is allowed to look-ahead from its parent transaction, the granularity of the exception is decreased and in order to guarantee consistency, the grandparent must be aborted. This situation is shown in Figure 6.3.

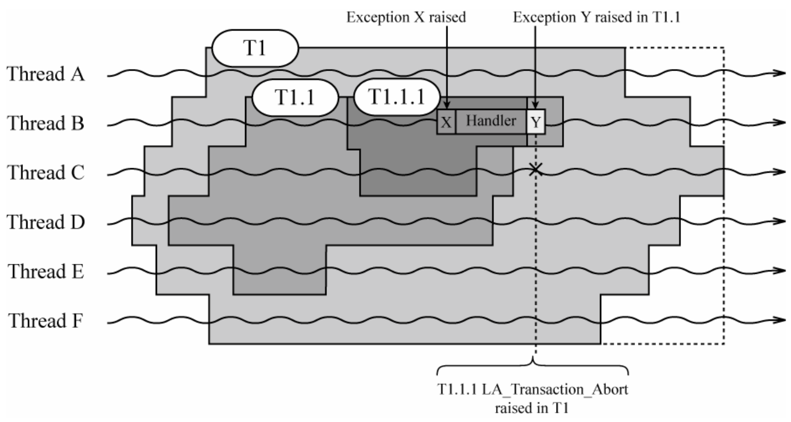


Figure 6.3 : Exception Handling in the "Leave" Model, with Nesting

Thread B, when it propagates the exception Y to T1.1, raises T1.1.1 `LA_Transaction_Abort` in thread C. As mentioned previously, if there was cooperative concurrency, introducing dependency between a former participant thread and any outside thread, an exception must be raised in every concerned thread, in case the programmer wants to handle it. Otherwise, the innermost containing transaction aborts, and results in the consistent state just before its beginning.

In general, if a thread is allowed to look-ahead from n nested transactions, assuming m is the actual level of the nested transaction looked-ahead from, the innermost transaction to abort is of level $(m-n)$. Stating that threads involved in top-level transactions cannot look-ahead assures that m is always strictly greater than n, trivially resulting, in the worst case, in aborting the top-level transaction *(level 1)*.

6.4 *Specific Limitations*

The limitations of the model are the following:

- Looking-ahead cannot take place in top level-transactions.
- The model is inefficient (parent abortion) if the blocking time in the initial model is long enough to execute a single conflicting operation with the former transaction: not only the work done in the former

transaction is lost, but also every operations that took place in the parent transaction before the beginning of the former transaction.
- The model is inefficient (parent abortion) in case of an exception raised in the former transaction, for the same reasons.

Using this model would only be worthwhile in systems using at least one level of nested transaction, and when the nested transaction contains a lot of threads, where the time difference between every blocking thread is not long enough to create conflicts but still significant enough to use look-ahead threads. Using this model is also possible when long-running threads, that postpone the outcome of the transaction, are executing operations that have practically no chance of creating a conflict. (i.e. waiting for a specific object state before committing, waiting for a specific event to commit: static time or network message).

6.5 *Discussion*

As expected, a simple model like "Leave" is not giving terrific results. The only case where the model is efficient is when no conflicting operations during look-ahead take place before the former transaction outcome, and that the former transaction is contained in at least one parent – this only happens when the blocking time in the initial model is not long enough to execute any conflicting operation. This inefficiency can be explained in not being able to recover at the same level as the problem was found, always having to abort the containing transaction.

6.6 *Variations*

This section provides improvements in the current model, or exposes different design choices that can be made.

6.6.1 New statements

A slight modification to the model can be changing the system's interpretation of the `commit` statement. So far, it was interpreted as allowing looking ahead, thus the synchronization between threads and their former transaction, if needed, would be done with the `wait-for-commit` statement.

Instead of this, the system could consider the commit statement as the normal blocking commit statement – as in the initial model, and add a new `speculate_commit` that allows look-ahead execution of the voting thread.

6.6.2 Exception Handling

In the exception handling section, solutions that do not give the user the possibility of handling `LA_Transaction_Abort` are favored. This is done for two main reasons:

- The model must be the most transparent possible for the user.
- The exception can be raised in any level of nested transaction. Therefore, if the programmer wants to handle it, not only the handler

might be complicated, but the programmer must provide a handler for each different level of nested look-ahead transactions, which might become too cumbersome.

- The exception can be raised in any thread, because of the possible cooperative concurrency between threads inside a transaction.

If we want the model to provide these features to the user, whether the cooperative concurrency rules must be redefined, or the cooperation between threads must be traced by the system in order to raise the `LA_Transaction_Abort` exceptions in the concerned threads.

6.6.3 Concurrency Control Issues

When a read/write dependency occurs, the solution is to abort the transaction and restart the execution disabling look-ahead, as exposed in section 6.2, page 50. A more sophisticated solution would be to disable looking-ahead only when exactly needed, as one pass of the process is already done, we know exactly where the conflict occurred. The drawback to this solution is that it could run *x* times if there are *x* more possible conflicts, whereas the simple algorithm would only re-execute once.

A better solution would then be to do a complete analysis of the executed code during the first try, so that the best synchronization, without conflict, would be found and executed during the second pass. This approach goes beyond the scope of the thesis and is not exposed anymore.

Chapter 7 : Stretch

7.1 *Concept*

The “Stretch” model is based on the “Leave” model, trying to overcome one of its major problems: incapacity of handling exception at the same level they were raised. In the “Leave” model, we had to abort the containing transaction in order to undo the (potentially erroneous) operations that look-ahead participants have executed after the commit. The “Stretch” model tries to overcome this problem by extending the transaction beyond its usual border: if a participant commits, it can continue executing code, but this code is still executed from within the same transaction context, keeping the look-ahead threads isolated from the containing transaction. The model still does not distinguish between look-ahead transactions and lone code, therefore, conflicts still pose problems. The improvement is that instead of aborting the parent transaction, only the stretched transaction must abort. Figure 7.1 illustrates the model.

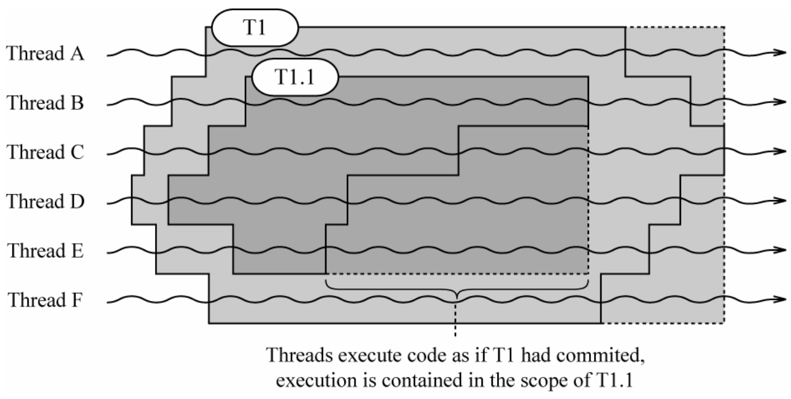

T1
T1.1
Thread A
Thread B
Thread C
Thread D
Thread E
Thread F
Threads execute code as if T1 had commited,
execution is contained in the scope of T1.1

Figure 7.1 : Illustration of the "Stretch" Model

The general idea is that whenever the transaction is aborted or an inconsistency is detected, we do not have to abort the parent but only the stretched transaction, thus retrieving the consistent state at the beginning of the former transaction. Stretching the former transaction is a way of setting a surely consistent recovery point at the beginning of the transaction.

7.1.1 Notes on Spawning Threads

Now that we are changing the transaction borders, we have to assure the rules defining thread creation and termination inside a transaction, page 7, are not bypassed. If ever T1.1 forks a thread, it has to terminate when committing, thus, it cannot look-ahead.

Thus, we only have to handle the following situations, commenting Figure 7.1:

- T1 forks a thread that joins T1.1 and tries to terminate inside T1.1 stretched context (T1).
- T1 forks a thread inside T1.1 stretched context.

In both cases, whenever we want to abort T1.1, resulting in aborting the stretched transaction T1.1, threads would have to be killed, which is undesirable. Therefore, creating or terminating thread must be blocked until the end of the stretched area. The termination of a thread is shown in Figure 7.2.

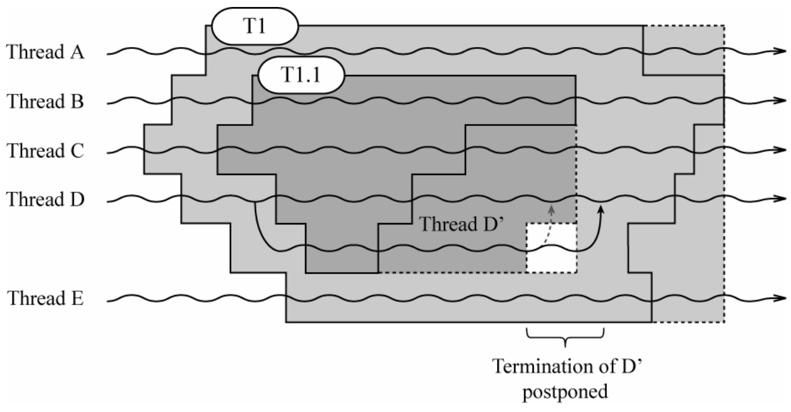


Figure 7.2 : Terminating Thread is Postponed in "Stretch" Model

When the forked thread D’ tries to terminate, it should normally be able to terminate in T1 context, but as the borders of the former transaction T1.1 have changed, the termination cannot take place in T1.1 stretched area. Therefore, the termination of the thread is postponed after the outcome of the former transaction T1.1 is known.

7.2 *Concurrency Control Analysis*

The read/write dependencies, as described in the “Leave” model are still present in the “Stretch” model, but their handling can be now improved significantly.

In case of conflicts, we do not want to abort the parent transaction but only the stretched former transaction, resulting in the consistent state just before the former transaction’s beginning. Doing so, we can use the same easy solution as in the previous model: restarting the former transaction with look-ahead disabled[1].

A behavior, which is specific to the “Stretch” model, is that execution of the look-ahead code, outside of the transaction is still done in the context of the former transaction. Thus, execution of operations in the stretched area is isolated from the outside world, in particular from the containing transaction. We want now to show that cooperative concurrency is not changed using the stretch model than if it was in the initial model.

Figure 7.3 shows an example of cooperative concurrency between threads.

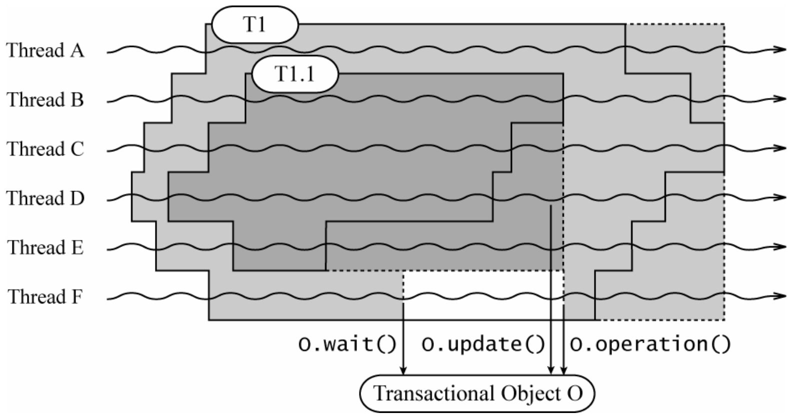


Figure 7.3 : Cooperative Concurrency in the "Stretch" Model

Thread F, which is outside of the former transaction executes code inside T1 and is synchronized with thread D waiting for changes on the object O. Waiting is not causing conflict anyhow, therefore the thread is sleeping until the object state changes. When thread D updates O, the changes are not made visible to the outside threads until completion of the former transaction, thank to the isolation rule. At this moment, thread F recovers its execution.

This is coherent with what would have happened in the initial model, as shown in Figure 7.4.

[1] The improvements given in section 6.6, page 53, can be applied to this solution.

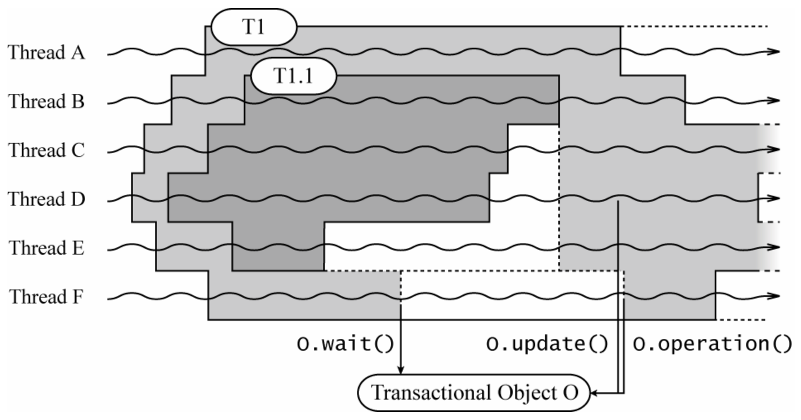


Figure 7.4 : Cooperative Concurrency in Initial Model

Thread F, waits for an update on O, as in the previous case. Now that thread D is blocked until thread B has completed its work for T1.1, its update to object O is postponed after T1.1 outcome is known, thus slowing down thread F a little more. As soon as the update is made, thread F is synchronized and can use O, to continue its normal execution.

What we can state, from this example it that isolating the lone code contained in the stretch area from the outside thread is, even in the worst case, equivalent or better than the initial model.

On the other hand, competitive concurrency can be turned into cooperative concurrency: a thread executing in the stretch model should be isolated from the former transaction, but actually executes as in the former transaction – isolation rule is threatened. This is a result of executing speculative code before the outcome of the former transaction is known. Normally, the speculative code is synchronized thank to the blocked threads. The problem is a result of the absence of cooperative/competitive rules (which were not needed in the initial model). This case is addressed in section 7.6.2, page 61.

7.3 *Exception Handling*

In case of an exception raised in the former transaction while other threads are looking-ahead, the thread involved first handles the internal exception locally and if needed, propagates an external exception to the containing transaction context, which triggers a `Transaction_Abort` in every participant thread[1]. Figure 7.5 shows this particular situation.

[1] Thank to the absence of cooperative concurrency problems, no outside thread can be involved in a possible exception handling, thus only aborting the stretched transaction, in the worst case.

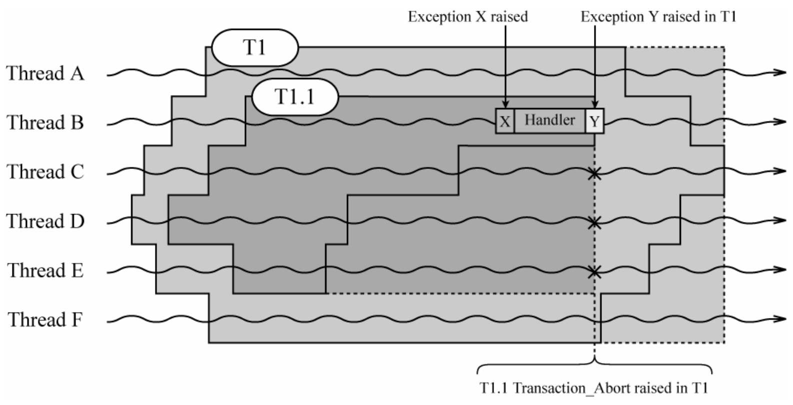


Figure 7.5 : Exception Handling in the "Stretch" Model

Thread B, when executing operations in the former transaction T1.1 raises an internal exception X that it cannot handle correctly, thus propagating exception Y to T1. In this case, T1.1 `Transaction_Abort` is raised in every former participant thread. It is not necessary in this case to have a new `LA_Transaction_Abort` as in the “Leave” model. This is a direct consequence of the absence of cooperative concurrency problem in the “Stretch” model. As mentioned before, the exception is raised only in the former participant threads that could not interfere with outside thread, which would have made them also dependent of the former transaction outcome.

7.3.1 Nesting

Exactly as in the previous subsection, in the worst case, the stretched transaction must be aborted. If a thread inside T1.1.1 is allowed to look-ahead to T1.1 (T1.1.1 stretched) and wants to look-ahead again, it is now executing code inside T1 (but still in the stretched context of T1.1.1). Thus, in case of an exception in the root former transaction T1.1.1 the stretched transaction must be aborted. This results in recovering the involved threads to the exact same state as right before the former transaction. Now that a thread is allowed to look-ahead in different level of nested transaction, a new `LA_Transaction_Abort` must be introduced.

Figure 7.6 shows the case where an exception is raised in a former transaction while one of its participants is already looking-ahead in another level of nesting. The stretched context is expanded until the former transaction’s end (commit or abort).

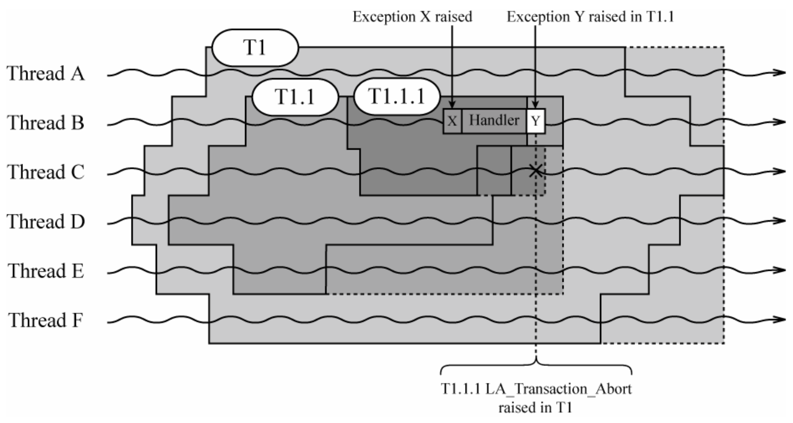


Figure 7.6 : Exception Handling in the “Stretch” Model, with Nesting

When thread B tries to handle the internal exception X, it propagates an external exception Y to T1.1. However, if thread C is allowed to look-ahead from T1.1.1 and also from T1.1, then it receives a T1.1.1 `LA_Transaction_Abort` in T1 (T1.1.1 stretched) [1]. In this case, it is very difficult for the programmer to handle the exception, as being in two higher levels than where the exception was raised – its handler must be declared in the scope of T1. If the programmer does not want to handle it, the system automatically aborts T1.1 stretched and restarts it disabling look-ahead or with any of the variations exposed in section 6.6, page 53. Thank to cooperative/competitive concurrency, when look-ahead threads are allowed to look-ahead from different nesting levels, the only – stretched – transaction to abort is still only the one that raised the exception.

7.4 *Specific Limitations*

As exposed in the previous sections, there are a couple of significant improvements over the "Leave" model, the main ones being:

- The model allows looking-ahead in top-level transactions.
- Abortion has been improved to recover the consistent state right before the start of the former transaction.
- Cooperative concurrency between look-ahead threads and outside threads is not a problem anymore.

Nevertheless, the model still does not take advantage of the solutions provided when dealing with look-ahead transactions, because it handles look-ahead transactions and lone code indifferently.

An issue in this model is that outside operations execute in the stretched context, i.e. T1 operations in T1.1 stretched. As expected, both transactions should be isolated from each other. However, in the current model, the outside operations are allowed to execute inside the stretched context, which turns the competitive concurrency into cooperative concurrency.

7.5 *Discussion*

The "Stretch" model overcomes the biggest problem of the "Leave" model, parent abortion, in giving a practical and general solution without any drawback compared to the "Leave" model. In addition, the model can now be used in case of top-level transaction, which is an important improvement.

The serious limitations that are kept are the inefficiency of aborting the former transaction in case of conflicting operations with the look-ahead.

[1] If the regular `Transaction_Abort` were used, it would mean for T1 in thread C, that T1.1 had aborted, which is not the case.

7.6 *Variations*

As summarized previously the “Stretch” model is a decent improvement over the “Leave” model. It could be simplified and one of its limitations could be overcome, as explained in the following sub-sections.

7.6.1 One-Level Nested Looking-Ahead

As exposed in section 7.3, page 58, the model does not need to add a new exception, in case of exception handling, when only allowing one level of nested look-ahead threads. This statement changes when unrestricted, and the exception handling can become very inconvenient to manage for the user. Therefore, a simple version of the “Stretch” model could only allow one level of nested looking-ahead, always providing the simplest exception handling situations for the user.

7.6.2 Competitive/Cooperative Concurrency Issue

As exposed in the limitations, the “Stretch” model has a conceptual problem in turning competitive concurrency into cooperative concurrency when threads in different transactions execute operations in the same context. A way to solve this particular case is to provide the concurrency control a way of differentiating between operations in the former transaction and operations in the stretched context. This could be done, in example by stamping the operations with their invoking transaction.

Chapter 8 : Create

8.1 *Concept*

In the previous models, we had no information on whether the look-ahead threads are executing look-ahead transactions or lone code. Thus, the solutions exposed in Chapter 5 : Look-Ahead & Concurrency Controls could not be of any use so far.

We know that in case of an abort in the former transaction, the code executed in look-ahead threads has to rollback, in order to fulfill the compatibility and transparency requirements, and because no information on the kind of code executed was provided. A dependency link between the former transaction and its look-ahead transactions is introduced: if the former transaction aborts, the look-ahead transactions must abort, or at least be informed of the situation. However, in the previous models, not only the potentially erroneous code was rolled back but also the former transaction itself, which this model tries to overcome. Thus, a model that is specifically dedicated to look-ahead transactions is introduced, as solutions for the read/write dependencies are provided for a proper handling. Therefore, look-ahead lone code is prohibited and a look-ahead thread is forced to create or join a look-ahead transaction.

Rules for the creation (section 8.3), joining (section 8.4) and ending (section 8.5) of a look-ahead transaction are described, giving several possibilities for an upcoming design.

8.2 *New statements*

In order to enable the “Create” model, we have to add additional commit possibilities. Keeping the standard `commit` statement that blocks the thread, new speculative look-ahead commit statements are added, which allow the current thread to start or join another open multithreaded transaction. Given that the transparency rule is not strictly enforced introducing these statements, we want to go further and differentiate between dependent and independent look-ahead transactions. Therefore, in order to create or join a semantically dependent look-ahead transaction, the following statements must be used :

- `dependent_commit_create`
- `dependent_commit_join`

In order to create or join a semantically independent look-ahead transaction, the following statements must be used :

- `independent_commit_create`
- `independent_commit_join`

Semantic dependency between transactions is explained in section 3.4.1, page 27. Differentiating independent transactions from dependent transaction results in more appropriate exception handling as exposed in section 8.8, page 69.

8.3 *Starting a Look-Ahead Transaction*

After its speculative commit vote, a thread can create a new transaction. This newly created transaction is bound to the initial one, because we speculate on the first one’s outcome. In the initial model, the new transaction would take place after the first one. Adding look-ahead capabilities introduce the fact that some operations may execute out of order (with respect to the initial model), and then can lead to read/write dependencies as exposed in section 4.2, page 29.

If a thread does not vote for a creation, it must whether use a joining statement or is blocked until the former transaction outcome, as in the initial model.

8.4 *Joining an Open Multithreaded Transaction*

After a speculative commit vote, a thread can also join an already existing transaction. In order to avoid circular dependencies, a thread is not allowed to join a transaction it has looked-ahead from (*re-join*). The problem is illustrated in Figure 8.1.

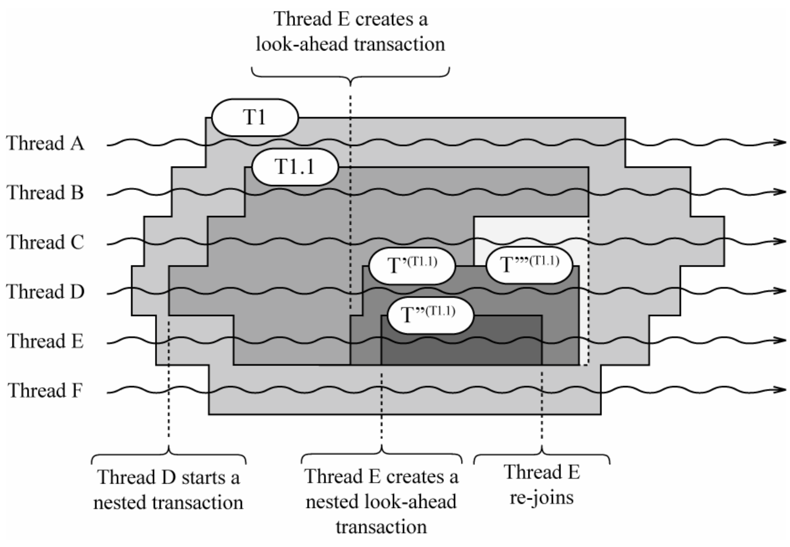

Thread E creates a look-ahead transaction
T1
T1.1
Thread A
Thread B
Thread C
Thread D
Thread E
Thread F
T'(T1.1)
T'''(T1.1)
T''(T1.1)
Thread D starts a nested transaction
Thread E creates a nested look-ahead transaction
Thread E re-joins

Figure 8.1 : Example of a Re-Joining Problem

Figure 8.1 shows the inconsistency problem of the T'$^{(T1.1)}$ transaction, which being re-joined is becoming T'''$^{(T1.1)}$. This means that it can only commit when T''$^{(T1.1)}$ has committed, which is only possible when T'$^{(T1.1)}$ has committed. As both T'$^{(T1.1)}$ and T'''$^{(T1.1)}$ are in fact the same transaction, we have a cyclic dependency that makes it impossible for either transaction to commit.

In addition to the "no re-joining rule", and depending on the situation and the probability of aborting the former transaction, we might want to restrict joining even further. We have several possibilities for the joining rules, as much for the look-ahead threads that for the regular thread. We introduce two sets of rules: *restrictive* and *open*. The restrictive set should be used in case of a relatively high aborting probability, whereas the open set should be used in a more optimistic environment. The classification is ordered by decreasing aborting probability of the former transaction.

8.4.1 Restrictive Set

The restrictive class allows look-ahead threads to only join look-ahead transactions. Regular threads are not allowed to join look-ahead transactions.

Self-Restrictive-Join

- Look-ahead threads of a transaction can only join look-ahead transactions that were created by a former participant thread of this particular transaction.
- Regular threads are not allowed to join look-ahead transactions.

Self-restrictive joining is illustrated in Figure 8.2.

This model is optimal in case of a high former transaction aborting probability. As we introduce a link between the former transaction and its look-ahead transactions, if for any reason the initial one fails to commit, the dependent look-ahead transactions are aborted. Since these transactions only contained other look-ahead threads, the abort does not cause any other work to be undone.

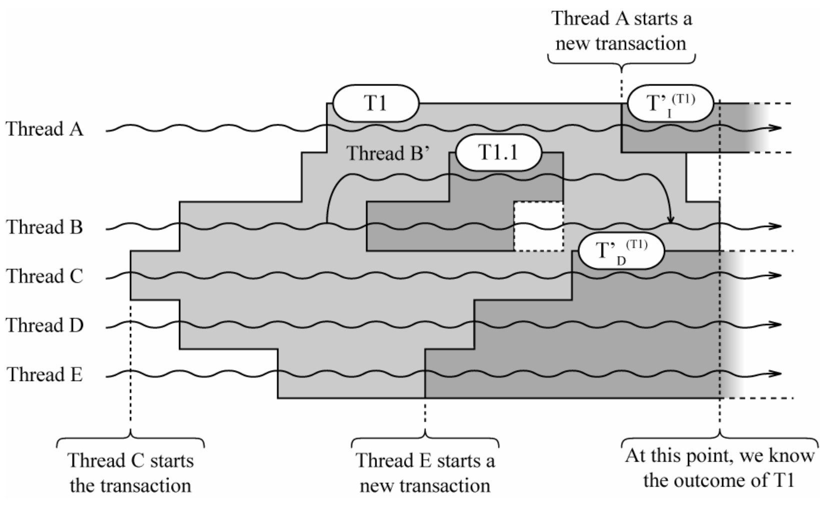

Thread A starts a new transaction
T1
T'$_I^{(T1)}$
Thread A
Thread B'
T1.1
Thread B
T'$_D^{(T1)}$
Thread C
Thread D
Thread E
Thread C starts the transaction
Thread E starts a new transaction
At this point, we know the outcome of T1

Figure 8.2 : Self-Restrictive Join

Figure 8.2 shows that thread E, which has finished first, votes for a dependent commit. It then creates a new transaction T'$_D^{(T1)}$, which depends on the outcome of T1. Participants threads, as soon as they have voted can join the newly created one (threads C and D). On the other hand, thread A has voted an independent commit. Instead of joining T1', it can start a new independent transaction T'$_I^{(T1)}$.

In the worst case, if T1 cannot commit, the dependent look-ahead transactions must be aborted. In case of independent look-ahead transactions, different solutions are proposed in section 8.8, page 69. This joining rule assures that only threads that were participating in the former transaction are affected.

Restrictive Join

- Former participant threads can join any active look-ahead transaction.
- Regular threads are not allowed to join look-ahead transactions.

The main difference here is that any look-ahead threads can join any look-ahead transaction.

The main reason why we differentiate between self-restrictive join and restrictive-join is that in this model, in case of abortion of the former transaction, not only former participant threads must be informed, but also every look-ahead thread that took part in any of the former transaction's look-ahead transactions.

8.4.2 Open Set

The open set relaxes the joining rules as much as possible. Again, there are two slightly different models: one-way permissive and two-way permissive.

One-Way Permissive

- Any thread is allowed to join a look-ahead transaction.
- Look-ahead threads are only allowed to join other look-ahead transactions.

Figure 8.3 shows an example of execution, where one-way permissive joining rules are applied.

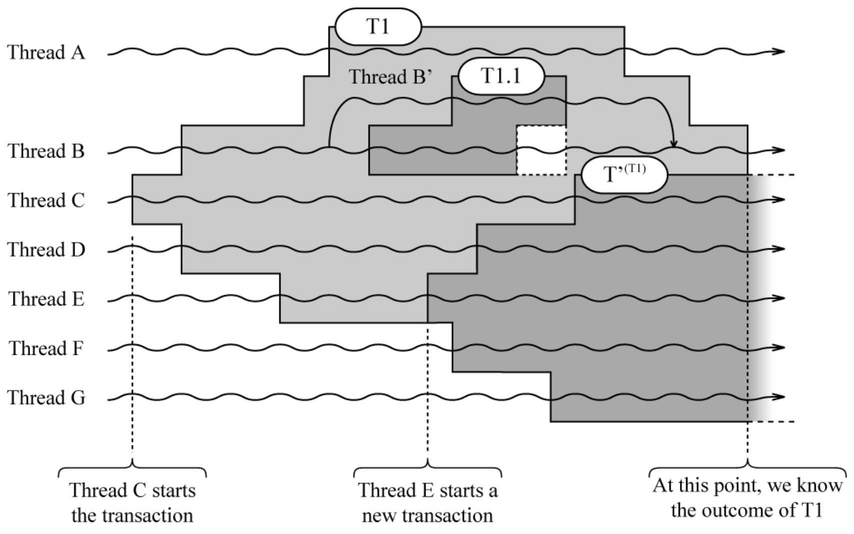

T1
Thread A
Thread B'
T1.1
Thread B
T'(T1)
Thread C
Thread D
Thread E
Thread F
Thread G
Thread C starts the transaction
Thread E starts a new transaction
At this point, we know the outcome of T1

Figure 8.3 : One-Way Permissive Joining Situation

One advantage of these rules is that look-ahead threads cannot join regular transactions that have started a long time ago, thus creating a dependency link. Of course, these rules can be adjusted in order to only allow look-ahead threads to join look-ahead transactions that were created by former participant threads, i.e. former participants threads of T1 cannot join any $T'^{(T2)}$, but only $T^{i(T1)}$.

Two-Way Permissive

- Let any thread join any transaction (whether it was a look-ahead one or not). An outside thread can join a look-ahead transaction and a look-ahead thread can join any regular transaction.

Letting any thread join any transaction is very permissive. However, adding a link between transactions can be very inefficient, as shown in Figure 8.4.

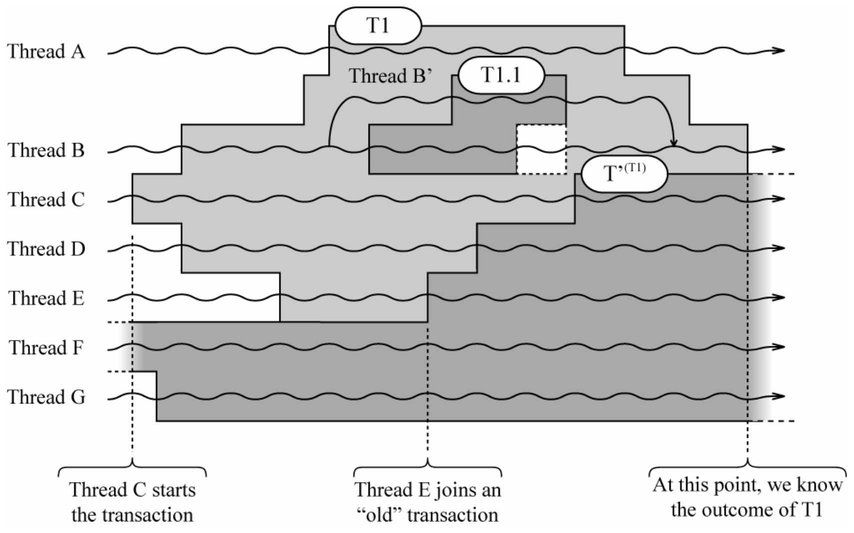

T1
Thread A
Thread B'
T1.1
Thread B
T'(T1)
Thread C
Thread D
Thread E
Thread F
Thread G
Thread C starts the transaction
Thread E joins an "old" transaction
At this point, we know the outcome of T1

Figure 8.4 : Two-Way Permissive Situation

Threads C, D and E decide to join a transaction created a long time ago (for instance). As joining it creates a dependency link between them, which means that if T1 aborts, we have to abort $T'^{(T1)}$ which might have started a long time ago.

This model can even lead to retroactive cascading abort as shown in Figure 8.5.

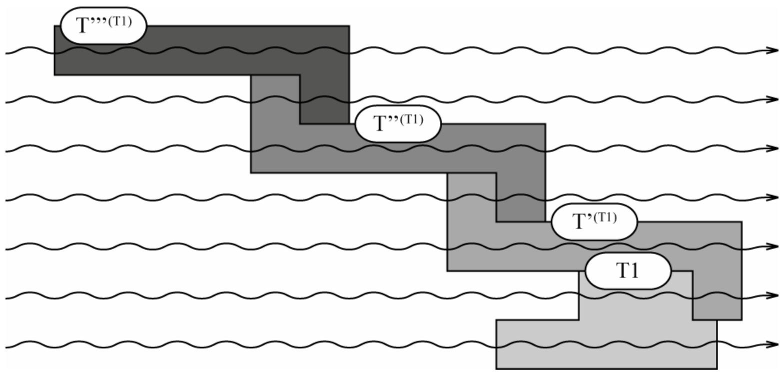


Figure 8.5 : Retroactive Cascading Aborts

T1, which was created a long time after T'''$^{(T1)}$ gets priority over it, and even worse, T'''$^{(T1)}$ is not be able to commit before T1.

A solution would be letting the threads only join look-ahead transactions (restrictive), or transactions created after a specific timestamp (T1's creation, in example).

8.5 *Ending a Look-Ahead Transaction*

Ending a look-ahead transaction or a regular transaction is done in the same way, with the normal `commit` statement. Since dependent look-ahead transactions can never commit before their former transactions, the system's response to this command must respect the rules given in section 4.5, page 32, about short look-ahead threads. In case of independent look-ahead transactions, the system should allow their commit vote to be executed out of order (section 8.8, page 69).

When the former transaction commits later on, every pending look-ahead transaction can commit and all other active look-ahead transactions are, from this point, behaving as regular transactions.

8.6 *Notes on Spawning Threads*

In the "Create" model, nothing has to be changed when dealing with spawning threads. The initial rules defining thread creation and termination must be followed.

8.7 *Concurrency Control Analysis*

In contrary with the previous models, the "Create" model only deals with look-ahead transactions, being possibly aborted at any time during their execution. Thus, whenever a read/write dependency occurs, the solutions given for each type of concurrency control (Chapter 5 : Look-Ahead & Concurrency Controls, page 39) can be used and applied. When a read/write dependency happens, the former transaction is not aborted, but only the erroneous look-ahead transactions. This is a great improvement over the previous models: the former transaction’s work, which is supposed to be correct, is never in danger because of look-ahead threads.

8.8 *Exception Handling*

In case of an exception raised in the former transaction while other threads are looking-ahead, the thread involved first tries to handle the internal exception, and propagates, if needed, an external exception to the containing context, raising a `LA_Transaction_Abort` in every former participant thread. The situation is shown in Figure 8.6.

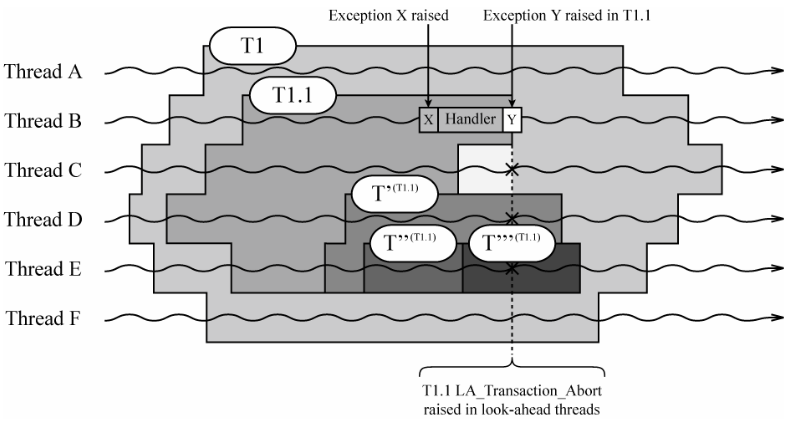


Figure 8.6 : Exception Handling in the "Create" Model

Thread B, executing code in the former transaction T1.1 raises an exception X, which it tries to handle, and then propagates an exception Y to the context of T1. Thus, the system raises T1.1 `LA_Transaction_Abort` in every former look-ahead thread. Thread C, having committed normally receives a regular T1.1 `Transaction_Abort`, as it is not looking-ahead. Thread D and E, in contrary, have now different choices, depending on whether the look-ahead transactions are dependent or independent.

If the former transaction aborts and has several look-ahead transactions, every dependent one must abort. In case of independent look-ahead transactions, the exception must not interrupt the independent look-ahead transaction until completion. When the exception is raised in the concerned threads context, the following possibilities are:

- To handle the exception.
- To forward the exception asynchronously to a specific handler in order to perform clean-up code,[1] thus not aborting the current execution.

In case the number of look-ahead transactions before the exception raised is greater than one (as in Figure 8.6), the system must assure the propagation of the exception from former transactions to their look-ahead takes place, assuring that

[1] Asynchronous handling of exception is not supported in current concurrent programming languages, yet. See section 10.4, page 82 for more information.

T’’$^{(T1.1)}$ receives the information from T’$^{(T1)}$. In Figure 8.6, T’’$^{(T1.1)}$ is informed by T’$^{(T1)}$, and must abort, in case it is a dependent look-ahead transaction.

8.9 *Nesting*

Up to now, a programmer, when voting commit, must immediately start or join a transaction at the same level. It is not possible to commit a transaction and its parent, and then start or join a transaction at a higher level. In order to support this, new functionalities must be provided.

There are two choices:

- Add a new `speculative_commit` statement. Several statements would follow each other in order to retrieve a higher level of nesting. Then immediately use create or join statements.
- A nesting level parameter is passed to the creation of a look-ahead transaction, i.e. `independent_commit_create(2)` would create a top-level independent transaction from a three level of nesting former transaction.

In case of semantically dependent transactions, the dependency link is inherited: if a T1.1.1 transaction wants to create a top-level, respectively level 2 dependent look-ahead transaction, the transaction is then T’$_D^{(T1)}$, respectively T’$_D^{(T1.1)}$.

Finally, we say that nesting, if it is allowed in an implementation, does not introduce any new problems in the “Create” model.

8.10 *Specific Limitations*

The limitation of this model is that it can only handle look-ahead transactions, not lone code. The transparency requirement is therefore not fulfilled, even without the new statements needed to support nesting. Compared to the previous models, the overall performance of “Create” is good, in case of several transactions following each other, and particularly in case of conflicting ones.

8.11 *Discussion*

A great improvement in this model is the fact that the former transaction does not abort anymore in case of conflicts with look-ahead. However, it might be too specific to be used and should be combined with one of the previous ones for a complete solution.

8.12 *Variations*

Some modifications on the current model are possible, in order to improve its performance or to decrease its design complexity.

8.12.1 New statements

In order to fulfill both the transparency and compatibility requirements strictly, dependent and independent statements could be regrouped, not distinguishing independent and dependent transactions:

- `commit_create`
- `commit_join`

If the differentiation between independent and dependent transactions turns out to be too complex to design and implement, this simple solution can be used. This would also result in an easier exception handling technique, always handling independent look-ahead transactions as dependent ones.

8.12.2 Complementary Use

The “Create” model could be used in combination with the “Leave” model, so that both look-ahead transactions and lone code is allowed. In this case, no changes have to be made for spawning threads, as both models do not introduce transaction borders’ modifications.

On the other hand, if the “Stretch” model handles lone code and the “Create” model the look-ahead transactions, attention must be paid with spawning threads, exactly as explained while introducing the “Stretch” model.

Chapter 9 : Implicit

9.1 *Concept*

The “Implicit” model is based on the “Create” model, taking its great advantage of never aborting the former transaction in case of conflict with a look-ahead transaction. We want to extend this feature in never aborting the former transaction in case of conflicts with lone code.

The main idea is to encapsulate the lone code automatically within an implicit transaction. Doing so, the lone code becomes easily recoverable and therefore, whenever a conflict occurs, the solutions given in Chapter 5 : Look-Ahead & Concurrency Controls can be used. Therefore, the “Implicit” model can be viewed as a modification to the “Create” model, adding support for lone code.

The implicit transactions are not visible to the user, in order to reach an optimal level of transparency. The end of an implicit transaction is defined by statements of creation or joining or by a commit of the former transaction or by committing the parent transaction. In the “Create” model, if the look-ahead transaction is still running when the former transaction completes, it becomes a regular transaction. The same applies here to lone code: if no transaction is created or joined before the former transaction outcome, the borders of the artificial implicit transactions simply disappear.

The general concept of the model is illustrated in Figure 9.1.

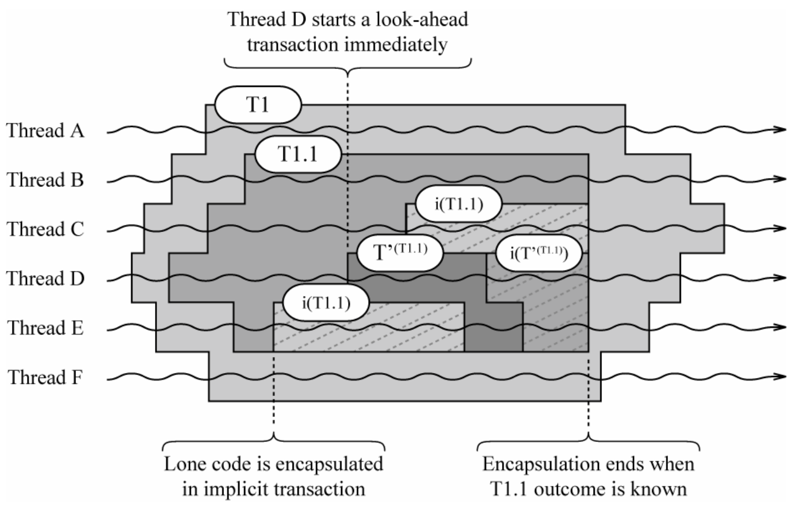

Thread D starts a look-ahead transaction immediately
Thread A
Thread B
Thread C
Thread D
Thread E
Thread F
T1
T1.1
i(T1.1)
T'(T1.1)
i(T'(T1.1))
i(T1.1)
Lone code is encapsulated in implicit transaction
Encapsulation ends when T1.1 outcome is known

Figure 9.1 : Illustration of the "Implicit" Model

An artificial transaction i(T1.1) is automatically created when thread E commits and looks ahead.

Thread D, when looking-ahead from T1.1 immediately creates a look-ahead transaction T'$^{(T1.1)}$ – thus not needing an artificial transaction – which thread E joins.

At this moment, both i(T1.1) and T'$^{(T1.1)}$, being short look-ahead transactions, are still pending, waiting for T1.1 outcome, to be committed.

Thread C, when committing joins the implicit transaction i(T1.1). Doing so, both threads C and E are allowed to cooperate as if they were in T1 but still isolated from the outside threads A and F. This prevents any potentially erroneous execution in the containing transaction T1 until the former transaction T1.1 is completed.

As soon as thread D looks ahead from T'$^{(T1.1)}$, it creates the artificial transaction i(T'$^{(T1.1)}$). The reason for it not to join i(T1.1) is that the look-ahead depends on T'$^{(T1.1)}$ and not T1.1. At this point, if thread D was allowed to cooperate in i(T1.1), and T'$^{(T1.1)}$ aborts, i(T1.1) would also need to abort – possible erroneous cooperation. Another reason for thread E not to join transaction i(T1.1) is that it would transgress the no-rejoining rule: thread E creates i(T1.1), joins T'(T1.1) and re-joins i(T1.1), there is a cyclic dependency, and none of the transactions would be able to commit. For more information about the re-joining problem, please refer to section 8.4, page 64.

9.2 *New Statements*

In the "Implicit" model, no statements specific to the model are needed. If the user needs synchronization amongst threads and transactions, the `wait-for-commit` must be used, but this is not specific to the model, but to allowing looking-ahead.

No statements are needed to define the borders of the implicit transactions, as we want the model to be as transparent as possible to the user.

If a thread of a former transaction looks ahead, an artificial transaction is automatically created (or joined) to encapsulate every operation following the commit statement. There is one implicit transaction for every former transaction possible.

Therefore, if several threads look ahead and execute lone code, they are all able to cooperate with each other, inside the same implicit transaction.

As soon as a thread creates or joins a look-ahead transaction, they are not allowed to re-join an implicit transaction. If they look-ahead from the look-ahead transaction, a new implicit transaction is created or joined.

The borders of an implicit transaction are defined by the creation or joining of a look-ahead transaction or by committing the parent transaction. In any of these cases, the look-ahead is considered as short, and the solutions of section 4.5, page 32 are applied.

9.2.1 Notes on Spawning Threads

We have to give simple rules for the model, in case of willing to create or terminate a thread inside an implicit transaction.

- Creation of a thread is blocked until the implicit transaction is committed.
- Termination of a thread is blocked until the implicit transaction is committed.

These two rules avoid the semantic problems that arise when a thread creates or joins an implicit transaction and terminate inside it, respectively when a new thread is created inside an implicit transaction and leaves it. The issue here is what happens to these threads if the implicit transaction aborts (i.e. in case of conflicts with the former transaction). If the all-or-nothing semantics of transaction is applied to threads, such participants would have to be re-created, respectively killed.

The only solution to this issue is to block the creation or termination of thread until the former transaction outcome is known. Doing so, in case of conflicting read/write dependency between an implicit transaction and one of its former transaction, the implicit transaction can be undone since it does not contain the creation or termination of threads. This is shown in Figure 9.2.

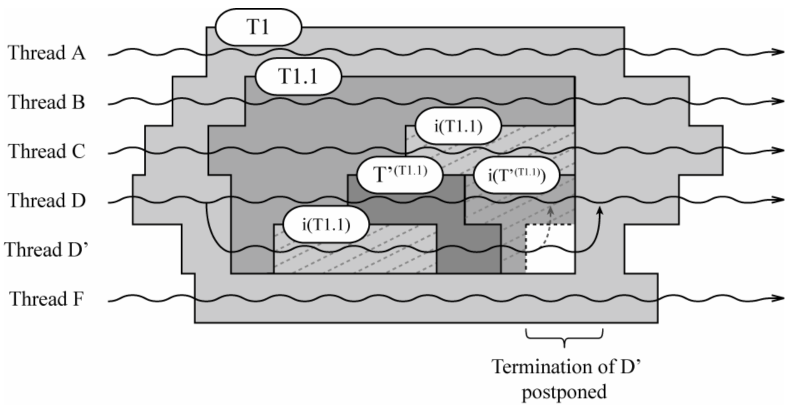


Figure 9.2 : Terminating Threads is Postponed in "Implicit" Model

Thread D' is forked in T1. It then joins T1.1 and looks ahead from it. Its work is thus encapsulated in i(T1.1) and it finally joins T'$^{(T1.1)}$. At this moment, when it looks-ahead from T'$^{(T1.1)}$ it executes code inside T1 and wants to terminate. However, the thread is still in a look-ahead context. Even if its former transaction T'$^{(T1)}$ is ready to commit, the root former transaction T1.1 is not. Therefore, the termination of the thread is postponed until the implicit transaction has committed.

9.3 *Concurrency Control Analysis*

The concurrency control of the "Implicit" model acts exactly as the "Create" model. Whenever a conflicting operation is found during the look-ahead execution, it is aborted, allowing the former transaction to continue normally.

9.4 *Exception Handling*

In case of an exception raised in the former transaction while threads are looking-ahead, the thread involved first tries to handle the internal exception, and propagates, if needed, an external exception to the containing context, raising a `LA_Transaction_Abort` in every former participant thread. The situation is shown in Figure 9.3.

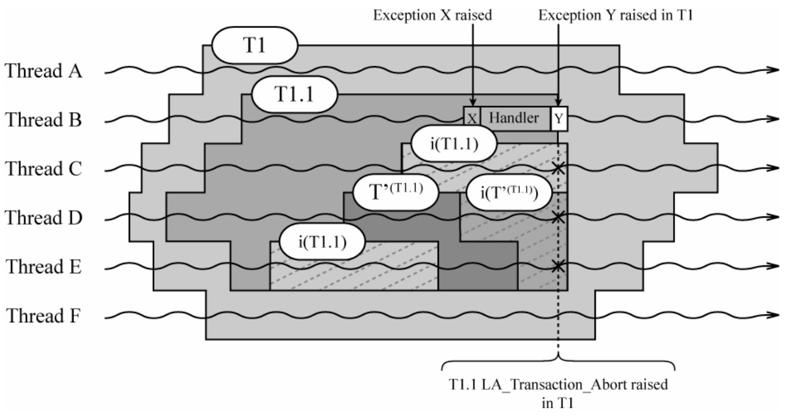


Figure 9.3 : Exception Handling in the "Implicit" Model

Thread B, executing code in the former transaction T1.1 raises an exception X, which it tries to handle, and then propagates an exception Y to the context of T1. Thus, it raises T1.1 `LA_Transaction_Abort` in every look-ahead thread. Please note that there are no handlers inside the implicit transactions, being invisible to the users. The only ones that can handle the exceptions are declared in T1.

Exactly as in the "Create" model in case of dependent transactions, if the former transaction T1.1 aborts, every look-ahead transaction, including the implicit ones, abort.

9.5 *Nesting*

If the programmer wants to allow a thread to look-ahead from a level three nested transaction (i.e. T1.1.1) to a top-level one (i.e. T2), it would `commit` twice, to be inside the context of T1, and at this moment, it would create or join a new transaction. As being totally transparent, nothing has to be done from a user point of view. Therefore, no problems due to allowing nested look-ahead possibilities appear.

9.6 *Specific Limitations*

This model, taking the best from the “Create” and “Stretch” models does not have limitations from a user point of view, except that handling exceptions for the user can be difficult when threads look-ahead from former transactions of different nesting level.

The described model does not differentiate between semantically dependent and independent ones, hence, modifications could be added, as exposed in section 9.8.

9.7 *Discussion*

The “Implicit” model can be enabled in any context, managing look-ahead transactions as well as lone code. Read/write dependencies between the former transaction and lone code are finally handled at its best, and exception handling is not more complicated as in the previous models.

9.8 *Variations*

This section contains modifications to the “Implicit” model described previously, adding new functionalities.

9.8.1 Semantic Dependency Information Retrieval

The “Implicit” model could be modified introducing new statements to retrieve information on the semantic dependency between look-ahead threads. This was exposed in section 8.2, page 64, and can be easily extended to lone code, with the following new `independent_commit` and `dependent_commit` statements.

Part III

Beyond

Chapter 10 : Future Improvements

This chapter indicates several directions for future research, in both the open multithreaded transactions model and looking-ahead theory.

10.1 *Cooperative Concurrency*

When dealing with cooperative concurrency, programming guidelines must be provided to the user, helping him to find a not only correct, but also efficient way of executing operations. Programming approaches with nested transactions and cooperative concurrency are complementary, but rules on when to use nested transactions and when to only use cooperative concurrency would help the user doing efficient programs, thus preventing bad surprises when allowing look-ahead threads.

10.2 *Nested Transactions: Concurrency Control*

Section 2.5, page 9, states that nested transactions must be isolated from their parent transactions. In order to assure this isolation, plenty of different concurrency control can be chosen. As exposed in section 4.2, page 29, read/write dependencies between transactions can lead to abort at least one transaction. In the case of nested transaction as in the look-ahead ones, it is necessary never to abort the containing one instead of the nested one – if aborting the parent one, the nested one must also abort. Therefore, we propose to give priority to the parent transaction over its child, exactly as done in look-ahead transactions when giving priority to the former transaction over its look-ahead transactions.

10.3 *Looking-Ahead Dedicated Concurrency Control*

As described in section 2.5, page 9, a lot of concurrency controls exist, even if only some of them are really used in practice. Having exposed every problem that can happen when looking-ahead, we want now to give hints on a dedicated concurrency control, specific to looking-ahead.

We have stated that looking-ahead is only useful when being optimistic about the issue of the former transaction, and therefore, we want to continue in our optimism to expose rules on a new dedicated concurrency control.

The concurrency control proposed is asymmetric: it gives priority to the former transaction, always aborting its look-ahead transactions (as exposed in the "Implicit" model, look-ahead lone code can be transformed in transactions).

The idea is the following:

- The former transaction is always completely isolated from its look-ahead transactions. If the look-ahead causes a conflict, it is aborted/restarted.
- A look-ahead transaction always reads the last version of the object. If it is a third level look-ahead, it reads the last version of the second level look-ahead, which reads the last version of the first level look-ahead, which finally reads the last version of the former transaction. If in-between two read operations, the version of the object has changed, every look-ahead depending on this version is aborted/restarted.

These rules define an optimistic concurrency control with asymmetric multi-version timestamp ordering, similar to the one given as solution for read/write dependencies.

What could be a real improvement is how to handle accesses between look-ahead transactions that have different former transactions and root former transactions. Rules could state that any look-ahead transaction reads the very latest version of an object, no matter what root former transaction it is connected to, or reads the very last version its former transaction has access to.

Research in this topic might be also very useful in case of nested transactions, as both conflicts resolutions are similar: always give priority to a specific transaction (parent or former).

10.4 *Asynchronous Handling*

As exposed while describing semantic dependency, an independent transaction or lone code can be very interesting when dealing with a model that manages out of order execution. Unfortunately, asynchronous handling of exception, that could allow an independent look-ahead thread to continue its execution as if nothing happened, is not yet implemented [KO02] in the concurrent programming language that is used in the OPTIMA Framework.

It could be therefore possible to deal with asynchronous handling in the OPTIMA Framework itself, adding for every possible independent thread a listener thread that would be used to handle the forwarded exception.

Chapter 11 : Conclusion

The classic open multithreaded transaction model prescribes synchronous participant exit, i.e. participants that have finished their work inside a transaction and vote commit are blocked until the outcome of the transaction is known. This leads to wasted time, especially in loosely coupled systems with long-running transactions.

This diploma thesis gives a complete overview of how the classic open multithreaded transaction model can be extended to avoid synchronizing participants upon transaction exit. Following an idea that was proposed in the context of atomic actions, participant threads are allowed to look-ahead from a transaction, and continue executing the following operations.

Four models have been laid out in detail:

The “Leave” model, simple and straightforward, is an almost direct transposition of the model used in the atomic action context. It provides the same guarantees than the original model and is transparent for the programmer. Because of its simplicity, though, exceptional situations are handled in a very inefficient way.

The “Stretch” model improves the “Leave” model by extending the transaction border until all participants have voted commit.

Unfortunately, it cannot distinguish between look-ahead transactions and lone code. Therefore, it cannot take advantage of concurrency control specific handling of conflicts.

The “Create” model forces the programmer to either immediately create or immediately join a new transaction after looking ahead. Hence, it is not transparent for the application programmer: it requires look-ahead specific application design. On the other hand, thanks to the absence of lone code, it can resolve conflicts in a flexible way.

Finally, the “Implicit” model takes the best from every other model. It transparently encapsulates lone code in transactions. It is the most powerful model, because it is completely transparent for the application programmer, and at the same time, it can use the most flexible conflict solving strategies.

Part IV

Annexes

Annex A : Bibliography

[BBP93] A. Bestavros, S. Braoudakis, E. Panagos, “Performance Evaluation of Two-shadow Speculative Concurrency Control”, *Technical Report BUCS-TR-1993-001*, Computer Science Department, Boston University, January 1993.

[BG83] P. A. Bernstein , N. Goodman, “Multiversion Concurrency Control – Theory and Algorithms”, *ACM Transactions on Database Systems (TODS)*, v. 8 n. 4, p. 465-483, December 1983.

[BGH87] P. A. Bernstein , V. Hadzilacos , N. Goodman, “Concurrency Control and Recovery in Database Systems”, *Addison-Wesley Longman Publishing Co.*, 1987.

[BR92] B. R. Badrinath , K. Ramamritham, “Semantics-Based Concurrency Control: Beyond Commutativity”, *ACM Transactions on Database Systems (TODS)*, v. 17 n. 1, p. 163-199, March 1992.

[EGLT76] K. P. Eswaran , J. N. Gray , R. A. Lorie , I. L. Traiger, “The Notions of Consistency and Predicate Locks in a Database System”, *Communications of the ACM*, v. 19 n. 11, p. 624-633, November 1976.

[Her90] M. Herlihy, “Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types”, *ACM Transactions on Database Systems (TODS)*, v. 15 n. 1, p. 96-124, March 1990.

[JM86] D. R. Jefferson, A. Motro, “The Time Warp Mechanism for Database Concurrency Control”, *Proceedings of the Second IEEE International Conference on Data Engineering*, p. 474-481, February 5-7, 1986, Los Angeles.

[Kie00] J. Kienzle, “Exception Handling in Open Multithreaded Transactions”, *ECOOP Workshop on Exception Handling in Object-Oriented Systems*, Cannes, June 2000.

[Kie04] J. Kienzle, “Open Multithreaded Transactions, A Transaction Model for Concurrent Object-Oriented Programming”, *Kluwer Academic Publishers*, 2004.

[KO02] A. W. Keen, R. A. Olsson, “Exception Handling During Asynchronous Method Invocation”, *Euro-Par 2002 Parallel Processing, Lecture Notes in Computer Science Series*, Number 2400, Springer-Verlag, B. Monien and R. Feldmann, Editors. 656-660, August 2002.

[KR81] H. T. Kung , J. T. Robinson, “On Optimistic Methods for Concurrency Control”, *ACM Transactions on Database Systems (TODS)*, v.6 n.2, p.213-226, June 1981.

[LA93] H. V. Leong, D. Agrawal. “Semantics-Based Time Warp Protocols”, *Technical Report TRCS93-10*, Department of Computer Science, University of California, Santa Barbara, 1993.

[Lom77] D. B. Lomet, “Process Structuring, Synchronization, and Recovery Using Atomic Actions”, *Proceedings of an ACM Conference on Language Design for Reliable Software*, p. 128-137, March 28-30, 1977, Raleigh.

[Ran75] B. Randell, “System Structure for Software Fault Tolerance”, *Proceedings of the International Conference on Reliable Software*, p. 437-449, April 21-23, 1975, Los Angeles.

[Ree83] D. P. Reed, “Implementing Atomic Actions on Decentralized Data”, *ACM Transactions on Computer Systems (TOCS)*, v. 1 n. 1, p. 3-23, February 1983.

[Rom01] A. Romanovsky, “Looking Ahead in Atomic Actions with Exception Handling”, *20th Symposium on Reliable Distributed Systems (SRDS 2001)*, p. 142-151, October 21-28, 2001, New Orleans.

[Sil82] A. Silberschatz, “A Multi-Version Concurrency Scheme with no Rollbacks”, *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, p. 216-223, August 18-20, 1982, Ottawa.

Annex B : Index

A

ACID Properties 3
- Atomicity 3
- Consistency 3
- Durability 3
- Isolation 3
- Persistence 3
- Serializability 3

Atomic Actions 21
- Looking-Ahead 23
- Synchronization on Exit 22

Atomicity 3

B

Backward Commutativity 18
Backward Validation 15
Broadcast Commit 15

C

Commutativity 17
Compatibility 24
Concurrency Control
- Lock-Based Protocols 11
- Multiversion Timestamping 13
- Optimistic 14
- Pessimistic 10
- Semantic-Based Concurrency Control 16
- Time Warp Protocols 13
- Timestamp-Ordering 12

Consistency 3

D

Durability 3

E

Exception Handling
- Pre-Emption 20

External Exceptions 19

F

Former Transaction 24
Forward Commutativity 17
Forward Validation 14
- Broadcast Commit 15

I

Internal Exceptions 19
Isolation 3

L

Lock-Based Protocols 11
Lone Code 26, 29
Look-Ahead Threads 24
Look-Ahead Transaction 27
Looking-Ahead 23, 24, 26
- Exception Handling 39
- Former Transaction 24
- Lone Code 26, 29
- Look-Ahead Threads 24
- Look-Ahead Transaction 26, 27
- Semantic Dependency 28
- Semantic Independency 28

M

Modifier 16
- Updater 16
- Writer 16

Multiversion Timestamping 13

O

Observer 16
- Reader 16

Open Multithreaded Transactions 7
- Concurrency Control 10
- Ending an Open Multithreaded Transaction 8
- Exception Handling 18
- Former Transaction 24
- Joined Participants 7
- Joining an Open Multithreaded Transaction 8
- Look-Ahead Threads 24
- Looking-Ahead 24, 26
- Participants 7
- Spawned Participants 8
- Starting an Open Multithreaded Transaction 8
- Synchronization on Exit 9

Optimistic Concurrency Control 14
- Backward Validation 15
- Forward Validation 14

Overall Speed Increase 24

P

Persistence 3
Pessimistic Concurrency Control 10

R

Read/Write Dependencies 31
- Read-After-Read (RAR) 32
- Read-After-Write (RAW) 33
- Write-After-Read (WAR) 32
- Write-After-Write (WAW) 33

Read-After-Read (RAR) 32
Read-After-Write (RAW) 33
Reader 16
Requirements 24
- Compatibility 24
- Overal Speed Increase 24
- Robustness 24
- Transparency 24

Robustness 24

S

Semantic Dependency 28
Semantic Independency....................................28
Semantic-Based Concurrency Control 16
    Backward Commutativity............................18
    Forward Commutativity...............................17
Serializability..3
Synchronization on Exit 9, 22

T

Time Warp Protocols 13
Timestamp Ordering 12
Transaction... 2
    ACID Properties.. 3
Transparency.. 24

W

Write-After-Read (WAR) 32
Write-After-Write (WAW) 33