The xTrans Transaction Model and FlexCP Commit Protocol

Anna-Brith Arntsen
annab@cs.uit.no
Computer Science Department, University of Tromsoe, Norway

Abstract

Traditionally, transactions are flat and atomic possessing the ACID properties. The traditional ACID transaction model has clear limitations in new application domains where transactions often are long-running and require properties that go beyond ACID. Structuring a long-running transaction as an ACID transaction will impede both performance and concurrency. To meet extended and varying transactional requirements, we have described a flexible transaction model, the *xTrans* model, providing support for both ACID and non-ACID properties.

Further, current transaction processing systems in distributed environments are inflexible with respect to supporting extended transactions. Thus, to assist in the execution of, among others, *xTrans* transactions, we have designed a flexible commit protocol: *FlexCP*. This report presents the *xTrans* transaction model and the *FlexCP* commit protocol.

1. Introduction

Traditionally, applications where build on the classical, flat model of a database transaction where a transaction is modeled as an atomic and isolated unit of work. Such transactions follow the traditional ACID (atomicity, consistency, isolation and durability) properties. While this flat transaction model was successful for small transactions performing simple operations, it is not appropriate for new, complex transactions found within new application domains like for instance real-time systems, mobile computing environments, electronic commerce, CAD, collaborative work, workflow management, and manufacturing control. Such transactions often need to access many data items and are often long-running [17]. A well-known example of a long-running transaction can be

found within travel arrangement scenarios. Such a long-running transaction may consist of sub-transactions like booking a hotel room (T1), a flight (T2), a car (T3), a restaurant table (T4), and a theatre ticket (T5). The sub-transactions, T1-T5, might as well have adjacent contingent transactions or some of them may be defines as not important for the commit of the overall transactions. Assuring ACID for long-running transactions will impede both performance and concurrency, and one would exclude all access to the data for hours.

The inflexibility of the flat transaction model has been addressed for decades, resulting in a number of theoretically described extended transaction models [18]. They address specific transactional requirements, such as relaxed atomicity and isolation, and offer some flexibility. However, these models where described with specific applications in mind, with fixed semantics and correctness criteria. Consequently, they are inflexible with respect to supporting wide areas of applications.

We have described an extended and flexible transaction model, *xTrans*, trying to overcome some of the inflexibility of the previous models. The *xTrans* transaction model described in this report is presented using ACTA [9, 18]. The *xTrans* transaction structure includes the ability to specify subtransactions as contingent, vital versus non-vital, reactivatable versus not-reactivatable and substitutable versus non-substitutable. *xTrans* introduces user-initiated flexibility to gain its desired level of flexibility, which denotes that the role of the individual sub-transaction can change (i.e. from vital to non-vital).

Generally, ACID are the requirements of distributed transactions and to assure ACID of such transactions, a two-phase commit protocol (2PC) or one of its variants (presumed commit or presumed abort) controls the execution. 2PC is implemented within existing middleware infrastructures like Microsoft Transaction Server (MTS), Sun's Java Transaction Server (JTS), and OMG's Object Transaction Service (OTS). implement 2PC according to the X/Open Distributed Transaction Processing (DTP) standard, where interactions with the underlying databases correspond to the X/Open XA-interface specification. Thus, existing transactional middleware platforms support the traditional flat transaction model with ACID guarantees, but with limited flexibility.

We believe that transactional middleware systems must be flexible in order to adapt to extended transactions. To support the execution of xTrans transactions and other extended transactions, flexibility is needed. Thus, we have designed a flexible commit protocol, *FlexCP*, committing transactions either one-phase or two-phase.

In the remainder of this report we first, in section 2, give necessary background information on extended transaction models, the ACTA language and distributed transaction processing. Then, the xTrans transaction model is presented in section 3. Section 4 presents prerequisites to the different parties in a distributed environment. Section 5 follows with a presentation of the FlexCP commit and its associated termination and recovery protocols. Finally, section 6 draws concluding remarks.

2. Background

In this section, we will first look at some extended transaction models and ACTA. Next, we introduce distributed transaction processing based on the X/Open DTP standard and the two-phase commit protocol.

2.1 Extended transaction models

ACTA, a language for specifying transactions

ACTA [9,18] is a framework developed for characterizing the whole spectrum of interactions found in new and extended applications. ACTA allow specification of transactions effects on other transactions and transactions effects on objects. This is done by providing a formalized facility to specify (1) the effects of transactions on other transactions, and (2) the effects of transactions on objects.

(1) Dependencies provide a convenient way of specifying and reasoning about the behavior of concurrent transactions. There are two possible dependencies that a transaction may develop on any other transaction: commit-dependency and abort-dependency. These dependencies, also called completion dependencies impose a commit order, which prevents transactions form prematurely committing and thereby preventing object inconsistencies.

Dependency set, denoted *DepSet*, is a set of inter-transaction dependencies developed during the concurrent execution of a set of transactions T.

Different types of dependencies:
- *Commit Dependency* (T_j $\mathcal{CD}$ T_i): if both transactions T_i and T_j commit then the commitment of T_i precedes the commitment of T_j.
- *Strong-Commit Dependency* (T_j $\mathcal{SCD}$ T_i): if transaction T_i commits then T_j commits.
- *Abort Dependency* (T_j $\mathcal{AD}$ T_i): if T_i aborts then T_j aborts.
- *Weak-Abort Dependency* (T_j $\mathcal{WD}$ T_i): if T_i aborts and T_j has not yet committed, then T_j aborts.
- *Termination Dependency* (T_j $\mathcal{TD}$ T_i): T_j cannot commit or abort until T_i either commits or aborts.
- *Exclusion Dependency* (T_j $\mathcal{ED}$ T_i): if T_i commits and T_j has begun execution, then tj aborts (both T_i and tj cannot commit)
- *Force-Commit-on-Abort Dependency* (T_j $\mathcal{CMD}$ T_i): if T_i aborts, T_j commits.
- *Begin Dependency* (T_j $\mathcal{BD}$ T_i): transaction T_j cannot begin execution until transaction T_i has begun.

- *Serial Dependency* (T_j $\mathcal{SD}$ T_i): transaction T_j cannot begin executing until T_i either commits or aborts.
- *Begin-on-Commit Dependency* (T_j $\mathcal{BCD}$ T_i): transaction T_j cannot begin executing until T_i commits.
- *Begin-on-Abort Dependency* (T_j $\mathcal{BAD}$ T_i): transaction tj cannot begin executing until T_i aborts.
- *Weak-begin-on-Commit Dependency* (T_j $\mathcal{WCD}$ T_i): if T_i commits, T_j can begin executing after T_i commits.

Weak-abort dependency is useful, for example, in specifying and reasoning about the properties of nested transactions. Begin-on-commit, begin-on-abort and force-commit-on-abort dependencies are useful for compensating and contingent transactions.

The list of dependencies is not exhaustive. Other dependencies that involve significant events besides the Begin, Commit and Abort event, can be defined. When new significant events are associated with extended transactions, new dependencies may be specified in a similar manner. ACTA is, in this sense, an open-ended framework.

(2) Transactions effects on objects are captured by the introduction of two sets, the *View Set* and the *Access Set*, and by the concept of delegation. The View Set contains all the objects potentially accessible to the transaction. Objects already accessed by the transaction are contained in another set, the Access Set. When an object in the View Set of a transaction is accessed by the transaction, the object becomes a member of the transaction's Access Set.

A transaction may *delegate* the responsibility for finalizing its effects on some of the objects in its Access Set to another transaction. This is achieved by removing the delegated objects from the Access Set of the first transaction, and adding them to the Access Set of the second transaction.

Nested transactions and Sagas

Nested Transactions [21] and Sagas [17, 18] are the earliest nontraditional transaction models. In the taxonomy of nested transactions, we differentiate between closed and open nested transactions because of their termination characteristics. As opposed to closed nesting, open nesting allows partial results of the transaction to be observed by other transactions.
In the Nested Transaction model, a transaction is composed of an arbitrary number of subtransactions that may be executed concurrently. Top-level transactions have all the ACID properties of traditional transactions. Subtransactions are atomic but share data with their parents and are by so means not fully isolated. Subtransactions are not durable, as an abort of their parent will cause their own abort. Top-level transactions, however, are not required to abort if a subtransaction fail, but can perform its own recovery. Even so,

the nested transaction as a whole remains globally isolated and atomic. Since the nested transactions form hierarchical structures, they reflect modular programming where a subtransaction corresponds to a nested procedure call.

The concept of Sagas is based on *compensating transactions* and *open nesting*. A Saga is a long-lived transaction, LLT, which can be broken into a set of relatively independent subtransactions able to interleave with each other. Associated with subtransactions are compensating transactions, which semantically undoes the effects of a transaction after it has committed. To execute a Saga, the system must guarantee that either all of the subtransactions in a Saga are complete, or any partial execution is undone with compensating transactions. By the notion of open nesting, Sagas relaxes the property of isolation by revealing its partial results to other transactions before it completes. Interleaving of subtransactions in any order may compromise consistency. However, a Saga still requires that all or none of its subtransactions complete. Saga preserves the atomicity and durability properties of traditional transactions.

DOM Transactions

The DOM Transaction Model [18] was developed for the DOM (Distributed Object Management) project to support application development in a distributed object-oriented environment that integrates various component systems. The systems being integrated may be autonomous and heterogeneous and non-database systems (such as file systems) as well as database systems.
DOM Transaction Model allows *closed nested* and *open nested* transactions and combinations of the two. Open nested transactions do not provide the top-level atomicity of closed nested transactions, so the partial results of the transaction may be viewed by other transactions. *Compensating transactions* can be specified to undo the effects of committed transactions, and *contingency transactions* can be specified that are executed if a given transactions fails. Subtransactions can be specified as *vital* or *non-vital*. If a vital subtransactions aborts, its parent transactions must abort. However, if a non-vital subtransaction aborts, the parent may continue. Subtransactions may be executed concurrently and dependencies may be specified, causing the subtransactions to be executed (committed) in a specific order.
The correctness theory for the DOM model has not yet been developed. Generally it is not possible to support the ACID properties for the global transactions since the component system that are integrated may be non-database systems with or without support for the ACID properties and the autonomy of the component system is preserved.
The DOM transaction model is described formally in [18], whereas protocols that cover the model are not found.

ConTracts

The ConTract model [18] was proposed for defining and controlling long-lived, complex computations in non-standard applications like office automation, CAD and manufacturing control. A ConTract defines a set of predefined actions with an explicit specification of control flow among them. The execution of a ConTract must be forward-

recoverable; it must be re-instantiated and continued from where it was interrupted. A ConTract is allowed to externalize its partial results before the whole ConTract is complete, and compensating transactions are used to obliterate the results of committed steps that are not needed. In addition, ConTract allows one to resolve conflicts in a more flexible way by specifying what to do when conflicts occur.

Split-Transactions

Split-transactions [18] were proposed for supporting open-ended applications; for example CAD/CAM projects, VLSI design and software development. Open-ended applications are characterized by uncertain duration, uncertain development and interaction with other concurrent activities. The principle of split-transactions is to split an ongoing transaction into two serializable transactions and divide its resources among the resulting transactions. When splitting a transaction T into two transactions A and B such that A is serialized before B, a set of properties must hold. The properties concern about in what order writes and reads to the same object must be done. The main purpose of split-transactions is to commit one of the split transactions (A in the above case) and release useful results from the original transaction. The other split transaction (B in the above case) continues.

Flex Transactions

The Flex Transaction Model [18] was designed to allow more flexibility. Consider a transaction composed of a set of subtransactions. For each subtransaction, the user may specify a set of *functionally equivalent* subtransactions, each of which when completed will accomplish the subtransaction. A Flex Transaction may proceed and commit even if some of its subtransactions fail as long as there is a functionally equivalent subtransaction able to commit. The Flex Transaction Model also allows the specification of dependencies between subtransactions: failure-dependencies, success-dependencies and external-dependencies. The user is allowed to control the isolation granularity of a transaction with compensating subtransactions.
The Flex Transaction model has been implemented in the Vienna Parallel Logic (VPL) language [22]

Cooperative Transactions

Cooperative transactions [18] need not be serializable; instead, the parent of the cooperative transaction defines a set of rules that regulate the way the cooperative transactions should interact with each other. With cooperative transactions, there is a notion of user-defined correctness criteria that allows different parts of a shared task to use different correctness criteria suitable for their own purposes. Because isolation is not required, the cooperative transaction hierarchies allow close cooperation between transactions and also help to alleviate the problems caused by LLT's.

2.2 Distributed Transaction Processing

The X/Open Distributed Transaction Processing (DTP) reference model [12], figure 1, is a standard for distributed transaction processing defined by the Open Group consortium. The X/Open architecture allows multiple application programs to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions. Applications (AP) define transaction boundaries through what is called the TX interface, and the transaction manager (TM) and resource managers (RMs) interact through the XA interface. The TM controls the execution of a two-phase commit protocol with presumed rollback to assure global atomicity. RMs communicate in the two-phase commit procedure and responds to services requested by the TM. A communication resource manager (CRM) control communication between distributed applications within or across TM domains. The XA+ interface supports global transaction information flow across TM domains.

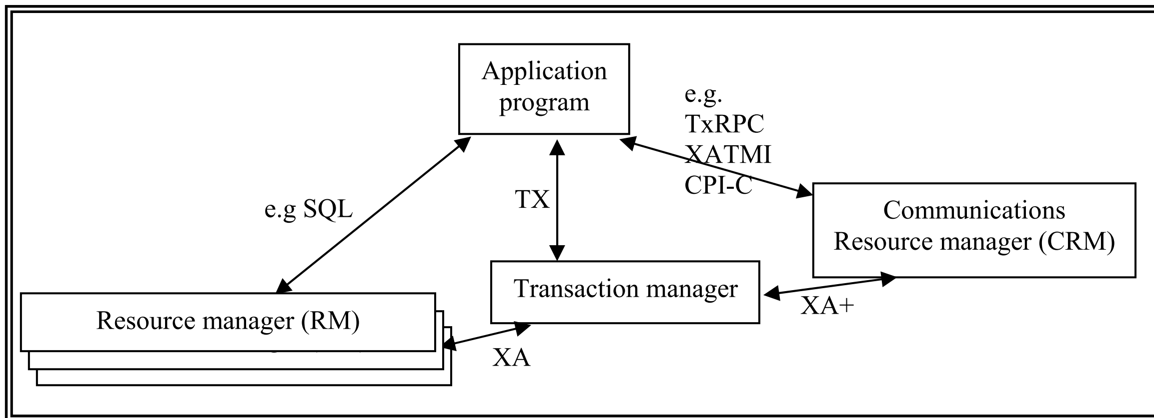


Figure 1. X/Open Distributed TP reference model

Local transaction control (logging and concurrency) is performed by the RMs where the actual operations are executed. A TM must deal with both normal processing and various failure scenarios. Thus, a TM incorporates protocols for termination and recovery. In case of a failure, the TM must coordinate recovery activities of the resource managers. A RM manages persistent and stable data storage system of a single node, participates in commit protocols, and executes rollback on request. With the help from the RMs, the TM preserves the properties of the transaction.

XOPEN/DTP Transaction Model
X/Open DTP model support flat transactions and do not include subtransactions nor nested transactions.

XID, the Transaction Identifier
When a transaction begins, the TM allocates it a unique transaction identifier, *tid*. The *tid* identifies a data structure, XID. The XID is a public structure used to identify a

transaction branch and records among other things the processes or participants who take part in the transaction. A global transaction has one or more transaction branches. A branch is a part of the work of a global transaction. Both RMs and TMs use the XID structure, which lets the RM work with several TMs without recompilation.

TX and XA interface

These interfaces are supported by almost every vendor developing products related to transaction processing, relational databases and message queuing. For instance Oracle, Sybase, Informix, and IBM's MQSeries [24]. Many ODBMS vendors like Versant and Object Design have announced X/Open DTP support for coming releases, or have already delivered first releases. The same applies for database access tool vendors like RogueWave (DBTools.h++) and Persistence Software.

X/Open compliant resource managers provide a XA RM Client Library that incorporates a *XA switch* [24]. The XA switch is an object provided by the RM implementing the XA interface. TM interacts with the database client library using the XA switch, and passes the ID of the transaction.

TX Interface

This interface defines the interaction between the AP and the TM modules [12,3]. The interface APIs are prefixed with tx_, which specify the transaction bracketing, transaction status and transaction control operations.

Some commonly used tx_calls

Calls	Purpose
tx_begin	Starts a transaction
tx_commit	Commits a transaction
tx_rollback	Aborts a transaction
tx_info	Gets the status of the transaction

Table 1. TX_calls

XA Interface

The XA interface defines the interaction between the TM and the RM modules [12,3]. The interface API's are prefixed with xa_ or ax_. The transaction manager issues xa_ calls to interface with the resource manager, and the resource manager issues ax_ calls to interface with the transaction manager. Both the transaction manager and the resource manager modules have to implement this bi-directional interface, which allows a transaction manager to interact with any resource manger (that is X/Open compliant).
A TM must call the xa_routines in a particular sequence. When the TM invokes more than one RM with the same xa_routine, it can do so in an arbitrary sequence.

Services in the XA Interface

Name	Description
ax_reg *ax_unreg*	Register an RM with a TM Unregister an RM with the TM

xa_close	Terminate the AP's use of an RM
xa_commit	Tell the RM to commit a transaction branch
xa_complete	Test an asynchronous xa_ operation for completion
xa_end	Dissociate the thread from a transaction branch
xa_forget	Permit the RM to discard its knowledge of a heuristically-completed transaction branch
xa_open	Initialize an RM for use by an AP
xa_prepare	Ask the RM to prepare to commit a transaction branch
xa_recover	Get a list of XID's the RM has prepared or heuristically completed
xa_rollback	Tell the RM to roll back a transaction branch
xa_start	Start or resume a transaction branch – associate an XID with future work the thread requests of the RM

Table 2. XA_Interface

xa_open - open a resource manager
A transaction manager calls *xa_open()* to initialize a resource manager. Return values indicate whether the call was successful or not. The transaction manager assigns an integer argument, Resource Manager identifier, *rmid,* that uniquely identifies the called resource manager instance within the thread of control. The *rmid* is passed on subsequent calls to XA routines to identify the resource manager. This identifier remains constant until the transaction manager in this thread closes the resource manager. If the resource manager supports multiple instances, the TM can call *xa_open()* more than once for the same resource manager. The TM then generates a new rmid for each call.

xa_start – start work on behalf of a transaction branch
A transaction manager call xa_start() to inform a resource manager that an application may do work on behalf of a transaction branch. Among the parameters send with xa_start are the XID and rmid. Since many threads can participate in a branch and each one may be invoked more than once, xa_start() must recognize whether or not the XID exists. If another thread is accessing the calling thread's RM for the same branch, xa_start() may block and wait for the active thread to release control of the branch (via xa_end()). RM responds according to the result of the call.

xa_end – end work performed on behalf of a transaction branch
TM calls xa_end() when a thread of control finishes, or needs to suspend work, on a transaction branch. This call must be issued within the same thread that accesses the RM. xa_end() return OK when the execution has a normal outcome, otherwise not OK (with indications on what went wrong).

xa_prepare – prepare to commit work done on behalf of a transaction branch
A TM calls xa_prepare() to request a RM to prepare for commitment any work performed on behalf of XID. The RM reports XA_OK if the execution was normal. If the transaction was read-only and has been committed, RM returns XA_RDONLY. And, at last, if the RM is not able to prepare to commit the work done on behalf of the transaction branch, it returns an XA_RB. XA_RB are returned with values indicating whether the

reason for rolling back was unspecified, due to communication failure, caused by a deadlock, integrity violation, protocol error, timeout or other.

xa_commit – commit work done on behalf of a transaction branch
A TM may call this function from any thread of control. All associations for the transaction must have been ended using xa_end(). xa_commit() returns "OK" if the execution was normal, "Retry" if RM is not able to commit the transaction branch at this time and "RB" if the RM has rolled back the transaction branch's work and has released all held resources. If the RM already completed the work heuristically, this function reports how the RM completed (committed, rolled back or mixed).

xa_recover – obtain a list of prepared transaction branches from a resource manager
Prepared transactions are currently either in a prepared or heuristically completed state. RM returns, if OK, an array of XID's of these transactions and the total number of XIDS's, otherwise an error message.

xa_rollback – roll back work done on behalf of a transaction branch
RM rolls back the branch by releasing all resources held restore modified resources and notify all associated threads of control of the branch's failure. If RM already completed the work heuristically, this function merely reports how the RM completed the transaction branch. Return value reflects the result of the call.

xa_close – close a resource manager
A TM must call this function from the same thread of control that accesses the resource manager. Once closed, the RM cannot participate in global transactions on behalf of the calling thread until it is re-opened. The return values indicate whether the call was successful or not. Trying to close an already closed resource manager has no effect, and the return value will indicate that the request was performed successfully. An error is returned if the TM calls xa_close within a transaction branch, i.e. the TM must call xa_end before xa_close.

Commit protocol
When the client requests that the transaction is committed, the TM begins the two-phase commit protocol [5]. Figure 2 illustrates that there can be more than one resource manager participating in the transaction. Each and one of them are involved in the two-phase commit protocol.

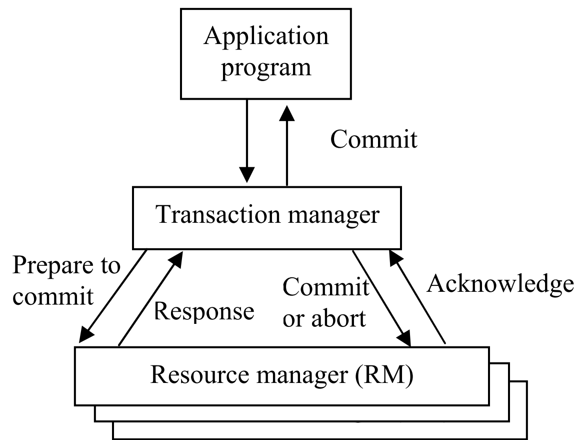


Figure 2: Two-phase commit in X/Open DTP model

TM issues a command to RM, and RM responds back to TM whether the call has been successful or not. Figure 3 illustrates a general interaction between a TM and a RM [12]:

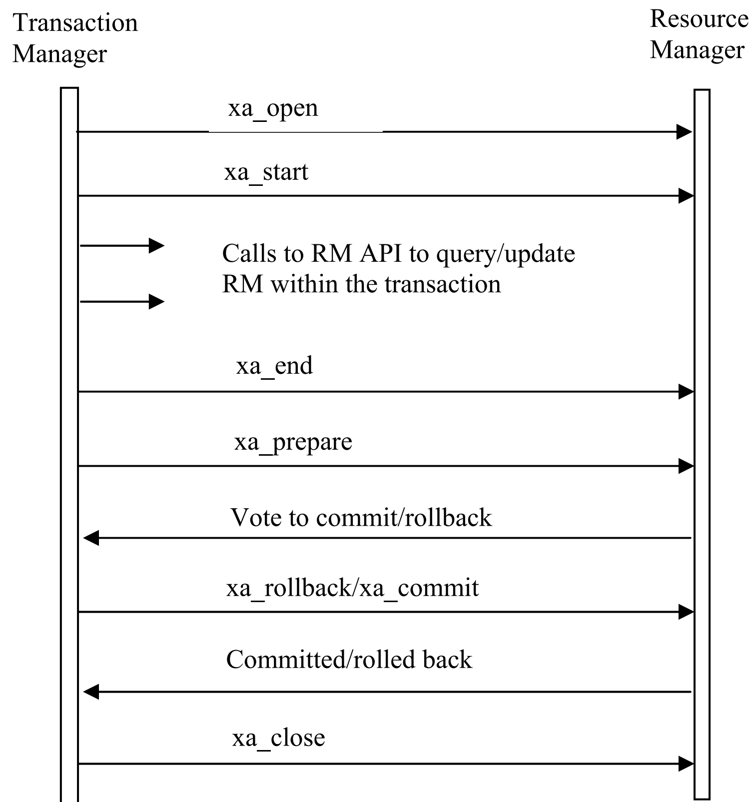


Figure 3. Interaction between a transaction manager and resource manager using XA

3. xTrans Transaction Model

New application domains require support for long-running beyond ACID transactions. Thus, we present a flexible transaction model, *xTrans,* providing support for both ACID and non-ACID transactional requirements. In this section, we first characterize xTrans transactions. Then, we present the variety of the transactional properties lying in the model before formalizing the model using the ACTA language. Finally, we discuss the models potential flexibility.

3.1 Characterization of the xTrans Model

The structure of an xTrans transaction

Generally, an *xTrans* transaction is a partial ordering of operations demarcated by begin-transaction and end-transaction. Splitting into subtransactions can be performed recursively, and will for instance, after two times, result in a transaction family tree with three levels with a top-level transaction T and a number of subtransactions (see figure 4). A node with descendants is a parent and the descendants its children. A top-level transaction does not have parents and leaf nodes do not have descendants. All database access operations are performed at the leaf nodes.

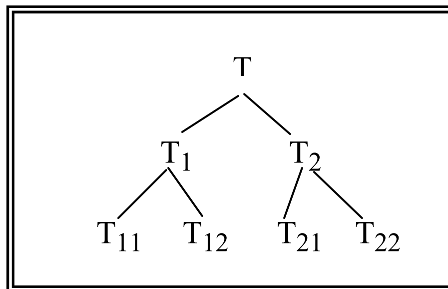


Figure 4. A transaction family tree

The subtransactions (components) of the transaction in figure 8: T, T_1, T_{11}, T_{12}, T_2, T_{21},T_{22},….,T_n are *primary transactions*.

Each primary transaction T_i $(0 \leq i < n)$ can have a set of associated contingent transactions, $ConT_i$. Contingent transactions are *secondary transactions* providing alternatives to the primary transaction. A contingent transaction is executed if the primary

transaction fails. Component transactions with adjacent contingent transactions are s*ubstitutable*; otherwise, *non-substitutable.*

Each component transaction T_i ($0 \leq i < n$) (primary or a contingent transaction), that can be compensated for have an associated compensating transaction, CT_i. A compensating transaction is a *secondary transaction* and is executed if the top-level transaction aborts. A compensating transaction CT_i undoes, from a semantic point of view, any effects of T_i. Both primary and contingent transactions may have an associated compensating transaction, but a top-level transaction does not. A component transaction with an associated compensating transaction is *compensatable* - otherwise *non-compensatable*.

A primary or a secondary transaction with the option to be reactivated in case of failure during trials, are *reactivatable*, otherwise *non-reactivatable*.

Primary and contingent transactions may be *vital or non-vital*. Compensating transactions are always vital. A vital subtransaction is one that is of severe importance to the top-level transaction. The top-level transaction can commit only if all of its vital subtransactions has committed or reached the commit point. Moreover, if a vital subtransaction aborts, the top-level transaction must abort. On the other hand, a top-level transaction can commit independent of its non-vital subtransactions.

Compensatable component transactions do not have to wait for the top-level transaction to commit, but can commit independently, release locks and reveal their results to other transactions. However, non-compensatable component transactions must wait. If a top-level transaction aborts, compensation must be performed for each committed subtransaction. Generally, a parent transaction cannot commit until all of its children have completed (committed or aborted).

A summary of characteristics of xTrans transactions:

- *Open nested transactions,* Compensatable component transactions are allowed to commit when they are finished and before top-level transaction commits. xTrans also support *closed nested* transactions, if there are no compensatable subtransactions, and a *combination of open and closed* nesting.
- *Contingency transactions*: Contingency transactions are alternative transactions that can execute if the original one fails to commit.
- *Compensating transaction*, which is a natural consistency of executing open, nested transactions.
- *Vital vs. non-vital transactions*,
- *Reactivation*,
- *Contingent vs. not-contingent*
- *Compensatable vs. not-compensatable.*

To exemplify, consider a travel arrangement scenario with long-running non-ACID transactions. A user requests a hotel room (T1), a flight (T2), a car (T3), a restaurant table (T4), and a theater ticket (T5). He may also want to specify alternative hotels and

restaurants. Not all reservations are equally important, and a restaurant table may for instance be omitted from the transaction. This transaction must be structured as a beyond-ACID transaction where intermediate results of individual tasks (subtransactions T1-T5) can be committed and revealed when finished. Commit of partial results requires compensating transactions to be specified and executed in case of rollback.

3.2 xTrans Transaction Properties

The atomicity and the durability properties of an xTrans transaction depend on which subtransactions constitute the transaction. The isolation and the consistency properties of the transaction depend on the concurrency control scheme used by the local (or global) transaction services. We will look closer at the atomicity and the durability properties of xTrans transaction model.

Atomicity
Three different compositions of a xTrans transaction have impact on the atomicity property:

1) The transaction consists of only vital and non-compensatable subtransactions. Sutransactions cannot commit until the top-level transaction commits, and every subtransaction must have voted to commit before the top-level transaction can commit. In this situation, *full atomicity* is achieved, and a top-level transaction is an all-or-nothing operation.
2) The transaction consists of only vital subtransactions where some might also be compensatable. The compensatable ones can commit before the top-level transactions decides to commit or abort. If the top-level transaction decides to abort, compensating activities takes place for the committed subtransactions. The result is *semantic atomicity*.
3) The transaction consists of one or more non-vital subtransaction. Top-level transaction can decide to commit even though one of the non-vital subtransactions has aborted. The top-level transaction is not an all-or-nothing operation, rather an all-something-or-noting operation. In this situation, we do *not achieve atomicity*.

We want 2) semantic atomicity to be assured. Consequently, we define that *semantic atomicity* is achieved by top-level transaction as long as all vital subtransactions commits. We do not care about the non-vital subtransactions. Even though not all non-vital subtransactions commits, the user get all of the transaction that is important to him. Atomicity is still preserved at subtransaction level. This is the local database system's responsibility.

Durability
Durability must be preserved for top-level transactions so that it cannot be rolled back. When a client is informed about the results of a transaction, he takes actions based on the results, and do not want them to be 'taken back'. On the other hand, durability is not

preserved for subtransactions. Subtransaction can commit, which indirectly means updating the database and reveal their results to other transactions. If then, the top-level transaction aborts, the committed subtransactions must be undone/ compensated for. This way, we violate durability at the subtransaction level. Durability is violated for compensatable primary and contingent transactions and preserved for compensating transactions.

Durability preservation at top-level transactions is a requirement. As long as we do not allow rollback of committed top-level transactions, the transaction model fulfills that constraint.

3.3 Formalization of XTrans Model

The xTrans transaction model consists of open, nested transactions with compensation. A transaction is decomposed to any level of nesting, which gives us a top-level transaction and a set of subtransactions (component transactions). These transactions are called *primary transactions*. *Secondary transactions*, compensating and contingent transactions, can be associated with the component transactions. Compensating transactions can also have associated contingent transactions.
Transactions can have several roles; they can be vital/non-vital, reactivatable/not-reactivatable, compensatable/not-compensatable, and substitutable/not-substitutable.

There are a lot of interactions in xTrans model, and we will capture them using dependency specifications as in ACTA [9,18]. There are interactions between the primary transactions, and there are interactions between the primary and the secondary transactions. ACTA is developed for characterizing the whole spectrum of interactions. In ACTA the semantics of interactions are expressed in terms of (1): transactions' effects on the commit and abort of other transactions and (2): the transactions effects on objects. In the following we will describe the dependencies between the subtransactions and the top-level transaction, and between the primary and the secondary transactions. We will not describe the effects of transactions on objects. This is because the effects of the transactions on objects deals with concurrency control, an issue not touched in this thesis.

A transaction structure, which conforms to our transaction model, consists of *four types of transactions:*

Primary transactions:

- Top-level or parent transaction
- Subtransactions

Secondary transactions:

- Contingent transactions
- Compensating transactions

Subtransactions, compensating transactions and contingent transactions are atomic transactions. Atomic transactions execute concurrently without any interference as though they executed in some serial order, and either all or none of the transaction's operations are performed. Top-level transaction maintains semantic atomicity as mentioned in 4.3.

Each type of atomic transaction is associated with the significant events: Begin, Commit, Abort. Begin is the initiation event for atomic transactions. Commit and Abort are the termination events associated with atomic transactions. These significant events are the primitives of our transaction manager. The specific primitives and their semantics depend on the specifics of our transaction model. For instance, Commit implies that the transaction is terminating successfully and that all of its effects on the objects should be made permanent in the database. Whereas the Commit of a subtransaction in a closed, nested transaction implies that all of its effects on the objects should be made persistent and visible with respect to its parent and sibling subtransactions.

XTrans transactions can have different roles. They can be vital, non-vital, reactivatable, not-reactivatable, compensatable, not-compensatable, substitutable and not-substitutable. These roles are specified as parameters when formalizing the XTrans model.

We summarize the different transactions, their roles (parameters) and their associated transactions in a table:

		Parameters				Associate Trans.	
Type	Classify	vital or ┐vital	reactive or ┐reactive	Subst or ┐subst [4]	comp or ┐comp	Cont [1]	comp [2]
Subtrans	Primary	Yes [3]	Yes	Yes	comp	Yes	Yes
Subtrans	Primary	Yes	Yes	Yes	┐comp	Yes	No
Comp [2]	Secondary	Vital	Yes	┐subst	┐comp	No	No
Cont [1]	Secondary	Yes	Yes	┐subst	comp	No	Yes
Cont	Secondary	Yes	Yes	┐subst	┐comp	No	No

[1] Cont = Contingent transaction and [2] Comp = Compensating transaction.
3) Yes = A transaction can be described with both the parameter and the inverse of the parameter, but not both of them at the same time.
4) Subst = A substitutable transaction

Table 3. Types of transactions with parameters.

From this table we can read that a non-compensatable primary transaction has no compensating transaction associated with it. Primary transactions can be vital or non-vital independent on whether it is compensatable or not. A compensating transaction is always vital, and we don't allow it to have contingent transactions associated with it. There exists, in addition, no compensating transaction to the compensating transaction. A contingent transaction has almost the same options as a primary transaction except that it has no contingent transaction associated with it.

When specifying the dependencies we use:

- P to denote a parent or top-level transaction

Primary transaction

- $T_{i\ (p1,\ p2,\ p3,p4)}$ denotes child transaction number i with its parameters:

Secondary transactions

- $CT_{i(p1,\ p2,\ p3,\ p4)}$ denotes compensating transaction number i with its parameters and
- $ConT_{i(p1,\ p2,\ p3,\ p4)}$ contingent transaction number i.

Transaction's roles are described using parameters:

Parameter 1, p1: vital versus ┐ vital (non-vital)
Parameter 2, p2: to be reactivatable/┐ reactivatable
Parameter 3, p3: substitutable/┐ substitutalbe
Parameter 4, p4: compensatable/┐ compensatable

The following are the dependencies used to capture interactions in XTrans model:
Partial ordering of primary transactions

- To establish a partial ordering of subtransactions, a *Begin-On-Commit* dependency can be established between those that are dependent on each other. For instance, when booking a vacation, we will not let the transaction continue if it is not possible to get a flight reservation. Then, for instance, booking a hotel room, transaction T_i, is begin-on-commit dependent on a transaction T_j: flight reservation. Flight reservation precedes hotel reservation. Booking a hotel room is also begin-on-commit dependent on T_j's contingent transactions if they are to be executed.
  ($T_{i(p1,p2,p3,p4)}\ \mathcal{BCD}\ T_{j(p1,p2,p3,p4)}$) Transction T_i cannot begin execution until T_j commits. If transaction T_j has associated contingent transactions, which needs to be executed because T_j aborts, T_i is also begin-on-commit dependent on the contingent transaction $ConT_k$ that has taken T_j's place:
  ($T_{i(p1,p2,p3,p4)}\mathcal{BCD}\ ConT_{k(p1,p2,p3,p4)}$)

Parent and Children

- ($P\ \mathcal{CD}\ T_{i(p1,p2,p3,p4)}$). Parent is *Commit Dependent* on all its children regardless of their parameters. If both transactions commits, then the commitment of T precedes the commitment of P in the history. This does not force P to commit if T commits, which means that subtransactions can commit independently. Nor does it force subtransactions to commit. P can decide to commit as long as all its subtransactions has terminated (committed or aborted).

Compensating transactions

- ($CT_{i(p1,p2,p3,p4)}\ \mathcal{BCD}\ T_{i(p1,p2,p3,comp)}$) & ($CT_{i(p1,p2,p3,p4)}\ \mathcal{BAD}$ P). Compensating transaction is *Begin-on-Commit* dependent on the transaction to which it is associated. I.e. compensating transaction for T_i cannot begin executing until T_i

commits. Compensating transaction is at the same time *Begin-in-Abort* dependent on its parent. This means that CT cannot start until the parent has aborted.

- (CT_i $\mathcal{CMD}$ P). Compensating transaction is *Force-Commit-on-Abort* Dependent on parent transaction. If the parent aborts, the compensating transaction commits. This is only for those subtransactions that have committed.

Contingent transactions

- ($ConT_{i(p1,p2,subst,\ p4)}$ $\mathcal{BAD}$ $T_{i(p1,p2,subst,\ p4)}$). Contingent transaction is *Begin-on-Abort* dependent on the transaction it is associated with. The dependency says that the contingent transaction $ConT_i$ cannot begin executing until transaction T_i aborts.
- ($ConT_{i(p1,p2,\ \neg subst,\ p4)}$ $\mathcal{BAD}$ $ConT_{i-i(p1,p2,\ \neg subst,\ p4)}$). Contingent transaction (i) is Begin-on-Abort dependent on the previous executed contingent transactions (i-1), if there is any previously executed contingent transaction.

Compensatable subtransactions

- ($T_{i(p1,comp,p3)}$ $\mathcal{WD}$ P). The compensatable subtransaction is *Weak-Abort* dependent on the parent transaction. The dependency guarantees the abortion of an uncommitted child if its parents abort. But it does not prevent the child from committing and making its effects on objects visible to others.

Non-compensatable subtransactions

- ($T_{i(p1,\neg comp,p3)}$ $\mathcal{CD}$ P). The non-compensatable subtransaction is *Commit Dependent* on the parent, which means that the subtransaction cannot commit until the parent are ready to commit.

- ($T_{i(p1,\ \neg comp,p3)}$ $\mathcal{AD}$ P). The non-compensatable subtransaction is Abort Dependent on the parent transaction. If the parent aborts then the subtransaction aborts.

Vital subtransactions

- (P $\mathcal{AD}$ $T_{i(vital,p2,p3)}$). The parent is Abort Dependent only on its vital subtransactions (both compensatable and non-compensatable). I.e. the parent must abort if a vital subtransaction aborts. The parent only decides to commit when all its vital subtransactions has committed.

Reactivatable subtransactions

- ($T_{i(p1,p2,reactivatable)}$ $\mathcal{BAD}$ $T_{i(p1,p2,reactivatable)}$). The subtransaction that is to be reactivated is Begin-on-Abort dependent on itself. This means that reactivation cannot start until the previous attempt has aborted.

These dependencies tell us that:

A partial ordering of the subtransactions (not necessarily between all of them) exists. If both the children and the parent commits, the children will commit before the parent. The parent transaction can decide its termination without regard to the children's decision. If the parent aborts and the children has not yet committed, then the uncommitted children will abort. The parent is only allowed to commit if all of its vital children has committed.
Non-compensatable descendants are not allowed to commit before the parent has decided to commit whereas compensatable children can commit independent of the parent. The compensating transactions are only allowed to start after its compensated-for transaction and the parent has committed.
Contingent transactions associated with the primary transaction are only allowed to start after the abortion of the primary transaction. Alternatively, if other contingent transactions have been executed on behalf of the primary transaction, the contingent transaction is only allowed to start after the previous executed contingent transaction.
Reactivation of a child is only allowed if the previous attempt of executing the transaction has finished abortion.

3.4 Flexible Extensions

The xTrans transaction model potentially embeds flexibility, pointed out in the following. Importantly, the dependency specifications described in the previous section remains untouched even though flexibility is realized.

At least two approaches to flexibility exist: *user-* and the *system-* initiated flexibility. User-initiated flexibility includes 'on-the-fly' decisions on the transaction or parts of it, whereas system-initiated flexibility includes for instance resource management (i.e. real time applications). Intuitively, in the wake of environmental changes, flexibility might be essential. On the other hand, the user may have changed his mind about the transaction. Both situations can be realistic for transactions residing in the system for a long time.

From vital to non-vital
The role of a subtransaction may be changed from vital to non-vital. The situation arises for instance when it is not possible to commit the vital subtransaction or one of its contingent transactions. Consider for instance the travel arrangement scenario and imagine that the car rental subtransaction is vital. If neither this subtransaction nor any of its contingent transactions succeed to commit, the transactions role may be changed from vital to non-vital. Whether there is a need to do changes the opposite way: from non-vital to vital, can be a discussed. We obviously do not care about the outcome of a non-vital transaction, so at first instance it makes no sense in changing a subtransaction from non-vital to vital. However, the user has the ability to do it.

Reactivation
We have the ability to reactivate parts of the transaction if it fails. The reactivation may be performed after a specified amount of time and, if necessary, a number of times. The transaction can change from having the ability to be reactivated to not, or vice versa. The

number of and the time between the trials may also be changed. For instance, consider booking a hotel room. This transaction can be reactivated a number of times. If the transaction does not succeed, the reactivation process may be stopped. Another example is one where we have a transaction trying to reserve tickets to an excursion. This transaction is not defined with the ability to be reactivated. However, if it fails to succeed, the user may want to interrupt and tell the transaction to be reactivated.

Contingent transactions
Dynamically change of contingent transactions. It may be desirable to update (add, change, delete) the list of contingent transactions when the application is running. If for instance, when trying to reserve a seat on a plane, the contingent transactions represent different airline companies on that route, there may suddenly be someone joining the market or other declared bankrupt.

4. FlexCP Prerequisites

This section presents the flexible commit protocol *FlexCP* (*Flex*ible *C*ommit *P*rotocol) supporting ACID as well as non-ACID requirements and its adjacent termination- og recovery protocols. We assume an X/Open DTP environment.

First, we present the general requirements bound to a RM and a TM in such an environment. Then, we look at the requirements for FlexCP in particular, and evaluate whether the XA interface is sufficient for FlexCP. Finally, FlexCP, termination- and recovery protocols are presented.

Resource Manager Requirements

A resource manager (RM) is compliant with the X/Open DTP model when it conforms to the following:

- RM's must provide xa-routines as specified in [12].
- RM's must be able to recognize and *accept XID's* from TM's.
- RM must support a *commit protocol* by providing an xa_prepare() routine and having the ability to report whether it can guarantee commit of a transaction branch. RM must hold resources until the transaction manager ™ directs it to either commit or roll back the branch. RM must support the one-phase commitment optimization and allow *xa_commit()* even if it has not yet received *xa_prepare()*.
- RM must have support for *recovery* and track the status of all transaction branches in which it is involved.
- RM's receives transaction context from the TM via the XA protocol. Each time a TM initiates start, end or commit of a transaction, the RM's are informed. In order

for TMs to use an RM's XA protocol, the RM must provide its RM-specific library for the TM to call. This is known as the XA switch, mentioned in

- It is important to check if the RM's XA library is *thread-safe*. If the library is thread-safe, it means that multiple threads in a process can be associated with the RM at any given time. This means the application can have many threads with RM connections open and can be performing work within calls to xa_start and xa_end. If the XA library is not thread-safe, this requires an application to only have one thread (and hence one transaction) associated with the RM at any given time. Essentially the TM will acquire an XA lock whenever xa_start is called by a thread, and will release this only when xa_end is called.

Transaction Manager Requirements

A TM is X/Open compliant when it conforms to the following:

- Service interfaces. TM's must use the xa_routines the RM provides to coordinate the work of all the local RM's that the AP uses. TM's must call xa_open() and xa_close() on any local RM associated with the TM.
- Transaction identifiers. A TM must generate XID's conforming to the structure described in [12].

A TM maintains the transaction context. Transaction context contains information about the transaction and reflects the transaction model defining the transactions. It contains what follows:

- Transaction identity; X/Open XID (transaction identifier) compatible. It can, as the OTS context, be a structure including a structure that can be transformed to an X/Open XID and vice versa
- Timeout value
- Parameters:
  - Can the transaction be reactivated, and at what intervals,
  - Is the transaction vital or non-vital to the parent transaction,
  - Is the transaction contingent or not, and
  - Is the transaction compensatable or not.
- Parent(s). The parents must be referred with their transaction identity
- A reference to the transaction's compensating transaction (if exists)
- A reference to the start of a list of contingence transactions (if any)
- At what RM will the transaction be executed

FlexCP Requirements

Two-phase commit (2PC) is the most common commit protocol for distributed transactions. As mentioned in 3.4, 2PC is an all-or-nothing operation assuring ACID and strict atomicity for its transactions. In 2PC, a top-level transaction cannot commit until all of its subtransactions have committed. For long-running transactions, this involves a

potential delay in that participating sites must wait and hold all their locks until the top-level transaction reaches a final decision.

Optimistic 2PC (O2PC) [23] is an approach in overcoming the performance drawbacks of 2PC. In O2PC, locks are released as soon as a site votes to commit a transaction, without waiting for the coordinator's final commit or abort message. If a failure occurs, transaction's effects are undone semantically using compensating transactions. O2PC is also an all-or-nothing operation preserving ACID with semantic atomicity.

Requirements from xTrans transactions (given in 3.4) gives that 2PC and O2PC are too stringent with respect to atomicity. xTrans transactions require relaxed atomicity and the ability to commit top-level transactions even though a non-vital subtransaction has aborted. .

Related to the xTrans transaction model, the following must be considered when designing FlexCP:

- The XTrans model requires compensatable subtransaction to be committed immediately. They must be committed by a protocol that allows independent commitment of subtransactions.
- Non-compensatable subtransactions must wait for the top-level transaction to commit. These subtransactions must be committed by a 2CP protocol.
- A top-level transaction can commit even though some of its non-vital subtransactions have aborted.

Consequently, FlexCP must have the ability to commit according to 2CP and to a protocol allowing independent commit of subtransactions. For the last case, we adopt one-phase commit (1CP), described in OTS [13] as a starting point. 1CP as implemented in OTS allows one-phase commitment only when the transaction works on one single resource, and it commits subtransactions independently by simply omit to send *prepare* before *commit*. However, we modify the 1PC part of our protocol to allow one-phase commitment regardless of the number of involved resources.

FlexCP functionalities:

(1) Start 1CP for compensatable subtransactions, and 2CP for non- compensatable subtransactions. Depending on which subtransactions a transaction consist of, FlexCP can run either:

- Only 1CP
- Only 2CP
- Or a combination of 1CP and 2CP

(2) When specified, reactivate failed transactions
(3) When specified, execute contingent transactions.
(4) Change the role of a transaction.

FlexCP will send *commit* to those subtransactions that are compensatable, and *prepare* to those that are non-compensatable. If a subtransaction fails to commit, FlexCP will involve the user to determine if the subtransaction's role must change. Then FlexCP will either reactivate the subtransaction or start a contingent transaction. When FlexCP is ready to commit (all vital subtransactions are either committed or ready to commit), it will send commit to those that have responded vote-commit on the prepare message. If FlexCP decides to abort the transaction (a vital subtransaction has aborted), a compensation transaction is activated for those subtransactions that has committed.
FlexCP maintains a general view over compensatable and non-compensatable, vital and not-vital subtransactions, and terminates the top-level transaction according to the final state of them.

Interface Requirements

Is the XA interface of the X/Open DTP model [12] sufficient in supporting FlexCP.

Section 2.2 describes a typical interaction between a TM and a RM. A TM sends a command to RMs, which in turn returns a response. TM issues *xa_prepare()* or *xa_commit()* when starting the commit process, and RMs responses indicate whether the command was performed successfully or not. If TM issues *xa_commit()* to a compensatable transaction without first sending *xa_prepare(),* it will set a flag that indicates one-phase commitment. Then RM knows that the transaction can be committed even though it has not received prepare first.
A RM can commit a transaction as soon as it has received xa_end() indicating end of the transaction branch. When the TM later issues xa_commit(), RM sends back a response informing TM about the transaction's termination.

The XA interface will both support 1PC and 2PC with its present set of commands. Consequently, we find the XA interface sufficient in supporting FlexCP.

5. FlexCP Protocols

This section presents the FlexCP commit protocol together with its adjacent termination- and recovery protocols. First, an overview of the initializing steps regarding transaction setup is given.

5.1 Transaction Manager Initialization

.

Transaction Manager initializing transactions:

- Decomposition.
- Define possible compensating transactions.
- Define possible contingent transactions.

- During the above steps: fill in all necessary information, parameters and references in the transaction context.
- Maintain two lists belonging to the transaction: one list of transactions that can be compensated for and one list with transactions that cannot be compensated for. This will help coordinator in executing the FlexCP protocol.

Algorithm Transaction Manager

```
declare-var
  trans:Transaction              {Transaction delivered to TM}
  tcontxt:Transaction context
  CONTXT: List of Transaction contexts belonging to both primary and secondary trans.

begin
  Decompose(trans, CONTXT)     {Subroutine that decompses trans and returns a pointer
                                 to the list of transaction contexts belonging to trans}
  for every subtransaction in CONTXT
  begin
     if compensatable transaction
       SpecifyCompensatingTransaction(tcontxt)
     endif
     if substitutable transaction
       SpecifyContingentTransactions(tcontxt)
     endif
  end
  for each needed resource manager
     xa_open() to the RM
  end-for
  for each subtransaction in CONTXT
     StartTransaction(tcontxt)
  end-for
  StartCoordinator(CONTXT)
  for each open resource manager xa_close() to the RM
end

StartTransaction(Transaction context: tcontxt)
begin
   xa_start() to the RM
   perform actions on behalf of the transaction
   xa_end() to the RM
  end
```

The commit protocol coordinator is explained in the next section. The coordinator will in turn invoke other subroutines. When a transaction is to be reactivated, or a contingent transaction executed, the above routine *StartTransaction* is invoked.

5.2 Transaction Service Protocols

The FlexCP transaction service protocols (commit-, termination- and recovery) for a distributed environment maintain semantic atomicity and durability of distributed transactions.

FlexCP Commit Protocol

FlexCP provides two commit options: compensatable sub-transactions are immediately committed, whereas the non-compensatable ones wait for the top-level transaction to commit. A top-level or a sub-transaction, T, cannot commit unless all its descendants has terminated.

Figure 5 shows the state transitions at the coordinator and the participant's site. The figure is a modified figure from [19]. Transaction manager starts the coordinator. The participant's are started either from the transaction manager or from the coordinator when reactivation or start of contingent transaction. All references to XA interface are from [12].
The coordinator moves from state Initial to Wait when having sent xa_commit() or xa_prepare() to the participants. In addition, from Wait to Commit or Abort depending on what the participant's votes. Participants that cannot be compensated for moves from Initial to Wait after having received Prepare from the coordinator. Compensatable participants, after received commit from the coordinator, moves from Initial to Commit or Abort depending on whether it decides to commit or abort.

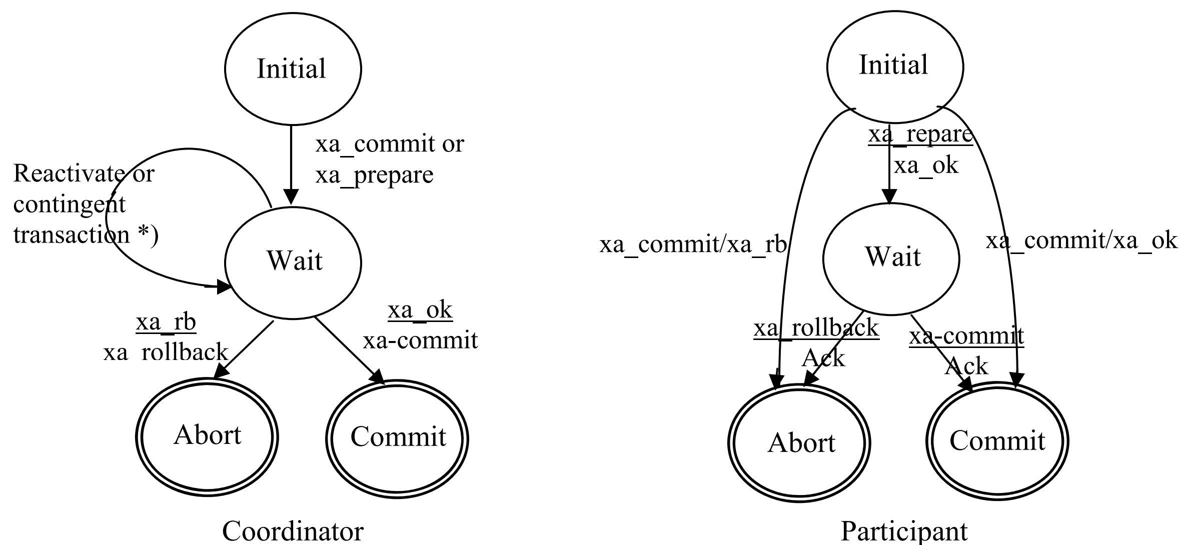


Figure 5. State Transitions in FlexCP

*) The coordinator will remain in the Wait state when Vote-abort is received from one of the participants and the participant can be reactivated or a contingent transaction can be executed. The actions taken are described below under the section: *Coordinator in Wait state*.

Coordinator initially:
Initially the coordinator writes a begin_commit record in its log, sends a *xa_prepare()* message to all non-compensatable sites, *xa_commit()* to every other site, and enters the Wait state.
When coordinator issues xa_commit() to a compensatable site, a flag that indicate one-phase commitment is set (2.2). With this flag set, participants either commit or roll back the transaction and cannot remain in a prepare state. Coordinator sends xa_commit() to non-compensatable sites from the Wait state.

One-phase commit participants:
A participant in Initial state receives a *xa_commit()* message with a one-phase commit flag set. The participant will either commit or roll back the transaction and move to Commit or Abort state. If the transaction can be committed, a commit record is written to the log, the transaction is committed, and XA_OK message sent to the coordinator. If the participant cannot commit the transaction, it writes an abort message to the log, roll back the transaction, and responds with XA_RB message. If a resource manager already completed the work heuristically, this function merely reports how the resource manager completed the transaction branch. A resource manager cannot forget about a heuristically completed transaction branch until the transaction manager calls xa_forget().
In the XA interface there is no specific primitive for one-phase commitment: an resource manager must consider an xa_commit() without preceding xa_prepare() as a request to perform a one-phase commitment. The coordinator gets an XA_OK or XA_RB (roll back) response back from the compensatable participant's RM.

Two-phase commit participants:
When a participant receives a xa_prepare() message, it checks if it can commit the transaction.

- If so, the participant writes a ready record in the log, sends a XA_OK message to the coordinator, and enters "Ready" state.
- If the transaction branch was read-only and has been committed, the participant returns XA_RDONLY and enters the "Ready" state.
- The RM returns XA_RETRY if it is not able to commit the transaction branch. All resources held on behalf of the transaction branch remain in a prepare state until commitment is possible.
- If the RM did not prepare to commit the work done on behalf of the transaction branch, it roll back the work, releases all held resources and returns an XA_RB.

Coordinator in Wait state:
The coordinator takes action when it has received responses from all participants and/or one of the participants has voted XA_RB.

- If all sutransactions has responded, and all vital subtransactions have responded XA_OK, the coordinator decides to commit the transaction globally, writes a globally commit record in its log, and issues xa_commit() to all non-compensatable transaction's that has voted commit and enters the Commit state.

- If the coordinator receives a XA_RB from one subtransaction, the user will be involved, and actions can be taken:

  (1) User can change the transaction's role; for example from vital to non-vital or change reactivation parameters. The user can redefine the transactions role as substitutable and/or rewrite its contingent transactions.
  (2) The coordinator can reactivate the transaction or start a contingent transaction by calling *StartTransaction* desribed in 5.2.

  If (2): the transaction is reactivated or a contingent transaction is started, the coordinator remains in the Wait state. See figure 6.

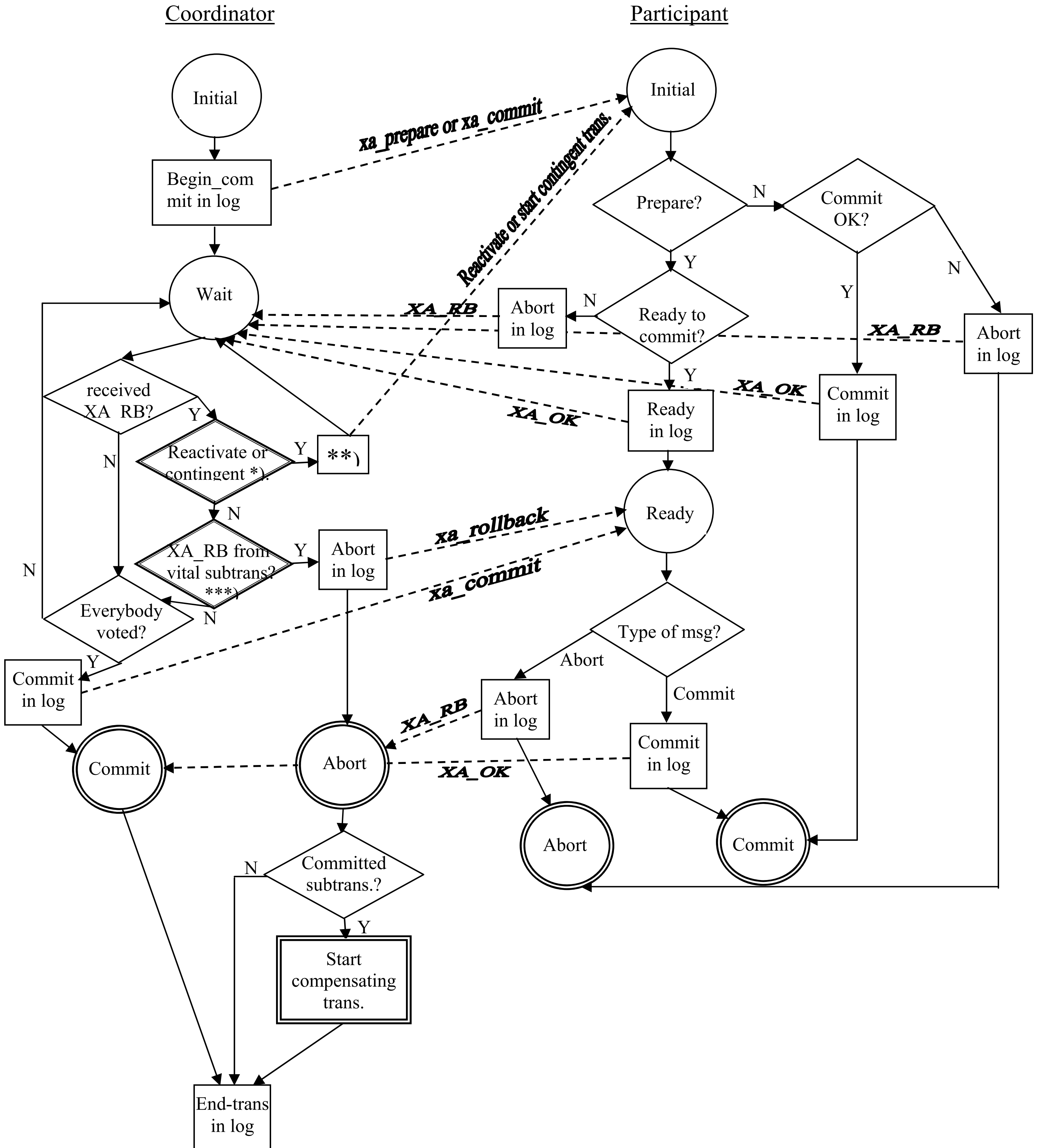
  If the transaction is non-vital, it cannot be reactivated any more and no contingent transactions can be executed, the transaction is 'forgotten'. The coordinator remains in the Wait state.

  If the transaction is vital (the user has not changed it to non-vital), and no possibility to reactivate or run a contingent transaction exists, the coordinator immediately decides to abort the transaction globally. The coordinator will not wait for all the participants to answer if XA_RB is received from a vital subtransaction that cannot be reactivated or has no associated contingent transactions. The coordinator:
  - Writes an global-abort record in its log,
  - Sends xa_rollback() message to all non-compensatable participants,
  - Enters the "Abort" state.

Coordinator in commit state:
Coordinator waits for responses to xa_commit() command sent to non-compensatable participants. The participants will respond with XA_OK or XA_RETRY. If a participant respond with XA_RETRY, coordinator reissues xa_commit() at a later time. When all participants has responden XA_OK coordinator writes end-transaction to log and finishes.

Coordinator in abort state:
A global abort decision is taken after receiving XA_RB from a vital subtransaction.
The coordinator starts compensation for those subtransactions that have committed. Then the coordinator writes end-transaction to log and finishes.

Figure 6. Protocol actions without failures



*) A subtransaction has failed and responded XA_RB to coordinator. A subroutine will be invoked to determine whether the subtransaction shall be reactivated any more, or if a contingent transaction shall be started. The subroutine will interact with the user, and the user can change the transaction's role. If a contingent transaction is to be executed, one is picked from the list of contingent transactions.

**) The subroutine in *) has decided to either reactivate the transaction or to start a contingent transaction, and invokes a transaction manager routine: *StartTransaction*. If the transaction is reactivated, its transaction context is updated. If a contingent transaction is to be run, the list of transaction contexts is also updated.

***) The coordinator has received XA_RB from a vital subtransaction. The subroutine from *) has, in interaction with the user, already decided that the transaction shall not change role from vital to non-vital. The coordinator must abort the transaction according to: Coordinator in Wait state.

Algorithm Commit Protocol Coordinator:

```
declare-var
  msg:Message
  ev:Event
  PL:  List of participants, non-compensatable
  PLC: List of participants, compensatable
  tcontxt: TransactionContext

begin
  WAIT(ev)
  case of ev                {Possible events are MsgArrival and Timeout}
   MsgArrival
   begin
     Let the arrived message be in msg
     case of msg
      Commit:               {Commit command from scheduler}
       begin
         write begin_commit record in the log
         send xa_prepare() message to all the participants in PL
         send xa_commit() message to all the participants in PLC
        if empty PL write commit in log
         set timer
       end
       XA_RB():             {One participant, from PL or PLC, has voted to abort}
       Interact with user()
       if  reactivation or start of contingent transaction
      begin
         Start_Transaction(tcontxt)
       end else
       if vital participant begin
```

```
      {Start global abort on those participating in 2PC, i.e. the PL list}
      write abort record in the log
      send xa_rollback() message to all the participants in PL
      if any committed participants from PLC
        Start their compensating transactions
    end
    XA_OK:             {Both xa_prepare and xa_commit responds with XA_OK **)}
    begin
      update the list of participants who have answered ***)
      if everybody voted and all vital participants have answered XA_OK then
      begin            {Start 2PC on those belonging to PL}
         write commit in the log
         send xa_commit() to all the participants in PL (the others have committed)
         set timer
      end else
          ask for response from the unanswering participants

       if all vital participants have answered XA_OK on xa_commit() then
       begin           {Coordinator can terminate the transaction}
          write end-trans in the log
          set timer
       end else
          ask for response from the unanswering participants
    end

    end-case
   end
   Timeout:
   begin
     execute the termination protocol            {will be discussed later}
   end
  end-case
end
```

Algorithm Commit Protocol Participant:

```
declare-var
 msg:Message
 ev:Event
begin
 WAIT(ev)
 case of ev                 {Possible events are MsgArrival and Timeout}
 MsgArrival:
 begin
  Let the arrived message be in msg
  case of msg
  xa_prepare():
  begin
   if ready to commit then
```

```
    begin
      write ready record in the log
      send XA_OK message to the coordinator
      set timer
    end
    else begin
          write abort record in the log
          send XA_RB message to the coordinator
          call local data processor to abort/roll back the transaction
          end
    end
    xa_commit:
    begin
      if ready to commit then
      begin
          write commit record in the log
          send XA_OK message to the coordinator
          call local data processor to commit the transaction
    end
    else if Heuristically completed then
    begin
          Send message to coordinator about the outcome of the completion
    end else
    begin
          write abort record in the log
          send XA_OK message to the coordinator
          call local data processor to abort/roll back the transaction
          end
    end
    xa_rollback:
    begin
          write abort record in the log
          send XA_OK message to the coordinator
          call local data processor to abort the transaction
    end
   end-case
  end
  Timeout:
  begin
   execute the termination protocol
  end
  end-case
end
```

*)
When reactivating a transaction or starting a contingent transaction the PL and/or PLC list must be updated. In case of reactivation are the transaction's context updated to reflect the action. When a contingent transaction is started the PL and/or PLC list are updated to contain the contingent transaction's context. The 'old' transaction is terminated and removed from the list.

**)
According to [12] is both xa_prepare and xa_commit issuing a XA_OK response when execution has terminated successfully.

***)
The coordinator maintains two lists of XID's that have responeded XA_OK. One list contains those XID's from PL that have responed XA_OK on a xa_prepare command. The other list contains those XID's from both PL and PLC list that have responded XA_OK on xa_commit command. The reason for these two lists is described in **).

Termination Protocol

The termination protocols serve the timeouts for both the coordinator and the participant processes. A timeout occurs at the destination site when it cannot get an expected message from a source site within the expected time period. We need to consider failures at various points of the execution of the commit protocol.

Coordinator Timeouts
There is three states in which the terminator can timeout: Wait, Commit, and Abort. We refer to previous description of the coordinator and the participant algorithm when using acronyms like PL and PLC.

1. Timeout in the Wait state: If the coordinator is in the Wait state it is waiting for the local decisions of the participants. The coordinator can unilaterally commit the transaction if it has received positive responses from all vital subtransactions. The coordinator then writes a commit record in the log and sends a commit message to all participants in PL list. However, it cannot commit the transaction if a vital transaction has not responded. The coordinator can decide to abort the transaction globally, in which case it writes an abort record in the log. Thereafter it sends a xa_rollback() message to all participants in the PL list and start compensation of those participants from the PLC list that have committed.

2. Timeout in the COMMIT or ABORT states. In this case, the coordinator is not certain that the commit or abort procedures have been completed by the local recovery managers at all the participating sites. Thus the coordinator repeatedly sends the xa_commit() or xa_rollback() commands to the sites that have not responded, and waits for their responses.

Participant timeouts. A participant can timeout in two states: INITIAL and READY:

1. Timeout in the INITIAL state. In this state the participant is waiting for a xa_prepare() or a xa_commit() message. The coordinator must have failed in the INITIAL or in the WAIT state. Coordinator is in the WAIT state if the participant is representing a reactivated or contingent transaction. The participant can unilaterally abort the transaction following a timeout. If the prepare or commit message arrives at this participant later, this can be handled in one of two possible ways.

2. Timeout in the READY state. The participant has voted to commit the transaction but does not know the global decision of the coordinator. The participant cannot unilaterally make a decision. Since it is in the READY state, it must have voted to commit the transaction. Therefore, it cannot change its vote and unilaterally abort it. On the other hand, it cannot unilaterally decide to commit it since another participant may have voted to abort it. In this case, the participant will remain blocked until it can learn from someone the ultimate fate of the transaction.

Algorithm 2PC-Coordinator-Terminate:

```
Timeout:
begin
 if in WAIT state then
 begin
   write abort record in the log
   send xa_rollback() message to all participants in PL list
   send xa_rollback() message to all in PLC list that have not committed
   start
 end
 else begin
   check for last log record
   if last log record=abort then begin
     send xa_rollback() to all participants that have not responded
     if already not started
       start compensating transaction for those from PLC list that have committed
 end else
   send xa_commit() to all the participants that have not responded
 end
 set timer
end
```

Algorithm 2PC-Participant-Terminate

```
Timeout:
begin
 if in INITIAL state then
   write abort record in the log
 else                         {participant in the ready state}
   send XA_OK message to the coordinator to vote commit
   reset timer
 end
end
```

Recovery Protocol

A protocol for use by a coordinator or a participant to recover their state when failure.

Coordinator Site Failures:

1. The coordinator fails while in the INITIAL state. This is before the coordinator has initiated the commit procedure. Therefore, it will start the commit process upon recovery.
2. The coordinator fails while in the WAIT state. In this case the coordinator has sent xa_prepare() or xa_commit() command. Upon recovery, the coordinator will restart the commit process for this transaction from the beginning by sending the message one more time.
3. The coordinator fails while in the COMMIT or ABORT states. In this case the coordinator will have informed the participants of its decision and terminated the transaction. Thus, upon recovery, it does not need to do anything if all the acknowledgements have been received. Otherwise, the termination protocol is involved.

Participant Site Failures:
1. A participant fails in the INITIAL state. Upon recovery, the participant should abort the transaction unilaterally. Let us see why this is acceptable. Note that the coordinator will be in the INITIAL or in the WAIT state with respect to this transaction.

6. Conclusion

We have seen that there is a gap between provided and required needs for extended transactional requirements. Even though a number of theoretically described transaction models have been proposed in order to close this gap, they are inflexible with respect to supporting wide areas of applications and varying transactional requirements. Thus, we have described an extended and flexible transaction model, xTrans, trying to overcome the restrictions of the previous models. We have used the language ACTA to describe and capture all interactions in the model.

Further, to support the execution of xTrans transactions, flexibility is needed. Present infrastructures supporting distributed transactions, like for instance TCM, OTS and JTS, mainly provide support for flat, ACID transactions. We believe that these infrastructures must be flexible in order to adapt to extended transactions. Consequently we have designed a flexible commit protocol, FlexCP, committing transactions either one-phase or two-phase – or both. The characteristics of each individual transaction decide how it is used. For instance, contingent transactions may exist, reactivation of transactions may be allowed, and the role of a sub-transaction may change through some intervention with a user or. Thus, FlexCP supports both ACID and non-ACID transactions. Moreover, we found the XA interface sufficient in supporting FlexCP.

Besides achieving flexibility within both the xTrans model and the FlexCP commit protocol, concurrency and performance have increased. Concurrency has increased by splitting and decomposing long-running transactions in to concurrently executing subtransactions. Performance has improved by the ability to commit subtransactions independently without waiting for the final decision of a top-level transaction. This of

course depends on the specification of compensating transactions. In addition, performance is increased as a top-level transaction immediately aborts if a vital subtransaction fail to succeed.

When designing the transaction model, XTrans, and the flexible commit protocol, FlexXP, we have assured that both semantic atomicity and durability are preserved at the top-level transaction.

7. References

[1] Linda G. DeMichiel, L. Ümut Yalcinalp: Enterprise JavaBeans Specification 2.0 Draft. SunMicrosystems Inc., August 10, 1999
[2] CORBA Component Model: http://www-106.ibm.com/developerworks/components/library/co-cjct6/
[3] Andreas Vogel, Madhavan Rangarao: Programming with Enterprise JavaBeans, JTS and OTS.
[4] Subrahmanyam. Java Transaction Service. http://www.subrahmanyam.com/articles/jts/JTS.html
[5] Ian Gorton: Enterprise Transaction Processing Systems
[6] Marek Prochazka: Advanced Transactions in Enterprise JavaBeans
[7] Marek Prochazka: Extending Transactions in Enterprise JavaBeans. Technical Report 3/2000, Department of Software Engineering, Charles University, Prague (May, 2000).
[8] JBoss: http://www.jboss.org/
[9] P.K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. ACM SIGMOD, 1990
[10] http://www.cs.uit.no/~weihai/D441Sv01/
[11] http://www.omg.org/technology/documents/
[12] The X/Open CAE Specification. Distributed Transaction Processing: The XA Specification. X/Open Document Number: XO/CA/91/300. December 1991.
[13] CORBA Services, Transaction Service Specification, v1.1. 1997
[14] Ramamritham, Chrysanthis. Advances in Concurrency Control and Transaction Processing. IEEE Computer Society. 1997.
[15] Anne Thomas. "Enterprise JavaBeans Technology: Server Component Model for the JavaTM Platform". Patricia Seybold Group (Prepared for Sun Microsystems, Inc.), Revised December 1998.
[16] Elsmari, Navathe. Fundamentals of Database Systems, Third edition
[17] H. Garcia-Molina, K. Salem. SAGAS. ACM SIGMOD conference 1987
[18] Ahmed K. Elmagarmid. Database Transaction Models for Advanced Applications.
[19] Õzsu, Valduriez. Principles of Distributed Database Systems. 1999
[20] Garcia-Molina, Silberschatz, Breitbart. Overview of Multidatabase Transaction Management.
[21] J. E. Moss. Nested Transactions. The MIT Press, 1985.
[22] E.Kuehn, F. Puntigam, A. Elmagarmid. Transaction specification in multidatabase systems based on parallel logic programming. IMS91, Japan, April 1991.
[23] Levy, Korth, Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. University of Texas at Austin. 1991
[24] Slama, Garbis, Russell. Enterprise CORBA. 1999, Prentice Hall
[25] Enterprise JavaBeans Technology, server component model: http://java.sun.com/products/ejb/white/white_paper.html
[26] Korth, Levy, Silberschatz. A Formal Approach to Recovery by Compensating Transactions. The 16th VLDB Conference, Australia, 1990
[27] Szyperski. Component Software. Beyond Object-Oriented Programming
[28] Brian Cantwell Smith. "Procedural Reflection in Programming Languages". PhD Thesis, Massachusetts Institute of Technology, 1982.