Looking Ahead in Open Multithreaded Transactions

Maxime Monod[1], Jörg Kienzle[2], Alexander Romanovsky[3]
[1] Swiss Federal Institute of Technology in Lausanne (EPFL), Lausanne, Switzerland
[2] School of Computer Science, McGill University, Montreal, Canada
[3] University of Newcastle upon Tyne, Newcastle upon Tyne, United Kingdom
Maxime.Monod@epfl.ch, Joerg.Kienzle@mcgill.ca, Alexander.Romanovsky@newcastle.ac.uk

Abstract

Open multithreaded transactions constitute building blocks that allow a developer to design and structure the execution of complex distributed systems featuring cooperative and competitive concurrency in a reliable way. In this paper we describe an optimization to the standard open multithreaded transaction model that does not impose any participant synchronization when committing a transaction, but still provides the same execution semantics. This optimization – letting participants "look ahead" and continue their execution on the outside of the transaction – makes it possible to speed up the execution of in individual transaction with multiple participants tremendously. The paper describes all technical issues that had to be solved, e.g. adapting concurrency control of transactional objects to be look-ahead aware, adapting joining rules for look-ahead participants, and re-defining exception handling in the presence of look-ahead.

1. Introduction

Large-scale, distributed systems often give rise to complex concurrent and interacting activities, and are therefore extremely difficult to understand, design, analyze and modify. In addition, as distributed systems grow bigger, they are inevitably exposed to an increasing number of faults: hardware faults, faults in the environment, faults in the underlying middleware, and even software design faults in the system itself.

In data-centric applications, where multiple objects must usually be accessed or updated together to correctly reflect the real world, great care must be taken to keep related objects globally consistent. Any interruption of updates to objects, or the interleaving of updates and accesses, can break the overall consistency of an object system. In order to reduce the complexity inherent in concurrent systems and to provide a base for implementing fault tolerance, researchers have used advanced and elaborate concurrency features such as *transactions* [1] and *atomic actions* [2], in the design of distributed systems.

The *Open Multithreaded Transaction* model (OMTT) [3] is an advanced transaction model that has been introduced for the development of reliable, data-centric, distributed systems with loosely coupled, cooperative and competitive concurrent entities. OMTTs provide building blocks that allow a developer to reason about the design and the execution of a system more easily. At the same time, OMTTs act as error confinement regions that prevent corrupted application state from contaminating other parts of the system.

Unfortunately, the standard OMTT model also imposes very tight synchronization between concurrent collaborating entities that slows down the execution significantly. This paper describes how the synchronization constraints of the OMTT model can be relaxed by allowing the participants of a transaction to look ahead and continue executing on the outside of the transaction before the outcome of the transaction is known. As a result, participants do not have to wait for each other anymore, which reduces the overall execution time tremendously.

The paper is structured as follows. Section 2 introduces the original open multithreaded transaction model. Its limitations are detailed in section 3. The look-ahead optimization and all the technical issues that had to be solved are described in section 4. A prototype implementation for Java is discussed in Section 5. Section 6 reviews related work in this area, and the last section draws conclusions.

2. Open Multithreaded Transactions

Open Multithreaded Transactions [3] constitute building blocks that allow a developer to design and structure the execution of complex distributed systems featuring cooperative and competitive concurrency in a reliable way. Open multithreaded transactions, just as standard transactions, encapsulate a set of operations and give them the so-called ACID properties: *atomicity*, *consistency*, *isolation*

and *durability* [1]. The difference is that OMTTs provide features for controlling and structuring not only accesses to objects, as usual in transaction systems, but also processes taking part in transactions.

2.1. Participants

The OMTT model allows several threads or processes, here called *participants*, to enter the same transaction in order to perform a joint activity. The participants are loosely coupled, i.e. they progress independently, but may collaborate or share information by using the same objects. OMTTs support nesting, i.e. participants of a transaction can enter subtransactions. Participants can also spawn new threads, but these threads have to terminate inside the transaction in which they were created.

All participants that entered the transaction have to vote on the transaction outcome before leaving. If all participants are satisfied with the work they performed inside the transaction, then the transaction commits, i.e. the changes to the application state are made durable and visible to the outside world. However, if only one of the participants votes abort, then all changes are undone, as if the transaction never happened. To guarantee this atomicity and isolation, participants are only allowed to leave the open multithreaded transaction once its outcome has been determined. As a result, participants of a transaction that commits have to wait for the slowest of all participants to vote, and then *exit synchronously* [2].

The upper part of figure 1 shows an open multithreaded transaction $T1$ that is created by participant C. Participants A, B, and D join the transaction later on. The synchronous exit rules dictates that, although B, C, and D finish their work inside $T1$ early by voting commit, they are blocked until participant A also votes commit.

2.2. Transactional Objects

Within an open multithreaded transaction, participants can access a set of transactional objects. Although individual participants evolve independently inside an open multithreaded transaction, they are allowed to collaborate with other participants of the transaction by accessing the same transactional objects. The transactional objects therefore must *preserve data consistency* despite concurrent accesses from within a transaction (cooperative concurrency), and at the same time *provide isolation* among concurrent accesses from different transactions (competitive concurrency). Typically, this is done by a concurrency control component (see section 4.6) that monitors all access to a transactional object.

2.3. Exception Handling

Open multithreaded transactions have been designed for reliable system development by providing error confinement and fault tolerance capabilities. The model incorporates disciplined exception handling adapted to nested transactions. It allows individual participants to perform forward error recovery by handling an abnormal situation locally. If local handling fails, the transaction support applies backward error recovery and reverses the system to a consistent state.

The model distinguishes internal and external exceptions. The set of internal exceptions for each participant consists of all exceptions that might occur during its execution. There are three sources of exceptions inside an open multithreaded transaction:

- An internal exception can be raised explicitly by a participant;
- An external exception raised inside a nested transaction is raised as an internal exception in the parent transaction;
- Transactional objects accessed by a participant of a transaction can raise an exception to signal a situation that violates the consistency of the state of the transactional object. Objects that perform such checks are called *self-checking transactional objects* [4].

All these situations give rise to a possibly inconsistent application state. If a participant does not handle such a situation, the application's correct behavior can not be guaranteed. A programmer of a participant should therefore prepare handlers for all internal exceptions. However, if by mistake an internal exception is not handled, then the external exception *Transaction_Abort* is automatically signaled, and the application consistency is restored by aborting the transaction. Likewise, if any participant of a transaction explicitly signals an external exception, the transaction is aborted, the exception is propagated to the containing context, and the exception *Transaction_Abort* is signaled to all other participants. If several participants signal an external exception, each of them propagates its own exception to its own context.

3. Limitations of OMTTs and Look-Ahead

Using open multithreaded transactions to structure the execution of complex concurrent systems makes the systems easier to understand, and helps reasoning about error containment and other fault tolerance properties [5]. However, using open multithreaded transactions also results in a heavy run-time overhead.

To guarantee isolation, open multithreaded transactions require participants that finished working on behalf of a

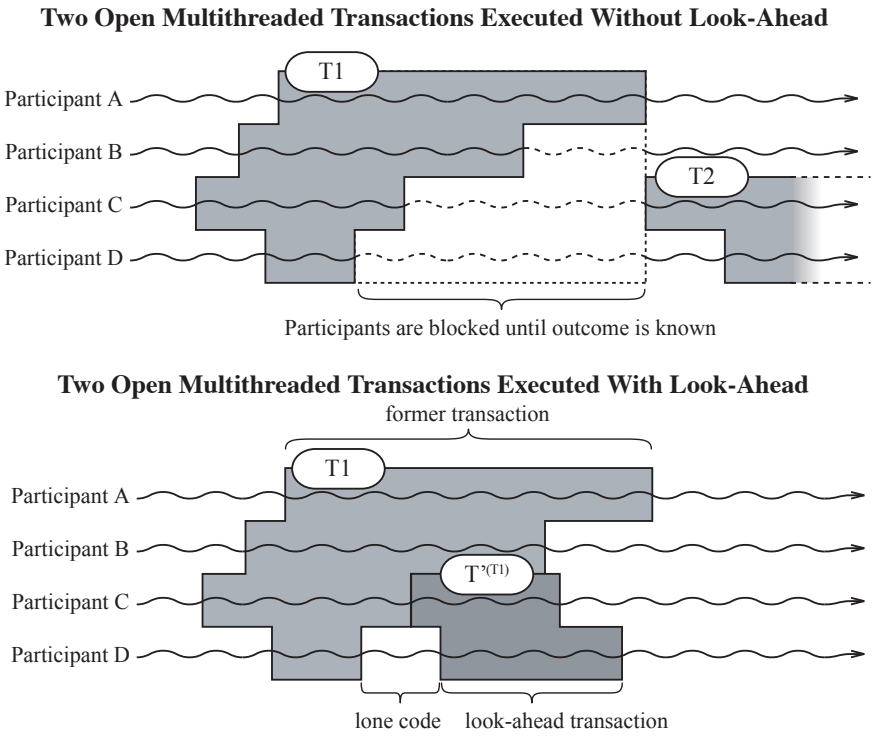

Figure 1. Standard vs. Look-ahead OMTTs

transaction to *wait* until all other participants of the same transaction have completed their work. Since in open multithreaded transactions the individual participants are only loosely coupled, they rarely complete their work at the same time. But still, all participants have to wait for the slowest one to finish. Especially for long running transactions, the blocking time of individual participants can be considerable. In [6], for example, a bidder participant in an online auction is blocked until the auction closes.

Following the idea suggested in [7] for the conversation scheme, we propose not to block participants of an open multithreaded transaction when they vote commit. Instead, they are allowed to *look ahead* from the transaction and start working on the outside, just as if the transaction had really committed. Looking ahead can be seen as optimistic transaction execution (see section 6).

Figure 1 illustrates the performance increase of look-ahead. The upper part of the figure shows two open multithreaded transactions $T1$ and $T2$ executed using the standard open multithreaded transaction model. Participants B, C and D are blocked until participant A finally votes commit. $T2$ is executed after $T1$. The lower part of Figure 1 shows how the same two transactions can be executed more efficiently if look ahead is allowed. $T2$, now named $T'^{(T1)}$, is started as soon as participant C finishes its work in $T1$.

Obviously, look ahead results in more efficient execution of open multithreaded transactions in case the *former transaction* ($T1$ in Figure 1) commits. However, if things go wrong, e.g. participant A decides to abort $T1$, then the execution of the *look-ahead transaction* $T'^{(T1)}$ is potentially erroneous too, since it was based on the assumption that the former transaction committed.

4. Look-Ahead OMTTs

The main issue with look-ahead is that in case the former transaction aborts, all operations executed by participants that looked ahead have to be undone too, because they were based on the assumption that the former transaction had committed. Unfortunately, only operations executed from within a transaction are undoable. There is no guarantee that a look-ahead participant immediately enters a new transaction when it exits the former one. The participant might execute *lone code* (see lower part of Figure 1), i.e. code that is not encapsulated within a separate transaction.

Other issues include dealing with nesting and joining of look-ahead transactions, creation and termination of participant threads, concurrency control and exception handling. The following subsections describe how our model addresses all these problems.

4.1. Dealing with Lone Code

We did not want to restrict looking ahead to participants which immediately start a new transaction. Therefore we decided to encapsulate the lone code of look-ahead participants in an *implicit transaction*, which is created at the moment the first participant looks ahead from a transaction. All subsequent operations of the look-ahead participant are done from within this implicit transaction. If ever another participant looks ahead from the former transaction, it also automatically joins the implicit one.

The implicit transaction behaves just like a normal transaction. It isolates look-ahead participants from the outside world, i.e. it prevents them to communicate results of the former transaction to others. This is good, since this prevents cascading aborts in case the former transaction aborts. The special thing about implicit transactions is that their boundaries are not known to the developer. Even if all look-ahead participants start or enter new transactions, the implicit transaction is not committed. It only ends when the former transaction commits.

Figure 2 illustrates the idea of implicit transactions. When participant E looks ahead of transactions $T1.1$, an implicit transaction $i(T1.1)$ is created. When participant D leaves $T1.1$, it immediately creates a look-ahead transaction $T'^{(T1.1)}$. The implicit transaction $i(T1.1)$ continues to exist, even when participant E joins $T'^{(T1.1)}$. It can't commit, because the outcome of $T1.1$ is not known yet. Later, when C looks ahead from $T1.1$, it joins the same $i(T1.1)$ again. In the mean time, when E looks ahead from $T'^{(T1.1)}$, a new implicit transaction $i(T'^{(T1.1)})$ is created, which ends as soon as participant D finishes $T'^{(T1.1)}$. Finally, D and E also join $i(T1.1)$, which commits and ceases to exist as soon as $T1.1$ commits.

4.2. Short Look-Ahead Transactions

Since operations made by look-ahead participants are only valid if the former transaction commits, we cannot allow look ahead transactions or implicit transactions to commit before the former transaction commits. Look-ahead transactions that are completed are therefore put on a commit queue and committed as soon as the former transaction commits.

4.3. Spawned Participants

The creation and termination of threads in open multithreaded transactions is restricted (see section 2.1): any participant joining an open multithreaded transaction has to leave as well, and participants created inside a transaction must also terminate inside it. The main reason for this restriction is that recreation of threads is very tricky. If participants would be allowed to enter a transaction from the outside and then terminate inside, an abortion of the transaction after the thread has terminated would require the recreation of the thread.

OMTTs do not restrict the creation and termination of threads in lone code. However, we encapsulate lone code executed by look-ahead participants within an implicit transaction. When lone code contains thread creation or termination operations, we therefore run into similar problems than the original model. Assuming that thread creation and termination is not used that often, we suggest to block creation and termination operations of look-ahead participants executing lone code until the outcome of the former transaction is known.

4.4. Nesting

Open multithreaded transactions can be nested. Our look-ahead scheme maintains this flexibility, i.e. it supports look-ahead between parent and child transactions. Multilevel look ahead is possible; our model keeps a list of look-ahead dependencies associated with each transaction. In case a former transaction aborts, all implicit and explicit look-ahead transactions are aborted. Since implicit transactions are created to encapsulate lone code, looking ahead from the top-level transactions can be supported as well.

4.5. Joining and Cascading Aborts

After looking ahead from a transaction, a participant may decide to join any already existing transaction. In case of look-ahead, this rule might lead to some complications.

Circular dependencies could arise if a look-ahead participant is allowed to re-join a former transaction. Re-joins are therefore forbidden. This is not a severe restriction, because such a situation is impossible to create in the standard model in the first place.

To avoid *cascading aborts*, look-ahead participants that try to join non look-ahead transactions are blocked until the outcome of the former transaction is known. If look-ahead participants would be allowed to join any ongoing transaction, the joint transaction would *become* a look-ahead transaction as a result. This problem is illustrated in Figure 3. $T1$ becomes $T'^{(T2)}$ once participant B looks ahead from $T2$ and joins $T1$. Likewise, $T2$ becomes $T'^{(T3)}$ when participant D looks ahead from $T3$ and joins $T2$. As a result, an abort of $T3$ triggers the abort of $T2$ and $T1$, which are transactions that started *earlier* than T3!

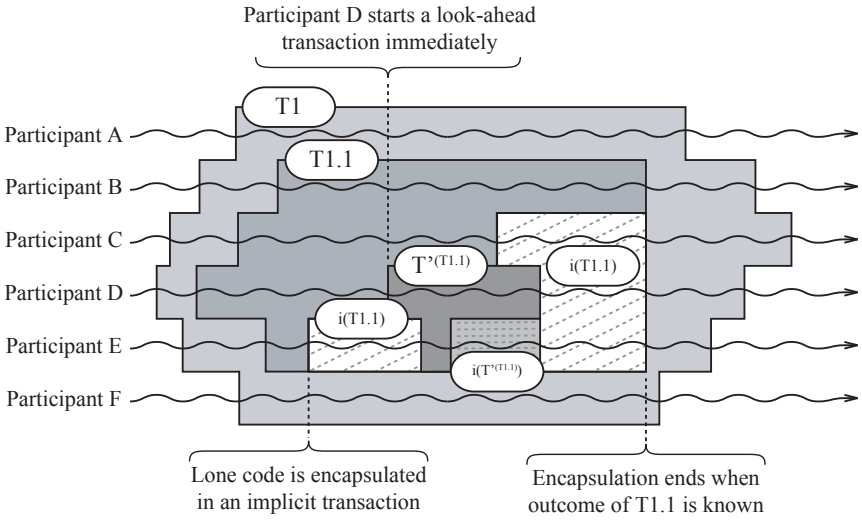

Figure 2. Implicit Transactions Hide Lone Code

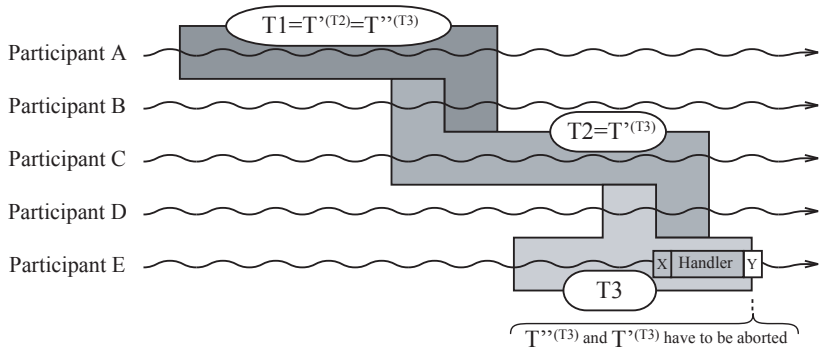

Figure 3. Cascading Aborts Due to Uncontrolled Joins

4.6. Resource Dependencies and Concurrency Control

Access to resources, i.e. transactional objects, is the most important issue that needs to be addressed when allowing look-ahead in OMTTs. In the standard model, where all participants wait until the outcome of the transaction is known, participants leaving a transaction and then entering a new one can safely access the same transactional objects. The problem arises when look-ahead is enabled, because a look-ahead transaction might use the same objects *while* or even *before* the former transaction uses them.

Figure 4 illustrates the situation. The look-ahead transaction $T'^{(T1)}$ invokes operation A on transactional object O, and subsequently the former transaction $T1$ invokes operation B on the same object. But since operation A was invoked from within the look-ahead transaction $T'^{(T1)}$, it semantically depends on the successful execution of the former transaction $T1$, which includes the execution of operation B. In order to get a correct deadlock- and starvation-free execution of look-ahead OMTTs, the concurrency control associated with a transactional object must be aware of this additional dependency.

Concurrency control comes in two flavors: *pessimistic* (conservative) or *optimistic* (aggressive), both having advantages and disadvantages. The principle underlying pessimistic concurrency control schemes is that, before attempting to perform an operation on any transactional object, a transaction has to get permission to do so. If a transaction invokes an operation that causes a conflict, the transaction is blocked or aborted. This can lead to deadlock situations. On the other hand, any transaction that successfully runs to completion is guaranteed to commit. The following section describes how the widely used pessimistic lock-based concurrency control can be adapted to support look-ahead.

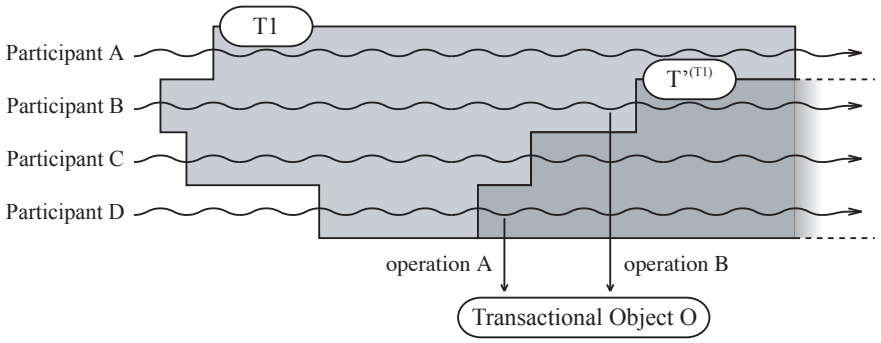

Figure 4. Object Dependencies

Lock-based Pessimistic Concurrency Control Lock-based concurrency control protocols use locks to implement permissions to perform operations. When invoking an operation on a transactional object, the participant must first request the lock associated with this operation 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 indeed would be a conflict, the participant requesting the lock is blocked, waiting for the release of the conflicting lock. Otherwise, the lock is granted, and the participant may proceed and execute the operation. The order in which locks are granted to transactions imposes an execution ordering on the transactions with respect to their conflicting operations. Two-phase locking [8] ensures serializability by not allowing transactions to acquire any more locks after they released a lock. This implies in practice that a transaction acquires locks during its execution (1st phase), and releases them at the end once the outcome of the transaction has been determined (2nd phase).

It turns out that the standard lock-based protocol cannot be used in the presence of look-ahead without causing deadlocks. For example, the look-ahead transaction in Figure 4 might acquire a write lock on the transactional object O when invoking operation A. As a result, the former transaction blocks when it wants to execute operation B, waiting for $T'^{(T1)}$ to complete. However, the look-ahead transaction $T'^{(T1)}$ can only commit after $T1$, because it has been executed with the assumption that $T1$ committed (see section 4.2). This circular dependency creates a deadlock.

The only solution is to make the lock-based protocol look-ahead aware. If ever a transaction wants to acquire a resource whose lock is held by one of its look-ahead transactions, then the lock should be granted anyway, thus invalidating the look-ahead transaction, which must be aborted (and restarted). Of course, this modification changes the nature of the lock-based protocol. In a strict sense the protocol is not pessimistic anymore : after granting permission to access an object to a look-ahead transactions, the permission might be revoked again. [9] talks about similar ideas in the context of flexible locking for monitor objects in real-time systems.

Optimistic Concurrency Control In optimistic concurrency control schemes [10], transactions are allowed to perform conflicting operations on objects without being blocked, but when they attempt to commit, transactions are validated to ensure that they preserve serializability. If a transaction is validated, it means that it has not executed operations that conflict with the operations of other transactions, and it then commits. A distinction can be made between optimistic concurrency control schemes based on *forward* validation or *backward* validation, depending on the manner in which conflicts are determined.

Forward validation checks to ensure that a committing transaction does not conflict with any still active transaction and, consequently, that the committing transaction's effects will not invalidate any active transaction's results. One possibility [11] is to abort the committing transaction if a conflict is detected with a transaction that is still active. This scheme can not be used with look-ahead, for a former transaction might abort because of a conflict with one of its look-ahead transactions, which would in turn then trigger the abort of the look-ahead transaction as well.

A different forward validation protocol, avoiding wasted aborts, such as broadcast commit [12] should be used with look-ahead. It guarantees to commit all transactions that reach their commit point. In this strategy, all active transactions including look-ahead transactions that have performed operations conflicting with the validating transaction are aborted.

Backward validation protocols check that the committing transaction does not conflict with previously committed transactions. If a conflict is detected, the committing transaction is aborted. Backward validation protocols work

fine with look-ahead, since the commit of look-ahead transactions is artificially delayed until their former transactions successfully committed (see section 4.2).

4.7. Exception Handling

Look-ahead complicates exception handling in two situations:

- An exception might be thrown in a former transaction from which one or several participants have already looked ahead;
- An exception is thrown in an implicit or explicit look-ahead transaction.

Exception in Former Transaction An internal exception that is thrown in a former transaction does not require any additional action to be taken. The participant that encountered the exception first attempts to handle it locally, and, if the handling is successful, the execution of all participants, including the look-ahead participants, can continue as is. However, if local handling fails, an external exception is propagated to the outside, which is treated as an abort vote. Hence, the former transaction must be aborted, together with all implicit and explicit look-ahead transactions.

The idea is illustrated in Figure 5. When participant B throws the external exception Y and votes for aborting $T1.1$, the work performed by look-ahead participants C, D, and E is automatically aborted as well.

An alternative to automatically aborting all look-ahead participants is to inform them of the abort by throwing a new pre-defined exception *LA_Exception* in their contexts. Transactions that have been designed with look-ahead in mind could then attempt to handle the situation. However, the difficulty of writing correct look-ahead handlers increases with the amount of look-ahead and difference in nesting levels, and is therefore not recommended (see section 6).

Exceptions Thrown in Look-Ahead Context If an internal exception is thrown in a look-ahead participant, the participant could attempt to immediately handle it locally. However, the exception might have been triggered by the fact that the former transaction is still in progress. This can happen, for instance, when the look-ahead participant tries to access an object that was supposed to contain a result that the former transaction was supposed to produce. In this case it makes sense to block the look-ahead transaction before performing any handling. Subsequently, if a resource conflict between the former transaction and the look-ahead is detected, then the look-ahead transaction is aborted and restarted. Only if there is no resource conflict and the former transaction commits successfully, the participant is unblocked and can then start handling the exception.

5. Implementation

5.1. Look-ahead Run-time

The run-time support for open multithreaded transactions has been implemented in an object-oriented framework called OPTIMA [3]. Class hierarchies implementing standard transactional behavior are provided by the framework, for example, support for optimistic and pessimistic concurrency control, different recovery strategies, different caching techniques, different logging techniques, and different storage devices. In addition, programmers can customize the framework by implementing their own support classes. This flexibility is achieved using design patterns.

In order to implement looking ahead in OPTIMA, the component that handles the transaction life-cycle had to be adapted to keep track of all look-ahead dependencies and to take care of aborting look-ahead transactions in case a former transaction aborts. Additionally, the concurrency control component implementing the pessimistic lock-based protocol had to be changed as described in section 4.6. These changes, however, are completely transparent to the application programmer.

Blocking the creation and termination of threads in implicit transactions encapsulating lone code turned out to be implementable using aspect-oriented programming techniques. The technique assumes that the transaction context is associated with a thread using *InheritableThreadLocal*. This class provided by the standard Java run-time ensures that newly created threads inherit the same transaction context as the thread that creates them. Using AspectJ [13], it is then possible to intercept the creation of a thread, determine if the creation has happened within an implicit transaction and block it accordingly. Since Java is a garbage collected language, intercepting the termination of threads is not easy. We have so far decided to prevent look-ahead of spawned participants from a nested transaction in our Java implementation of OPTIMA.[1].

5.2. Look-ahead Interface

An important part of OPTIMA is the interface it offers to programmers. Good interfaces should be *easy to use*. They should not require the programmer to write complicated or contrived code, but offer clear abstractions that

[1] In our Ada implementation of OPTIMA, blocking thread termination is feasible. By associating a controlled object with a thread, its destructor is invoked before the thread terminates.

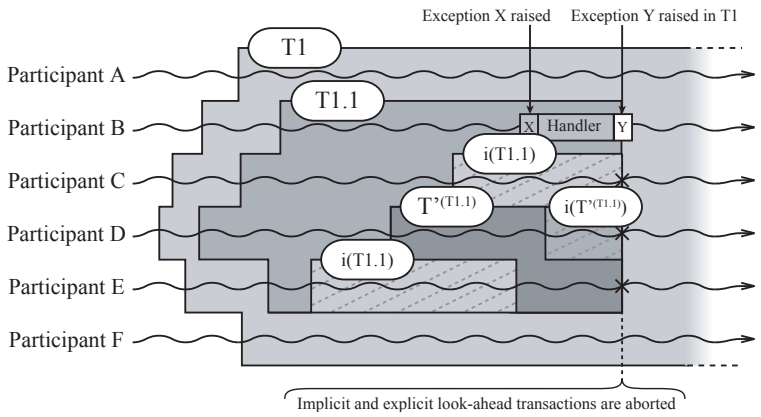

Figure 5. Exception Handling and Look-Ahead

are integrated with all other programming language features. An additional requirement for good interfaces, especially in the context of reliable system development, is *safety*. A programmer should not be able to make mistakes that would result in an incorrect program when using the framework. Based on these criteria, a procedural, an object-based, and an object-oriented interface to OPTIMA have been developed in [14] for the Ada programming language. An aspect-oriented interface for the Java version of OPTIMA is described in [15].

Unfortunately, none of the previously developed interfaces to OPTIMA can support look-ahead. The main challenge is that the framework must be able to restart look-ahead transactions and implicit transactions encapsulating lone code in case a former transaction aborts. In mainstream programming languages such as Java and Ada, it is not possible to jump to any point in the program and start execution from there. This is a major problem when introducing look-ahead, as illustrated in Figure 6.

The code snipped attempts to buy a computer by means of an online auction, and then, if there is enough money on the account left, buys a printer at an online store. The bottom of Figure 6 shows a piece of AspectJ code that uses the aspect-oriented interface of OPTIMA to make the *bid* method transactional (marked with a $\star^1$). This is done by defining a pointcut that intercepts all calls to public methods of *Auction* objects (marked with $\star^2$). The actual code that starts and commits the bid transaction is hidden inside OPTIMA.

With look-ahead, the participant executing the Java code optimistically looks ahead from the bid transaction as if the auction was successful, continuing the execution, querying the balance and buying the printer. But other participants of the bid transaction might abort the auction later on. All changes made to transactional objects by the look-ahead participant, such as buying the printer, can be undone, since they have been executed from within transactions. However, after aborting all the changes, the execution has to resume *inside* the while loop. This is unfortunately not possible, not even with the help of aspects.

```
successful = false;
while (!successful} {
   // browse online auctions for computers
   try {
     auction.bidForComputer();                    ⋆1
     successful = true;
   catch (Transaction_Aborted e)
     { successful = false; }
}
if (myAccount.queryBalance()) > 1000
   { onlineStore.buyPrinter(); }

aspect TransactionalExampleAspect extends
   Optima.TransactionalMethods {
   pointcut MethodsToMakeTransactional() : ⋆2
   call (public * Auction.*(..)); }
```

Figure 6. Problematic Transactional Java Code

To solve this problem, we defined a high level interface to OPTIMA, in which a programmer has to split his code into chunks, and schedule the chunks to be executed with or without transactional semantics. The programmer can then define dependencies among the block in a work-flow like manner.

Figure 7 shows a flow diagram that illustrates the idea[2]. The programmer can specify that the bidding activity has to execute successfully before continuing. Since this interface forces the programmer to write a separate code chunk for the bidding, OPTIMA can jump back into the while loop, and then start a new bid.

[2]Programmer details of the interface are not shown in this paper due to space reasons.

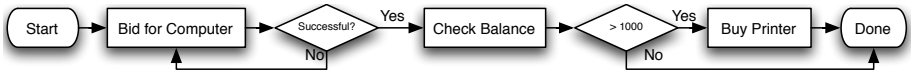

Figure 7. Look-Ahead Interface for OPTIMA

6. Related Work

6.1. Atomic Actions

Atomic actions [16] are atomic units intended for structuring the execution of collaborative concurrent systems. Based on conversations [17], atomic actions provide additional support for forward error recovery and exception resolution. In atomic actions, a *fixed* number of participants enter an action and cooperate inside it to achieve joint goals. They are designed to cooperate inside the action and are aware of this cooperation. The participants share work and explicitly exchange information in order to complete the action successfully. To guarantee action atomicity, no information is allowed to cross the action border. Just like open multithreaded transactions, atomic actions enforce synchronous exit, i.e. participants have to leave the action together when all of them have completed their job.

Since participants can communicate with each other directly, errors can spread from one participant to the other. Therefore, all participants have to be involved when returning the system into a consistent state. Atomic actions provide cooperative exception handling for that purpose. When an exception is raised in any participant, appropriate handlers are initiated in all participants. Concurrent exceptions are resolved using a resolution graph.

In [18], an extension of atomic actions has been developed that does not impose any participant synchronization on action exit. The challenge that has to be faced when introducing look-ahead into atomic actions is cooperative exception handling. In order to still guarantee atomicity and isolation in case of exceptional situations, the closest common parent action to all participants and look-ahead participants has to be identified, and cooperative handling has to be initiated at that level. To this aim, [18] describes a distributed protocol that finds, for any exception raised, an action containing all potentially erroneous information, aborts all of its nested actions, resolves multiple concurrent exceptions and involves all the action participants into cooperative handling of the resolved exception. In the scheme, no service messages are sent and no service synchronization is introduced if there are no exceptions raised.

Comparison Between Look-Ahead in AA and OMTT Although the idea of looking ahead in atomic actions is similar to looking ahead in open multithreaded transactions, the issues that had to be addressed are fundamentally different in each case.

Participants of an atomic action are collaborating closely. They have been designed together, they work together, and they even collaborate to handle exceptional situations. Exception handling has to be done at the closest common parent level, and that level might be high, especially if participants look ahead several times. Writing handlers for looking ahead at a higher level can be tricky, since the details of nested actions are in general unknown to higher level actions. Also, since handling has to be done in the parent action, look-ahead in atomic actions does not allow looking ahead from a top-level action. Since atomic actions do not allow their participants to access external objects / resources, the atomic action look-ahead scheme does not have to address resource sharing issues.

Participants in open multithreaded transactions on the contrary are only loosely coupled. They might collaborate with other participants, but they do this by accessing shared objects. Dealing with resource dependencies between former and look-ahead transactions (see section 4.6) is one of the main issues in look-ahead OMTTs, and requires extending traditional concurrency control algorithms. Exceptions are less problematic, since a participant first attempts to handle them locally. If this is done successfully, the other participants do not have to be involved. In case of an abort caused by an external exception, the look-ahead participants have to be aborted and restarted. Look-ahead situations can therefore be handled without involving parent transactions, and hence looking ahead from a top-level transaction is possible.

6.2. Duality of Fault-Tolerant System Structures

A very interesting paper that describes somehow related work is [19]. The authors analyze transactions – what they call the *Object and Action* model – and atomic actions – what they call *Process and Conversation* model. They suggest that these two models are dual and provide a mapping between them. Using this mapping, they show that mechanisms used in one model can have interesting counterparts in the other model. For example, they analyze locking schemes used in transactions, and show that in atomic actions this corresponds to entering and exiting the action. In order to guarantee isolation, transactions release all acquired locks in one go when the transaction is finished (so-called 2-phase locking [8]). Similarly, participants of atomic actions have to leave synchronously – all

together – to guarantee isolation.

Following these ideas, a parallel can be drawn between looking-ahead in OMTTs and optimistic concurrency control schemes in standard transactions. Optimistic concurrency control schemes allow a transaction to go ahead and modify transactional objects whenever they want to – optimistically hoping that there will be no conflicts. However, all accesses are closely monitored, and when a transaction attempts to commit, validation ensures that the transaction did not perform operations that violate isolation. Similarly, looking ahead allows participants to exit a transaction – optimistically hoping that it will not perform operations that violate isolation. However, all operations performed by look-ahead participants are closely monitored and their effects kept under control by means of implicit transactions.

6.3. OASIS Business Transactions

In June 2002, the Organization for the Advancement of Structured Information Systems, short OASIS, has introduced the *Business Transaction Protocol* [20]. It has emerged based on the observation that maintaining all of the transactional ACID semantics of transactions in a loosely coupled environment such as the Internet is not practical. In such an environment, transactions might take hours, days or even longer to complete, and resources cannot reasonably be locked or reserved for a potentially indefinite amount of time. Intermittent connections and varying load also make it hard to guarantee availability or progress.

While still providing the options of using standard transactions (here called *atoms*), the business transaction protocol also defines *cohesions*. A cohesion may deliver different termination outcomes to its participants; some participants will confirm whilst the others will cancel. The isolation property is relaxed, allowing the effects of a cohesive interaction to be externally visible before the interaction is committed. Consistency is determined by agreement and interaction between the initiator and the coordinator.

Open multithreaded transactions with look-ahead are somehow half way between strict transactions and cohesions. If applied to a distributed Internet application, look-ahead OMTTs do not slow down the loosely coupled participants (similar to cohesions), but can still guarantee the ACID properties in case of exceptional situations (similar to transactions).

6.4. OMTTs and External Devices

OMTTs can be extended to support atomic manipulation of external objects (including devices and files) following, for example, the ideas from [21]. This paper identifies four types of objects which can be made "transactional" under some assumptions and with additional software support. More specifically the reversible objects are classified into objects which can be rolled back by compensation or by step-by-step rollback, the non-reversible objects are classified into objects for which execution can be postponed or which require fictitious execution. To support this we will need to implement a special middleware layer which, depending on the type of the object, logs all the requests directed to the object (with or without executing them). These logs can be either cleared when the transaction aborts or used to avoid double execution after the aborted transaction restarts. For some types of objects all the logged operations are executed on transaction commit.

7. Conclusion

Open multithreaded transactions provide a powerful concept that facilitates the development of reliable, distributed systems with cooperating and competing concurrent entities. Unfortunately, OMTTs also impose strict synchronization between participants of a transaction at commit time. As a result, all participants are blocked until the slowest one finishes its work. Especially for long running transactions, the waisted computation time can be significant.

We have shown in this paper that this synchronization constraint can be relaxed, allowing participants to look-ahead from a transaction and continue on the outside as if the transaction had committed. As a result, participants do not have to be suspended any more, and hence the execution speed of an individual transaction is significantly increased. Our approach is transparent to the application developer, since it preserves the execution semantics of the original model.

To achieve this smooth integration, several technical issues had to be solved:

- Non-transactional code that is executed after a transaction commits had to be encapsulated in implicit transactions.
- The concurrency control of transactional objects had to be made look-ahead aware.
- The joining and forking rules for look-ahead transactions had to be refined.
- The exception handling rules had to be adapted to support exceptions thrown in the former transaction, as well as exceptions thrown in look-ahead transactions.
- The interface to the Java version of OPTIMA, the runtime support for OMTTs, had to be changed in order to make it possible to restart transactions.

Our results show that OMTTs lend themselves perfectly for look-ahead, mainly because of the fact that participants of open multithreaded transactions only collaborate

loosely through transactional objects. Looking ahead is especially applicable in distributed Internet applications such as e-commerce systems, where individual participants of a transaction are very independent entities.

This contrasts with other approaches such as atomic actions or conversations, where due to the tight collaboration of the individual participants the introduction of look-ahead could not be achieved in a transparent manner.

8. Acknowledgments

Jörg Kienzle is partially supported by the Natural Siences and Engineering Research Council of Canada. Alexander Romanovsky is partially supported by the FP6 IST RODIN Project.

References

[1] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Mateo, California: Morgan Kaufmann Publishers, 1993.

[2] R. H. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 811 – 826, August 1986.

[3] J. Kienzle, *Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming*. Kluwer Academic Publishers, 2003.

[4] J. Kienzle, A. Romanovsky, and A. Strohmeier, "Open multithreaded transactions: Keeping threads and exceptions under control," in *Proceedings of the 6th International Worshop on Object-Oriented Real-Time Dependable Systems, Universita di Roma La Sapienza, Roma, Italy, January 8th - 10th, 2001*, pp. 209 – 217, IEEE Computer Society Press, 2001.

[5] A. Romanovsky, "On structuring cooperative and competitive concurrent systems," *The Computer Journal*, vol. 42, no. 8, pp. 627 – 637, 1999.

[6] J. Kienzle, A. Romanovsky, and A. Strohmeier, "Auction system design using open multithreaded transactions," in *Proceedings of the 7th International Worshop on Object-Oriented Real-Time Dependable Systems, San Diego, California, USA, January 7th - 9th, 2002*, (Los Alamitos, CA), pp. 95 – 104, IEEE Computer Society Press, 2002.

[7] K. H. Kim and S. M. Yang, "Performance impacts of look-ahead execution in the conversation scheme," *IEEE Transactions on Computers*, vol. 38, pp. 1188–1202, August 1989.

[8] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger, "The notion of consistency and predicate locks in a database system," *Communications of the ACM*, vol. 19, pp. 624 – 633, November 1976.

[9] C. Kloukinas and S. Yovine, "Synthesis of safe, QoS extendible, application specific schedulers for heterogeneous real-time systems," in *5th Euromicro Conference on Real-Time Systems (ECRTS'03), Porto, Portugal*, pp. 287–294, July 2003.

[10] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems*, vol. 6, pp. 213 – 226, June 1981.

[11] J. R. Haritsa, M. J. Carey, and M. Livny, "On being optimistic about real-time constraints," in *PODS Õ90. Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems: April 2 - 4, 1990, Nashville, Tennessee* (ACM, ed.), vol. 51 (1) of *Journal of Computer and Systems Sciences*, (New York, NY 10036, USA), pp. 331 – 343, ACM Press, 1990.

[12] D. A. Menascé and T. Nakanishi, "Optimistic versus pessimistic concurrency control mechanisms in database management systems," *Information Systems*, vol. 7, no. 1, pp. 13 – 27, 1982.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *15th European Conference on Object–Oriented Programming (ECOOP'2001)*, (June 18–22, 2001, Budapest, Hungary), pp. 327 – 357, 2001.

[14] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martinez, "Transaction support for ada," in *Reliable Software Technologies - Ada-EuropeÕ2001, Leuven, Belgium, May 14-18, 2001*, no. 2043 in Lecture Notes in Computer Science, pp. 290 – 304, Springer Verlag, 2001.

[15] J. Kienzle and R. Guerraoui, "AOP - Does It Make Sense? The Case of Concurrency and Failures," in *16th European Conference on Object–Oriented Programming (ECOOP'2002)* (B. Magnusson, ed.), no. 2374 in Lecture Notes in Computer Science, (Malaga, Spain), pp. 37 – 61, Springer Verlag, 2002.

[16] P. A. Lee and T. Anderson, "Fault tolerance - principles and practice," in *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, 2nd ed., 1990.

[17] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. 1, no. 2, pp. 220 – 232, 1975.

[18] A. Romanovsky, "Looking ahead in atomic actions with exception handling," in *The 20th IEEE Symposium on Reliable Distributed Systems (SRDS '01)*, pp. 142 – 151, IEEE, October 2001.

[19] S. K. Shrivastava, L. V. Mancini, and B. Randell, "The duality of fault-tolerant system structures," *Software — Practice & Experience*, vol. 23, pp. 773 – 798, July 1993.

[20] Organization for Advancement of Structured Information Systems, "Business transaction protocol," June 2002.

[21] A. B. Romanovsky and I. V. Shturtz, "Unplanned recovery for non-program objects," *Computer Systems Science and Engineering*, vol. 8, pp. 72–79, April 1993.