Migrating Transactions

Johannes Klein, Andreas Reuter

Department of Computer Science
University of Stuttgart

Abstract

Long lived activities still remain a problem in today’s transaction based systems. They decrease system performance and are likely to be aborted because of deadlocks or system failures. This becomes more and more important in distributed systems consisting of unreliable nodes, e.g. a local area network consisting of a set of workstations. We propose a concept, which permits long activities to be broken into several steps realized by short transactions. To maintain the consistency of the data correct termination of the transaction-set is guaranteed. Invariants can be passed to subsequent actions permitting an application designer to specify the assumptions, actions of an activity rely on. The control of the activity is not fixed on a certain node (e.g. node of origin) thus making the execution of the activity independent of nodes it worked on.

1 Motivation

Long lived Transactions still remain a problem in existing transaction processing systems [1,6,7]. They access many objects and take a long time, hours or even days, to execute. This hurts system performance, when other transactions are waiting for objects hold by them. Further they are very likely to encounter system failures and be aborted [8], especially in a distributed system. On the other side there is an increasing interest in using the transaction concept to build up reliable distributed computing systems [9,11].

In reality there are many tasks which do not require to hold on all accessed objects until their final commitment, sometimes even the opposite is necessary. Consider a set of workstations interconnected by a local area network running an office information system or a manufacturing control system. Some tasks are performed by sets of transactions, e.g. claiming travel expenses, organizing a business trip etc. The task is started by invoking its first step, i.e. transaction(s). In the sequel the transaction(s) invoke(s) transactions at other nodes to complete the task and commit(s) without awaiting the outcome of the other transactions.

However, the whole set of transactions, i.e. the task, constitutes a unit which is either executed completely or the system has to run compensations for committed transactions (semantic atomicity) [5,10]. Compensations are transactions for which the system has to guarantee that they will eventually commit, and which re-establish - semantically - the system state before execution of the original transaction.

As already stated, this type of task does not fit into the concepts developed for short transactions. We call this type of transaction - referring to the all-or-nothing property - a migrating transaction (MTA) reflecting the observation that the activity seems to migrate in the network. A similar concept was investigated for centralized systems [6].

The next section shows some of the problems when using transactions for long activities. In section 3 we informally introduce our model to control the execution of a long activity. In section 4 we focus on operational aspects. Section 5 presents a sketch how to implement a control mechanism.
2 What is The Problem to be Solved?

Distributed databases have, without questioning the decision at all, adopted the operational model of conventional centralized databases, which is the transaction [2]. In other words, the only sphere of control is an atomic state transition, which replaces logical consistency by the formal criterion of *serializability*. However, considering a distributed environment means more than just extending conventional techniques to a larger number of computers. So, rather than asking the technical question of how to implement this synchronization algorithm or that commit protocol in a distributed system, we start by looking at typical types of interactions in such a system and then try to define control structures required to support them.

Fig. 1 Claiming travel expenses: Example of a distributed activity

One of the key properties of a distributed transaction in today's systems is the root node, or the node of origin. This is the node where the transaction was invoked, from where sub-transactions are spawned at other nodes, and where the transaction will eventually commit. Notwithstanding the possibility of re-assigning the commit-coordinator in a 3-phase commit protocol [2], the root node is generally assumed to be fixed for each individual transaction. This may make sense for, say, a debit/credit transaction, yet it still is all but a deliberate decision. It simply reflects the centralized heritage of the transaction concept.

In contrast to this view, let us assume a locally distributed system of PCs and/or workstations in a large company. The distributed database, among other things, is to support all personal-related administrative applications of the enterprise. One activity we would like to model in that environment is claiming travel expenses. Fig. 1 shows a decomposition of this activity into a set of basic steps. To simplify our discussion, we will assume that all steps will be executed electronically, i.e., as interactions with the workstations only - an office automation paradise. Considering the sequence of steps and the way they are typically executed, we can make a number of observations:

a) The whole activity may take a long time (a couple of days) to be completed. If the manager of another person involved is on vacation, it may well take weeks.

b) Yet the whole activity is (from the employee's point of view) one consistency preserving state transition, which, once it has been started, should be completed in finite time (a cheque should be in the mail), or there should be an error notification.

c) Once the activity has moved from one workstation to the next one, there is no reason to return to the previous one. In particular, no root transaction must be maintained on the employee's workstation until the last step has been taken.

d) Still the employee must be able to check the status of the activity when he has not received the cheque yet.

e) While one such activity is going on, the employee may move; so may the manager or the people in administration. They may also be given new workstations with different node identifiers, or the job in the administration is...
simply taken over by somebody else. A last complication, which we will not consider here, but which must be dealt with in a real implementation is the employee quitting while the activity is still in progress.

f) Last but not least: Of course, you do not want to keep the department’s travel budget (i.e. the attribute in the corresponding tuple) locked for update for the duration of the entire activity.

Although this is not a comprehensive analysis of this type of a distributed activity, it should be sufficient to illustrate the idea of the kind of control mechanism we need in a general distributed environment. Let us briefly review how far we can get with a classical distributed transaction as we find it in state-of-the-art distributed databases. Basically, we have two options.

First, we can think of mapping the whole activity into one distributed transaction. This will definitely take care of observation b, but it will violate all others. With a transaction in progress, we cannot shut down the root node (obs. a,e), we cannot reconfigure the network or replace participating nodes (obs. e), and, of course, we have to keep long locks (obs. f).

Second, we can make a transaction out of each step and run them in a row. This fulfills the requirement implied by observation a,c,e, and f, but it definitively violates the requirements resulting from b and d. In particular, there is no way we can enforce a certain execution sequence of transactions in conventional database systems. And if something goes wrong in such a sequence of steps (see Fig. 1), it is totally left to the application to do something to abort it. A normal transaction, once committed, stays committed.

So what we need for modeling this type of activity is a distributed control mechanism, which allows to bind a number of local steps (i.e. short lived transactions) together according to some predefined order, maintain the global context of the activity and support some kind of undo methods in case of a failure. In the next chapter, we will introduce an informal conceptual model for such a mechanism.

3 Conceptual Model For a Control Mechanism

We start from a proposal for reconciling activities of long duration with the normal style of transaction-oriented processing. It is called saga, and basically conveys the following idea [6].

A long lived activity is decomposed into a sequence of transactions $T_1,T_2,...,T_n$. After each $T_i$, all updates are committed and all locks are released, so the decomposition must be done such that $T_i$’s modifications to the database, which will be used by $T_{i+1}$, will most likely not be updated at all, or - if they are - do not affect the outcome of $T_{i+1}$. Consider the above example: While a claiming activity is in progress, the department’s travel expense budget may well be changed by others - as long as enough money is left for our activity. But this will most likely be the case.

Now the saga concept gives the following guarantee. Provided, we have transactions $T_1,T_2,...,T_n$ (to be executed in that sequence) and compensating transactions $C_1,C_2,...,C_{n-1}$ for each of them except the last one (which can always be automatically backed out), then a saga will always have one of the following results:

1) normal case: $T_1,T_2,..,T_n$

2) abnormal case: $T_1,T_2,..,T_k+1,C_k,..,C_l$ (This assumes a failure or consistency violation $T_{k+1}$.)

Note that the compensating transactions $C_i$ are essential, because there is no way to automatically roll back committed transactions. Therefore, the $C_i$ have to be provided by the application developer - just like the $T_i$. There are some means for supporting the designer in deriving a $C_i$ from a $T_i$, but this is beyond the scope of this paper.

For executing a long lived activity in a distributed environment, the following has to be satisfied:

We must guarantee termination. Once we have started such a sequence of selected transactions (which we will call migrating transactions (MTA) in the following), we must be able to either reach a normal end or compensate
back to the first step, irrespective of node or communication failures.

We must express more general dependencies among subsequent transactions. Sagas use the ordering as the only dependency, i.e. \( T_i \) precedes \( T_{i+1} \). If there are any semantic dependencies, their checking is left to the application. But remember that we want to model a long activity. If we map it into one transaction (which for the reasons mentioned above is not feasible, but just assume it for a moment), then we would impose a control of the following type:

\[
\text{durallon} \quad \gg \quad 0
\]

Note that an MTA is something totally different from a super-transaction having the \( T_i \)s as its sub-transactions. First, no locks are held above any \( T_i \), and second, there is no fixed place for such a super transaction to reside. An MTA is a totally distributed phenomenon. This should become more obvious during the following chapters.

4 Operational Aspects

As explained in the previous section, to guarantee the consistency of the data(base) each migrating transaction has to terminate correctly, i.e. either all actions are executed successfully or compensations are run for committed actions. Further, each transaction is (conceptually) executed only once. To achieve these properties the state of a migrating transaction has always to be recoverable.

We restrict our discussion to predefined migrating transactions. Their definitions are stored by the system and an entry is made in a public catalog. This allows users to track the state of running migrating transactions of this type.

The values of the parameters and the status of the transactions are recorded in recoverable storage. This can be done by using general transactions facilities as provided by [11]. In the sequel we call the parameters atomic or local variables, because operations on them are performed within transactions and they are local to the migrating transaction, i.e. they are deleted after it finishes.

\[\text{More general structures than a sequence are easily conceivable, like trees or graphs.}\]
Atomic variables are used - as mentioned - to exchange values between transactions and record data for executing compensations. For each transaction an implicit status variable is allocated which can be questioned to obtain the state of the transaction (undefined, committed, aborted). The variables are owned by the migrating transaction. Access rights can be granted to the outside to allow others to track the state of a migrating transaction and to allow actions or compensations to read or update the values of a variable. In order to grant access to actions or compensations, invoked by migrating transactions, but running on behalf of users, it must be possible to bind access rights to a single transaction which return them after completion. This function has to be supported by the underlying transaction facility.

Conditions defined for each transaction (action, compensation) determine the execution order within a migrating transaction. For defining the conditions predefined predicates such as COMMIT or ABORT are used to question the status of other transactions. They are combined by logical operators such as conjunctions and disjunctions. In extension to the basic model this permits an execution order to be a graph like structure, e.g. Fig. 4. However, the graph has to be acyclic and the execution of an action should never depend on the execution of a compensation.

Since disjunctions are permitted to be used in conditions, several sets of actions can accomplish the work of a migrating transaction. Therefore, our criterion for correct termination of a migrating transaction has to be adjusted to that, i.e all actions constituting an action set have to be executed successfully or compensations are run for actions, which only belong to failed action sets.

As an example, the abort of T5 in Fig. 4 only causes to compensate for T2, however, the abort of T3 can cause compensations to be run for T1,T2,T3,T4 and T7. Permitting actions to abort independently (e.g. T2 and T5) requires that other actions running concurrently (e.g. T3,T4,T6,T7) do not access any local variables modified by them.

More general conditions can be defined by predicates ranging over all kinds of local variables, but we restrict the discussion to predicates testing the status of transactions.

---

Fig. 4: A migrating transaction with multiple action sets.

<table>
<thead>
<tr>
<th>Condition</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>'user'</td>
<td>T1</td>
</tr>
<tr>
<td>T1</td>
<td>T2</td>
</tr>
<tr>
<td>T2</td>
<td>T3</td>
</tr>
<tr>
<td>T3</td>
<td>T4</td>
</tr>
<tr>
<td>T4</td>
<td>T5</td>
</tr>
<tr>
<td>T5</td>
<td>T6</td>
</tr>
<tr>
<td>T6</td>
<td>T7</td>
</tr>
<tr>
<td>T7</td>
<td>T8</td>
</tr>
</tbody>
</table>

Table: The definition of a migrating transaction

<table>
<thead>
<tr>
<th>Condition</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 \neg T3 \neg T4 \neg T6 \neg T7 \neg T8</td>
<td>C1</td>
</tr>
<tr>
<td>T2 \neg T3 \neg T4 \neg T6 \neg T7 \neg T8</td>
<td>C2</td>
</tr>
<tr>
<td>T3 \neg T4 \neg T6 \neg T7 \neg T8</td>
<td>C3</td>
</tr>
<tr>
<td>T4 \neg T3 \neg T6 \neg T7 \neg T8</td>
<td>C4</td>
</tr>
<tr>
<td>T5 \neg T3 \neg T4 \neg T6 \neg T7 \neg T8</td>
<td>C5</td>
</tr>
<tr>
<td>T6 \neg T4 \neg T7 \neg T8</td>
<td>C6</td>
</tr>
<tr>
<td>T7 \neg T6 \neg T8</td>
<td>C7</td>
</tr>
<tr>
<td>T8</td>
<td>..</td>
</tr>
</tbody>
</table>

4.1 Context of Transactions

To guarantee that a migrating transaction terminates correctly, even if the environment changes during its execution, the location of the context, i.e. the location of data and users, must not depend on physical addresses, user names etc. The users

---

516
and their data should be determined by specifying their functions in the environment assuming that users change their physical location or might be assigned to other functions, but the organizational structure of the environment remains stable.

Therefore, we assume that users and data are located by using roles and that there is a schema of the organizational structure describing how to locate them, e.g. the schema of the organizational structure of an enterprise which allows to locate the managers of departments, the salesmen for certain products etc.

4.2 Executing Transactions

Transactions, actions or compensations, are executed as soon as their conditions are satisfied, but only once per invocation of a migrating transaction. During commit processing new versions for modified local variables are created. Additionally, actions verify that their compensations are still valid and "lock" them to prohibit incompatible modifications.

In contrast to actions, compensations never abort. Therefore, the abort of a compensation always causes its restart. However, this only makes sense, if a compensation is aborted because of a system failure, a deadlock or other reasons, which disappear after a while. For other circumstances such as a program error etc., a system exit has to be provided, which allows the problem to be solved by hand.

Instead of declaring an action as aborted and initiating necessary compensations, restart points can be defined. Returning to a restart point always causes the corresponding transaction to be re-executed. Compensations are run for all actions on the path up to the restart point. The status of the actions is reset to undefined. Example: If T2 is defined as a restart point and T6 aborts, compensations are run for T5 and T2, after that T2 is invoked again. Defining each action as a restart point allows to omit all compensations because the migrating transaction never fails.

Ambiguities arise when multiple restart points are reachable, e.g. T4 and T2. Either there is an arbitrary choice or the user defines preferences. As with independent aborts of actions, actions on the path up to the restart point can be independently compensated, if no other action accessed local variables modified by them, e.g. when restarting the migrating transaction from T4 after T6 aborts, it is necessary to compensate for T7 - if it is already committed - because it accessed at least the status variable of T4. Additionally, it might be impossible to use T4 at all, if T3 or T2 accessed local variables modified by T4 and we only permit to restart actions, which are successors of the restart point, e.g. T7 is a successor of T4.

4.3 Coping with Interleaving Transactions

As explained in the previous section using transactions for long activities to keep data invariant is unfeasible. Therefore, the idea of invariance predicates was introduced, which allow to declare what are the prerequisites for subsequent steps of an activity.

Basically, a transaction submits a predicate (invariant) to the transaction system. The invariant gets a unique identifier and its value (true/false) can be questioned by later transactions. As an extension, it can be specified, which values bound to the free variables of the predicate should be kept constant, thus allowing explicitly to describe the values seen by a (migrating) transaction.

Example: A subsequent action needs to verify that a price-range agreed upon between a user and a travel agency is still valid. The following invariant (selection) is issued as a relational expression [8]:

\[ \sigma_{(\text{Price}>1\text{,Des}>2)}\text{Offer}(\text{Price}, \text{Dest}, \text{Hotel}) \]

A subsequent action not only requires that the range agreed upon remains stable but also the destinations seen, the following invariant is declared:

\[ \text{Agreement}(\text{Dest}) := \pi_{\text{Dest}}\sigma_{(\text{Price}>2)}\text{Offer}(\text{Price}, \text{Dest}, \text{Hotel}) \]

In general invariants are only re-validated. If they are changed, i.e. false, the issuing transaction is aborted. Additionally, the predicates of the invariants can be used as locks [4], if there is a high probability that they might be invalidated. However, they should only be held for short pe-
riods. Other transactions requesting conflicting locks are aborted to avoid deadlocks between migrating transactions or short transactions waiting for long locks. However, compensations break conflicting 'invariant' locks in order to guarantee their termination. For compensations locks on data can be preclaimed, but this requires that it is done for every compensation.

5 Implementational Issues

The (distributed) system we consider in the sequel, consists of a set of data servers supporting the execution of transactions accessing data managed by different data servers, i.e. support a distributed commit protocol [2]. Some of these might be user written servers [11], others are database systems designed for managing huge amounts of data, but supporting only a limited set of data types.

The execution of a migrating transaction is controlled by a set of migrating transaction managers, which evaluate the conditions, execute transactions (actions/compensations) and send messages to other migrating transaction managers to indicate the status of a transaction. For storing their data they use data servers and perform operations within transactions to guarantee that their state is always recoverable. In detail, they consist of an input queue and an output queue for storing incoming and outgoing messages, a step table which contains the definitions of transaction steps and an active-steps table storing copies of step definitions for active steps.

Our discussion is restricted to predefined migrating transactions. They are compiled and the result, i.e. the steps are numbered and the transaction step definitions are inserted into the step-tables of the responsible transaction managers. Each entry consists of:

- the identifier of the transaction to be executed,
- the condition,
- a set of addresses and step numbers where to send messages after a successful execution,
- a set of addresses and step numbers where to send messages in case of an aborted transaction.

For running a migrating transaction data servers are defined and an entry is inserted into a public catalog, which allows to locate the variables owned by a migrating transaction and to track its status. For availability reasons the data servers can be replicated. Appropriate synchronization protocols must be provided.

The atomic variables of a migrating transaction are encapsulated in data servers. They provide operations for creating and destroying variables for activations of a migrating transaction. Values of a variable are never overwritten. Each modification creates a new version. At the end of a migrating transaction all variables are released.

At invocation of a migrating transaction the responsible migrating transaction manager creates a unique identifier. This can be accomplished by using the value of the invocation counter of this migrating transaction type. Within a second step messages are generated for all actions which have to be started. They are inserted into the output queue for further processing. These messages consist of

- the migrating transaction identifier
- the transaction step to start
- the status of the 'sending' transaction.

After receiving a message the migrating transaction manager checks if there is already a copy of this step in its active step table, if not it creates a copy and adjusts the ranges of outstanding invocations. Messages of migrating transaction outside these ranges are simply discarded. During processing of an incoming message the condition of the corresponding step is evaluated. If it evaluates to
true, the step is marked ready and its associated transaction can be called.

For each step marked as ready the corresponding transaction, i.e. action or compensation, is called. Its outcome is recorded by the migrating transaction manager in the active step table. Depending on the outcome of the user transaction outgoing messages are inserted into the output queue and the step is removed from the active step table.

In case of an abort of the user transaction a restart counter, maintained for each transaction, is decremented and the step is made available for further processing. If the restart counter is less than one, the transaction is marked as aborted and messages to start necessary compensations are generated and the step is deleted from the active step table.

6 Conclusion

Our intention was to show that transactions, as known from state-of-the-art transaction based systems, are not an appropriate concept for executing long lived activities in a distributed environment. Instead of using a long transaction and serializing it with all other transactions, a long lived activity is broken into several steps, which are executed as short transactions. To maintain the consistency of the data we require the correct termination of the resulting set of transactions. To cope with interleaving transactions, a method is provided to specify invariants, which are prerequisite for subsequent actions. They permit a migrating transaction designer to declare, which objects have been seen by an action and, as a consequence, what are the assumptions, further processing relies on.

Eliminating the need for long lived activities to hold long locks on objects is a consequence of our concept. Further, the outcome of a migrating transaction does not depend on the availability of nodes it touched during processing, in particular the node of origin. This is a requirement when executing a migrating transaction in an environment consisting of unreliable nodes, e.g. workstations in a local area network.

We are currently implementing a prototype running on a local area network of workstations (e.g. Ethernet, Sun, PCS Cadmus). Ingres is used as the transaction system.

We choose the name migrating transaction indicating that long lived activities usually do not return to nodes touched during processing. Thus the activity seems to migrate in the network.

References


