Taking Transactions to a Higher Level

Johan Fabry
Johan.Fabry@vub.ac.be*

April 17, 2001

1 Introduction

An important problem in distributed systems is concurrency management. The concurrent behavior of different clients has far-reaching implications on the data managed by a server computer. When the data is shared by different clients, i.e. used in multiple interactions, it is highly probable that the data will become inconsistent.

A well-known technique for concurrency management in n-tier distributed systems is the use of transactions. A transaction defines an atomic sequence of read and write operations on a database. A transaction takes a database from one consistent state to another consistent state. The use of transactions allows for concurrent access to the data, with the guarantee that no inconsistencies will be introduced into the data.

The important properties of transactions are often abbreviated into a single word: ACID, which stands for:

- Atomic : Either all actions in a transaction are performed, or none of them are performed.
- Consistent : A transaction takes the system from one consistent state to another consistent state
- Isolated : A transaction does not see the effects of other (non-committed) transactions; intermediate effects of a transaction are not seen by other transactions.
- Durable : The changes made by a transaction are kept in permanent storage.

However, using transactions in a distributed system can be a complex task for the programmer. At a low level we have the guarantee that data will be kept consistent, but if we want the application to behave nicely, there is more to take into account. Side-effects of using transactions, such as roll-backs, require extra work to handle correctly.

We feel that we can make transactions more usable and achieve a higher performance by going up to a higher conceptual level. Using extra semantic

*Author funded by a doctoral grant of the Flemish Institute for the advancement of scientific-technological research in the industry (IWT)

information available at this level many tasks can be automated, and more intelligent scheduling decisions can be made for sequences of transactions.

In this paper we will first give a short overview of some of the issues, that have to be taken into account when using transactions, and suggest possible solutions. Second we propose an overall strategy to ease handling these issues: going to a higher conceptual level.

2 Issues

Transactions provide the programmer with a low-level consistency guarantee. However, this guarantee comes at a price: the programmer has to take into account possible roll-backs of transactions, and when using long-term transactions unnecessary resource starvation can occur.

Both of these issues have an important impact on the behavior of the distributed system. The programmer will need to invest a significant amount of effort to ensure that the application behaves correctly in the face of these problems.

2.1 Handling Roll-backs

One major side-effect of using transactions is that they may be rolled back by the transaction management system. Whenever a deadlock occurs between multiple transactions, a transaction will have to be aborted, and rolled back to break the deadlock. Similarly, some transaction management algorithms will only detect interference between different transactions at commit time of a transaction, and rollback the transaction being committed to ensure consistency.

Whenever a transaction is rolled back, the modifications made to the database within that transaction are not stored into the database. This effectively means that some, but not necessarily all, of the work of the transaction is lost. If the code within a transaction has side-effects which are not stored in the database, these side-effects will not be lost. Take for example that at some point in a transaction a form is printed out. If that transaction is rolled back the form will still be printed out, and some undo action must be taken.

It is clear that a well-behaved application should recover gracefully from the rollback of a transaction. Such a recovery mechanism can be complex to build, which significantly increases the burden on the developer. It should be possible to ease recovery by providing some automatic recovery mechanisms to the programmer.

Assume that there are two possible recovery strategies from a rollback: To recover from a rollback, we can either try the transaction again, possibly with some modifications, or abort the work we were performing, possibly including the execution of some undo actions.

Envision a transaction control mechanism that can be instructed to automatically perform such a recovery action when needed. When starting a transaction, the programmer could specify that, for instance, all database accesses performed during the transaction be repeated identically. This “replay” mechanism would effectively try the transaction again, presumably with a successful result.

For example, take a money transfer transaction for a bank application. This transaction subtracts a given amount of money from one account, and adds this amount to a second account. If this transaction is rolled back for some reason, we can just replay it to recover from the rollback.

However, this replay mechanism has only a limited use. In many cases we cannot perform a replay because some preconditions have to be met before a transaction is executed. We can extend the power of the replay mechanism, by allowing for some preconditions to be specified.

Consider our bank example: Usually accounts have a minimum balance amount. The precondition for the transfer transaction should be that the account contains enough funds so the transfer can be performed without exceeding the minimum balance limit.

If the preconditions are not met at replay time, the replay will not occur, and another recovery strategy must be used. An option here is to abort the work we were performing. Again this work can include database accesses, and this could be automated. The programmer can specify "undo" actions on the database, which need to be performed when a transaction cannot be completed. This is useful because although a transaction represents one atomic unit of work, this unit may be one of a sequence. If one unit of work is aborted, the sequence may need to be aborted, which implies the undoing of the already performed units.

Whenever an abort action is taken, the programmer should be made aware of this, for example by throwing an appropriate exception. This is needed for undo actions which are not related to the database to be performed, for example, printing out a cancellation form.

It may seem as if no advantages were gained, but this is not the case. Firstly, replays are performed totally transparently and secondly, a significant part of the abort work is performed for the programmer. All that needs to be done is specifying for a transaction if it can be replayed, and what its' compensating transaction is.

The concept of compensating transactions is heavily used in sagas[GMS87], a concept to enhance the performance of long-lived transactions, which we will discuss next.

2.2 Long-lived transactions

A second side-effect of using transactions is that in some cases resources will be rendered inaccessible for a long time. To preserve isolation between different transactions the transaction management system will in many cases prohibit concurrent access to shared resources.

Exclusive access is a significant problem for long-lived transactions, such as transactions which at some point request for user input. While the transaction is running, resources will not be available for the rest of the system. Other transactions will wait for these resources to be freed, causing a significant performance hit. Also, it has been shown that long-lived transactions have a significantly larger probability to be involved in a deadlock[GMS87].

A well-known concept which addresses these problems is the use of sagas [GMS87]. A saga is a long-lived transaction which is split up in a sequence of sub-transactions. The trick is that each sub-transaction requires exclusive access to a lesser amount of resources than the long-lived transaction requires. Therefore, splitting up into sub-transactions frees unused resources earlier, allowing for more parallelism in the system, and lessens the probability to be involved in a deadlock.

However, the saga still represents an atomic unit of operation, so either all sub-transactions should be performed in sequence, or none of the transactions should be performed. For this to be possible, we need to define compensating transactions for each sub-transaction in the sequence. Applying the compensating transaction for a transaction will, from a semantic point of view, undo the effects of this transaction. Note that this is not equal to restoring the database to the previous state. Take for example a transaction which withdraws 100EUR from a bank account. The compensating transaction would be to deposit 100EUR on the bank account, and not to restore the account balance to the previous state, because in the mean time the account balance may have been altered by another transaction.

By requiring that all sub-transactions in a saga have a compensating transaction, we can 'rollback' a saga at any time. To rollback a saga, for example because a key sub-transaction cannot be completed, the transaction manager will execute the compensating transactions for the committed sub-transactions, in reverse sequence. Note that concurrently running transactions, outside of the saga, will see the intermediate effects of the saga. If a saga is rolled back these transactions will not be informed of this change, indeed it may even be possible that these transactions have already been committed.

We can use sagas if a long-lived transaction conceptually consists out of a number of independent steps, each step may observe the database in a different state, and each step is undo-able. An example is calculating and adding the interest for all accounts of a bank. One step of a saga equals calculating the interest of one account in the database, the compensating action is removing this interest from the account.

As implied above, determining if a transaction can be split up in different steps, and defining compensating steps requires semantic information about the transaction. Once this semantic information is provided, for example by the programmer, an advanced transaction manager could decide to modify a long-lived transaction into a saga.

There is an important similarity between the two issues we have described here: To be able to resolve them we need to have more information than which is currently available. Preferably, we want to have some extra semantic information about the transaction. When this information is available, we can take transactions up to a higher level.

3 Up to a Higher Level

Current transaction mechanisms are difficult to use because at the core, the transactions is a very low-level construct. As we have seen above, if we want to make transactions more usable and achieve a higher performance we need to provide higher-level, semantic information.

Lets look at this from a different viewpoint: consider the well-known three-tier architectures. Transactions are classically a construct used in the database layer, where little or no semantic information is available. A transaction is started, data is read and written, and the transaction is committed (or aborted). There are no options for intelligent managing of transactions at this level, because no information is available. Now let us go up a level, and look at the middle tier. In this tier all business logic is performed, and consequently a large amount of information can be put at the disposal of the transaction mechanism.

By using all this extra information, on the one hand available at the middle tier and, on the other hand supplied by the programmer, we can implement an advanced transaction mechanism. For this we propose a meta-level approach for implementing the system.

First we envision the programmer annotating the code at the meta-level, for example to specify if a transaction is repeatable. Secondly a powerful meta-level tool will inspect these annotations, and reason about the code to determine, for example, compensating actions for a set of operations. In a third step, this meta-level knowledge can be used to modify the code. The modified code will steer the advanced transaction mechanism.

3.1 An Example

Consider the following expanded example: a money transfer application for a bank. The developer first writes a `TransferMoney` method, which takes as arguments source and destination account numbers, and the amount of money to be transfered. The body of the method just contains the application logic: deducting the given amount from the source account, and adding the amount to the destination account. No transaction management is performed within this method.

In a separate file, the programmer specifies that when the method is executed, it should be performed within a transaction. This has as a first consequence that a "begin transaction" is added to the beginning of the method, and "commit" at the end of the method. Further inspection by the meta-tool reveals that the code does not have any external dependencies or side-effects. Therefore, the transaction can just be replayed when an error occurs.

The programmer may not agree with this default error-handling strategy, and when specifying that the method should be performed within a transaction, specify an error-handling mechanism. This could be a conditional replay; the source account must contain enough funds to be able to perform the transfer. Also an additional error-handler can be specified here, for when the condition of the replay can not be met.

We could even go further than this; by using data-flow analysis we could try to determine a compensating transaction for this code. Due to the fact that the method is fairly simple, this succeeds. The compensating transaction for this transaction deducts the amount from the destination account and adds this amount to the source account. This means that this piece of code can be considered as a sub-transaction for a saga. We can imagine a scenario where a bank clerk inputs transfer orders manually, the entire process of inputting and performing a transfer would be a long lived transaction. If other steps of the long lived transaction can be found and their compensating transactions identified, the meta-tool could suggest turning the long lived transaction into a saga.

It is clear that using data-flow analysis on larger, more complex methods will be problematic, if not practically impossible. In these cases we can resort to using higher-level descriptions of the code, if available, for example we could use a description of the business logic behind a piece of code.

The ideal would be that as much data as possible is extracted from the code, from annotations of the programmer and from high-level descriptions. The more relevant information is available, the less code the programmer will have to write to deal with transactions, which implies a faster development cycle.

3.2 Building the system

However, building such a system is not an easy task. The main challenge here is to identify what semantic information is relevant, where it can be found, and how it can be put to good use. We have give two examples above: handling deadlocks and splitting long-lived transactions into sagas.

If the relevant semantic information is acquired, it should not be hard to modify the code to provide this information to the advanced transaction manager. This manager is a run-time component which will use the extra information to schedule transactions in a more intelligent fashion. We imagine that creating such a manager should not be a daunting task.

We can, in fact, use an existing transaction mechanism as a basic building block for our advanced system. Our transaction manager will still use the existing transaction technologies, benefitting from all work done in this area.

4 Conclusion

In this paper we propose a strategy which eases development of distributed systems that use transactions. Current transaction mechanisms are too low-level and require a significant amount of work to be used correctly. Much of this work can be automated by going to a higher conceptual level, and making use of extra semantical information at that level.

For this, we propose an advanced transaction mechanism consisting of two parts: The first part is a meta-level tool which can process semantical information that is provided by the programmer, and is contained in the code and high level descriptions of the application. This processed information is used to modify the code to guide the second part of the mechanism, which is a run-time transaction manager. This manager will use the extra information to schedule the transactions in a more intelligent fashion.

The important problem is to determine what information is relevant, see if it can be deduced from the available information, and if so, how.

References

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.