ADAPT

IST-2001-37126

Middleware Technologies for Adaptive and Composable Distributed Components

Transaction Support

Deliverable Identifier:	D5
Delivery Date:	3/21/2003
Classification:	Public Circulation
Authors:	Ricardo Jiménez-Peris, Marta Patiño-Martínez
Document version:	1.0, 3/20/2003
Contract Start Date:	1 September 2002
Duration:	36 months
Project Coordinator:	Universidad Politécnica de Madrid (Spain)
Partners:	Università di Bologna (Italy), ETH Zürich (Switzerland), McGill University (Canada), Università di Trieste (Italy), University of Newcastle (UK), Arjuna Technologies Ltd. (UK)

Project funded by the European Commission under the Information Society Technology Programme of the 5th Framework (1998-2002)

ist information society technologies

Contents

1 Introduction 3
1.1 Introduction to Transactions . . . 3

2 Advanced Transactions: Models and Frameworks 3
2.1 Nested Transactions . . . 4
2.2 Sagas . . . 4
2.3 Open Nested Transactions . . . 4
2.4 Flexible Transactions . . . 4
2.5 ConTracts . . . 5
2.6 S-transactions . . . 5
2.7 The DOM Transaction Model . . . 5
2.8 Multi-Coloured Actions . . . 6
2.9 The Activity Service . . . 6
2.10 Workflow . . . 7

3 Specifications for Transaction Processing 7
3.1 Basic Services . . . 7
3.1.1 Java Transaction Service and Java Transaction API . . . 7
3.1.2 Transactional support in EJB containers . . . 8
3.2 Composite Services . . . 9
3.2.1 Business Transaction Protocol . . . 10
3.2.2 WSCoordination/WSTransaction . . . 11

4 Dynamically Adaptable Transactional Storage 12
4.1 Implementation of Database Replication Protocols . . . 12
4.2 Replicated Clients . . . 13
4.3 Dynamic Adaptability . . . 14

5 Transactional Composite Services 15
5.1 Our Vision for Transactional CSs . . . 16
5.2 Description of Advanced Transaction Models . . . 17
5.3 Flexible Transaction Management Support for CSs . . . 17

1 Introduction

In this document we explore the different alternatives to provide transactional support for basic (BSs) and composite services (CSs). In Section 2 we study some advanced transactional models proposed in the literature, paying special attention to those suitable to support CSs. A more detailed description of most these models here can be found in [Elm92, DA93, JK97]. In Section 3 we look at the specifications and technology for transaction processing for both BSs and CSs more closely related to the goals of ADAPT, such as J2EE for BSs and transaction coordination protocols for CSs. In Section 4 it is discussed which transactional support will be provided in ADAPT for BSs. Since transactional state is kept on databases, we describe the planned support for dynamically adaptable databases. In Section 5 we discuss the support to be provided at the CS level, both to compose transactional web services and to run them. In the rest of this section, some basic concepts and terminology are introduced.

1.1 Introduction to Transactions

Transactions [GR93] were originally proposed in the context of databases to guarantee the consistency of data in the presence of failures and concurrent accesses. A transaction *commits* if it finishes successfully. Otherwise, it *aborts*. Transactions exhibit the so-called ACID properties:

- Atomiticity: A transaction is executed completely (commits) or the final effect is as it has never been executed (aborts).
- Consistency: The code of the transaction should guarantee that if applied to a consistent state of the data it should bring the data to a new consistent state.
- Isolation: The execution of concurrent transactions accessing common data should be equivalent to a serial execution of them.
- Durability: The effects of a committed transaction should be preserved even in the advent of failures.

Transactions guarantee the isolation property by means of concurrency control mechanisms (such as locking). The atomicity and durability properties are enforced by recovery protocols. Transactions can be distributed, that is applications can manipulate data at different locations within the scope of the same transaction. Some synchronization is required among all the sites participating in the transaction in order to guarantee the ACID properties at the global level. This synchronization is called distributed atomic commitment [GR93]. The two-phase commit (2PC) protocol is the most well-known of such a protocols.

2 Advanced Transactions: Models and Frameworks

The traditional transaction model is simple and provides clear semantics. However, the model does not fit all application needs. For instance, if a long-duration activity (it can take hours or months) is run as a transaction, it is not desirable to undo completely the transaction if there is a crash just before finishing. At the same time, such a long-lived transaction may access many data items. If another (short) transaction needs to access any of these items, then it will end up waiting for the long-lived transaction to finish. A wide variety of extended transaction models have been proposed over the last decade in order to overcome the limitations of traditional transactions found in different application scenarios [JK97, Elm92]. According to [WS91] new transaction models and correctness criteria have been proposed for the following reasons: to provide better support for long-lived activities in advance database applications, to relax the classical ACID paradigm, to support cooperation between members of a group of designers in CAD/CAM or CASE, to capture more semantics about the operations, to enhance inter-transaction parallelism, to define intra-transaction savepoints, to deal with autonomous subsystems in federated databases, and to deal with conversational transactions. In general, extended transaction models relax either isolation and/or atomicity properties of traditional transactions.

2.1 Nested Transactions

Traditional transactions evolved incorporating the idea of nesting into the traditional (flat) transaction model [Mos85]. The basic idea behind *nested transactions* is that new transactions (*subtransactions*) can be started within another transaction, therefore yielding to a transaction tree structure. The root of the tree is known as top-level transaction. Top-level transactions preserve all the ACID properties. Subtransactions are atomic and isolated with respect other subtransactions in the tree that are not ancestors nor descendants. If a subtransaction aborts, its parent transaction is notified about the fact. The parent transaction can choose to abort or to perform a contingency action. The commit of a subtransaction is not final, and depends on the commit of all its ancestors. If a transaction aborts all its children transactions (even they if they committed) are aborted, and transitively, all its descendants. Nested transactions exhibit two properties: increased parallelism and fault-tolerance. Subtransactions can be executed in parallel, which increases concurrency within a transaction. Subtransactions also allow a finer granularity of failure handling (by tolerating abortions of subtransactions).

2.2 Sagas

Sagas [GMS87] were designed to deal with long-lived transactions by relaxing the isolation property. A Saga is a sequence of subtransactions. Each subtransaction has associated a *compensating transaction*, which logically undoes its effects. The effects of a subtransaction are made visible after its commitment. The Saga either completes successfully or in case of abortion each of the committed subtransactions are undone by applying the corresponding compensating transactions in commit reverse order. A Saga commits if all its subtransactions also commit.

2.3 Open Nested Transactions

Open nested transactions [WS92] relax the isolation property of nested transactions in order to improve performance. Open subtransactions externalise their results as soon as they finish. Those results can be visible only to those subtransactions that *commute* (their effects are the same regardless of the order in which they were executed). Open nested subtransactions are persistent. If a transaction aborts, its committed open nested subtransactions are not automatically undone neither compensated. The only way to undo a committed open nested transaction is that the programmer explicitly invokes a compensating transaction. The persistence property of open nested subtransactions makes them suitable for long-lived activities. In general, some subtransactions might be marked as open and other as closed. Closed subtransactions exhibit the same semantics as nested subtransactions.

2.4 Flexible Transactions

Flexible transactions [ELLR90, MRSK92, ZNBB94] were proposed as a suitable model for heterogeneous autonomous distributed databases. This model relax the atomicity and isolation by means of compensation. The structure of transactions is a two-level nested transaction. The top level transaction is called a global transaction and may execute at different sites. Subtransactions are executed at a single site. The Flex transaction model allows the user to specify a set of functionally equivalent subtransactions (*alternative subtransactions*). They provide additional flexibility achieving a particular subgoal by different means. Each alternative subtransaction, when completed, will accomplish the desired task. A preference function can be associated to alternative subtransactions (e.g. minimal cost) so they are tried in a particular order. The model allows the specification of dependencies on subtransactions. The dependencies indicate the execution order of subtransactions and the events that start/end a subtransaction.

Different kinds of subtransactions are identified in this model: compensatable, retriable and pivot. *Compensatable* subtransactions are those transactions that can be undone after commitment by using a compensating transaction (i.e. they are backward-recoverable). *Retriable* subtransactions are guaranteed to commit after a finite number of submissions (i.e. they are forward-recoverable).

Pivot subtransactions are those transactions that are not compensatable nor retriable. In this model, compositions should fulfil the following rules: (1) At most one pivot subtransaction is allowed; (2) Prior the execution of the pivot subtransaction (or non-compensatable subtransactions in the absence of a pivot subtransaction) only compensatable subtransactions can be executed; (3) After executing the pivot subtransaction, only retriable subtransactions can be executed. The outcome of a transaction is determined by the outcome of the pivot transaction. If the pivot aborts, all the compensatable subtransactions are compensated. Otherwise, the retriable subtransactions are executed until they commit. This model has been extended to permit more than one pivot transaction whilst atomicity is relaxed by allowing alternative subtransactions for a given task [ELLR90, MRSK92].

2.5 ConTracts

The *ConTract* model [RS92, RS97] was proposed for controlling long-lived computations in non-standard applications, like workflow and CAD. ConTracts relax the isolation property by externalising partial results before the contract finishes. A ConTract is a fault-tolerant execution of a sequence of actions (steps) according to a control flow description (script). ConTracts are forward recoverable. The outcome of ConTract steps are made stable so that, in the advent of a failure, the ConTract continues from where it was interrupted. Steps are traditional ACID transactions. Steps can be grouped in transactions. There are events associated to steps and transactions. An step (transaction) starts execution when its event predicate becomes true (*entry invariant*). After step (transaction) termination some events (which may depend on the step result) may become true (*exit invariant*), which trigger other steps (transaction). Contingency plans can be defined to be used in case the entry invariant does not hold. Updates can be externalised at the end of a step. For each step there exists a compensating step, which is used to undo the effects of steps in case the ConTract is cancelled.

2.6 S-transactions

The *S-transaction* transaction model supports cooperation between organizations and can be applied in any environment where the autonomy of the organizations must be preserved [EV88]. They were defined to support cooperation of the banking system. The basic building block of s-transactions (*sub-s-transactions*) are traditional ACID database transactions. Their combination gives rise to dynamically generated hierarchical distributed transactions (nested transactions). S-transactions do not preserve the isolation property. Sub-s-transactions commit as soon as they finish, externalising their results. A transaction decides on its result based on the results of its subtransactions. If it fails, the failure is propagated to its committed subtransactions. Compensating transactions are used to semantically undo committed subtransactions.

2.7 The DOM Transaction Model

The goal of the DOM project [BzHG92] was to develop an object oriented environment in which autonomous and heterogeneous systems can be integrated. The model supports long running activities (isolation is not preserved) and the execution of system-triggered activities within transactions. The model is based on nested transactions (*toptransactions*), which can be defined by the user or being the result from the firing of rules while executing a transaction. The structure of a nested transaction in the former case is specified by the coupling mode of the rule. That is, the subtransaction is executed at the point of the parent transaction where the rule was fired (*immediate*) or at the end of the parent transaction but before it commits (*deferred*). Or the new transaction is another parallelizable transaction without any commit dependencies on the spawning transaction (*detached*). *Causally dependant* detached execution results in also a parallelizable subtransaction but, with commit dependencies on the parent transaction. These subtransactions can only see the latest committed state (they cannot see the modifications done by the spawning transaction).

Toptransactions make their results visible when they commit. Toptransactions can be combined into *multitransactions*. The component transactions of a multitransaction (either toptransactions or multitransactions) can be *vital* or *non-vital*. If a vital transaction aborts, its parent transaction also aborts. Compensating transactions are used to undo committed components of a multitransaction (they are applied in the inverse order of commit order). The model also considers *contingency* transactions, which are alternative transactions (possibly with weaker goals) in case the primary fails. Components of a multitransaction can be executed in parallel. Precedence rules can be used to enforce sequential behaviour (start/commit after the precedent transaction commits).

2.8 Multi-Coloured Actions

Multi-coloured actions [SW90] are based on nested transactions, however they are suited for long-lived computations. They increase the concurrency of nested transactions and allow partial results (subtransactions) to became permanent in the advent of failures of the enclosing transaction. The authors propose three different types of nested transactions in order to achieve these properties. Subtransactions of a *serializing action* do not preserve the atomicity property. If a subtransaction commits and its parent transaction aborts later, the effects of the subtransaction are not undone. *Glued actions* are intended to increase concurrency when long-running transactions are considered. They allow locks on some objects to be passed from one transaction to another. Those objects remain unchanged in between the two transactions. Objects that are not passed are released when the first transaction commits. *Top-level independent actions* are started within a transaction, however they are independent transactions. The spawning transaction can wait for the spawned transaction to finish or they can be executed in parallel. Coloured transactions are a framework to implement these types of transactions. Colours define types of locks and are assigned statically to transactions. Transactions are serialized according to their colours.

2.9 The Activity Service

The *Activity Service* is a specification by the Object Management Group to support a large variety of transaction models [OMG02, HLR$^+$01]. The activity service is not expected to be used by end-user application programmers. It is a low level infrastructure for the coordination and control of application activities.

An *activity* is a unit of (distributed) work. An activity may have transactional and non-transactional periods and can run over an arbitrary period of time. Activities can be composed of other activities.

Activities interact with the rest of the system through *Signals* and *Actions*. An activity may need to communicate some data to other activities, for instance, the activity has terminated. This is done using signals. Instead of sending directly the signal to the activity, they are sent to actions. This allows activities to be independent of other activities and to establish the coordination and control points outside the activities. An action will use the signal information in a manner specific to the transaction model. For instance, given a signal an action can start another activity. The information within a signal depends upon the implementation of the transaction model.

The signals sent to actions depend on the transaction model and are associated with *SignalSets*. When a signal is raised it does so in the context of a signalset and only actions associated to that signalset are notified. There is an *activity coordinator* per activity. Activities that need to be informed when another activity send a signal register an action with the activity coordinator of that activity. When an activity sends a signal, the activity coordinator forwards it to all registered actions for that signal. The activity coordinator deals with the outcomes of these actions by passing them to the signalset for their composition. The signalset encompasses all the semantic information about how to compose results and which signal send next. When the signalset does not have more signals to send determines the final outcome for the activity.

The activity services does not require any specific mechanisms for persistence and recovery. Since the application imposes the meaning to signals, actions and signalset, the application logic will also drive the recovery process.

2.10 Workflow

A workflow involves the coordinated execution of multiple tasks which can be heterogeneous and performed by heterogeneous processing entities [KS95]. When a workflow terminates abnormally, completed tasks may be undone or compensated (*backward recovery*). Alternatively, incomplete tasks may need to be redone (*forward recovery*). Persistent storage is used to keep the state of each task and the inputs and outputs of tasks for the purpose of forward recovery [KS95, AM97]. Tasks can be either transactional, which are a unit of atomicity and isolation, or not. Transactional tasks are resubmitted for execution, if they are not marked as executed in the permanent storage. A mechanism is needed to know whether non-transactional tasks were completed or they need to be idempotent. If none of the two conditions are met, human assistance becomes necessary [KS95]. Backward recovery has limited applicability because it is either not possible to undo some tasks or might have an additional overhead.

Although it is useful to incorporate transactional semantics to workflows, existing advanced transaction models do not provide a sufficient framework for modeling large-scale enterprise-wide workflows [WS97, SR93]. The authors of [AAA+96] argue that workflows are a superset of advanced transaction models. They show this by implementing several advanced transaction models on top of a commercial workflow system.

Transactional workflows involve the coordinated execution of tasks that require access to heterogeneous autonomous distributed systems and support selective use of transactional properties for individual tasks or entire workflows [SR93]. A report on transactional workflows can be found in [DA93].

3 Specifications for Transaction Processing

3.1 Basic Services

We define a basic service (BS) as a web service provided by a single organization that does not invoke any other web service. Web services are based on three fundamental specifications: WSDL (to describe services), UDDI (to advertise and syndicate), and SOAP (for communication purposes). All these specifications are based on XML and are platform independent.

BSs are usually implemented using a middleware infrastructure such as J2EE. Such middleware infrastructures provide support for transaction processing. A number of specifications have been proposed for transaction processing and interoperability of transactional systems, such as, X/Open [XOp96], the CORBA Object Transaction Service (OTS) [OMG], and the Java Transaction Service (JTS) [Sun]. Since basic services in ADAPT will be based on J2EE technology (see deliverable D1), we will focus on the specifications around J2EE, that is, JTS and its associated API, Java Transaction API (JTA).

3.1.1 Java Transaction Service and Java Transaction API

J2EE includes support for programmatic distributed transactions through two specifications, Java Transaction API (JTA) and Java Transaction Service (JTS). JTS implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the low-level. JTS uses the CORBA OTS interfaces for interoperability and portability to generate and propagate transaction contexts between JTS transaction managers using the Internet Inter-ORB Protocol (IIOP). Programmers do not use this low level interface. JTS specifies the implementation of a transaction manager that supports the JTA.

JTA is a high-level, implementation independent, protocol independent API that allows applications and application servers to use transactions. The JTA specifies interfaces between a transaction manager and the parts involved in a distributed transaction: the application, the resource manager and the application server. Fig. 1 shows the different parties and the involved interfaces.

- The transaction manager (JTS) provides functions for transaction demarcation, transaction context propagation, resource management and synchronization, as well as for querying the transaction status (prepared, committing, committed).

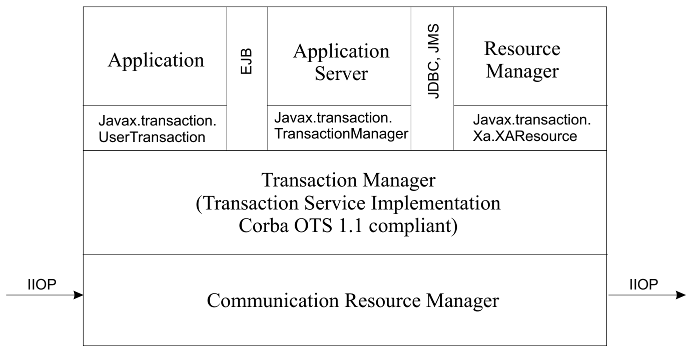
Application
EJB
Application Server
JDBC, JMS
Resource Manager
Javax.transaction. UserTransaction
Javax.transaction. TransactionManager
Javax.transaction. Xa.XAResource
Transaction Manager (Transaction Service Implementation Corba OTS 1.1 compliant)
IIOP
Communication Resource Manager
IIOP

Figure 1: Participants in a distributed transaction in J2EE

- The application server provides the runtime infrastructure for applications running on it. For instance, an Enterpsise Java Bean (EJB) container provides transactional support for the enterprise beans running on it.
- Transactions may be demarcated explicitly by client applications or implicitly by invocating transactional components, such as EJBs, supported by an application server.
- The resource manager is a java mapping to the XA interface based on the X/open specification required by the transaction manager to coordinate the distributed transactions. A resource adapter, such as JDBC, for a resource manager implements the XAResource interface to support association of a global transaction to a transaction resource, such as a connection to a relational database.
- The underlying communication resource manager acts as propagator of transaction contexts along the system.

The JTA does not support for nested transactions.

3.1.2 Transactional support in EJB containers

EJB containers provide support for transactions in two flavours: container-managed transactions (CMT) and bean-managed transactions (BMT). In the former case, the transactional semantics is defined declaratively in the bean descriptor and managed automatically by the container. In the latter case, transactions are managed programmatically in the bean's code.

Container-managed transactions

Container-managed transactions simplify development because the code does not include statements that begin and end a transaction. The container begins a transaction immediately before an enterprise bean method starts. It commits the transaction just before the method exits. Each method can be associated with a single transaction. Nested or multiple transactions are not allowed within a method. Entity beans can only have container-managed transactions.

When deploying a bean, it is specified which of the bean's methods are associated with transactions by setting the transaction attributes. The values of the attribute are: Required, RequiresNew, Mandatory, NotSupported, Supports and Never. The attribute can be specified for an entire bean or for methods.

- The `Required` attribute specifies that if the client is running within a transaction and invokes a bean's method, the method executes within the client's transaction. If the client is not associated with a transaction, the container starts a new transaction before running the method.
- If the attribute has the `RequiresNew` value and the client is not associated with a transaction, the container starts a new transaction before running the method. If the client is running within a

transaction, the container suspends the client's transaction, starts a new transaction, executes the method and resumes the client's transaction after the method completes.

- The `Mandatory` value indicates that the client must invoke the method within a transaction. Otherwise, the container throws an exception.
- The `NotSupported` attribute value indicates that the method is not run as a transaction. If the client is running within a transaction and invokes the enterprise bean's method, the container suspends the client's transaction before invoking the method. After the method has completed, the container resumes the client's transaction.
- The `Supports` value indicates that if the client is running a transaction and invokes the enterprise bean's method, the method executes within the client's transaction. Otherwise, the method is not run within a transaction.
- A method with a `Never` value attribute should not be invoked within a transaction. Otherwise, an exception is thrown.

An EJB container plays the role of application server and gives the illusion to the bean that transaction management support is local and directly accessible to the bean. The application server uses the `javax.transation.TransactionManager` interface to forward transaction management operations to the transaction manager.

A container-managed transaction may be rolled back because a system exception is thrown (not an application exception). Or by invoking the `setRollbackOnly` method of the `EJBContext` interface. The container automatically undoes the changes made to entity beans when a transaction is rolled back. A session bean must explicitly reset any instance variables changed within a rolled back transaction. The `afterCompletion` method in the `SessionSynchronization` interface can be used to refresh the bean.

Bean-managed transactions

In a bean-managed transaction the transaction boundaries are marked explicitly, either using JDBC or JTA.

JDBC transactions are controlled by the transaction manager of the DBMS. Each individual SQL statement is a transaction, unless the `setAutoCommit` method is used. This method informs the DBMS not to commit each individual SQL statement. In this case, a transaction begins with the first SQL statement that follows the most recent commit, connect or rollback statements. A transaction ends when the commit or rollback methods are invoked.

A method of a stateless session bean must commit or rollback a transaction before returning. However, transactions can span to several invocations of a stateful session bean. With a JDBC transaction, the JDBC retains the association of the transaction and the bean unless the connection is closed.

If the transaction is demarcated with JTA, the transaction is retained until the instance completes the transaction (even if one or more database connections are open and closed). In a stateless session bean with bean-managed transactions, a method must commit or roll back a transaction before returning. However, a stateful session bean does not have this restriction. In a stateful session bean with a JTA transaction, the association between the bean instance and the transaction is retained across multiple client calls. Even if each method called by the client opens and closes the database connection, the association is retained until the instance completes the transaction. A JTA (distributed) transaction can span updates to multiple databases from different vendors.

3.2 Composite Services

Composite services are made out of basic services and/or other composite services. Components of a composite service are autonomous and can be provided by different organizations.

The specifications for transaction processing presented in the previous section deal with transactional services in which all the transaction participants share a common middleware infrastructure, such as J2EE. This common infrastructure requirement might be acceptable within a single organization. However, composite services are inherently inter-organizational and therefore, heterogeneous and autonomous. For this reason, web services were proposed as well as other specifications for transaction coordination and context propagation. Web services allow the composition of services across organizations independently of the underlying middleware platforms used at the different organizations. In this setting, two specifications for transaction coordination for web services have been proposed: the Business Transaction Protocol (BTP) [OAS02], an OASIS standard, and the WSCoordination/WSTransaction [CCF$^+$02, CCC$^+$02] specification supported by IBM, Microsoft and BEA Systems. Both specifications define XML messages that are exchanged among the parties involved in a concrete protocol. These standards make possible to add transactional semantics to web services, taking care of the transactional context propagation across the different involved organizations and the required coordination among the transaction managers at each of them. An important feature of these two standards is that both support non-fully ACID transactions, an important requirement to support transactional composite services.

3.2.1 Business Transaction Protocol

The BTP [OAS02] is an OASIS standard to allow the coordination of web-based services between multiple participants of different organizations. It supports the requirements of long-running web-based applications.

The protocol is a two-phase protocol with a preparation phase and a completion phase. Protocol messages and their content are defined using XML, and the specification provides (but does not mandate) a SOAP binding,.

BTP permits the outcome to be predefined (2PC style, all the work is confirmed or none), ensuring atomicity, or the application can select which work to confirm based on business rules, relaxing the atomicity. The former kind of protocol is called *atom* (atomic business transaction) while the latter is called *cohesion* (cohesive business transaction).

BTP participants may use before- or after-images or compensation to provide roll-forward, roll-backward capacity of atomic business transactions. Each participant can determine what it means to confirm or cancel a transaction.

BTP names *actors* to software agents that are able to take part in the BTP exchanges. Actors play roles in the sending, receiving and processing of messages. Given a transaction an actor plays a particular *role*. The party that coordinates a transaction (initiating element) plays the role of *superior* and the rest of the transaction participants (activities that create effects) play the role of *inferior*. The superior gathers information from its inferiors (whether they are prepared) in order to know whether they should be cancelled or confirmed. This superior-inferior relationship can form a tree structure, where leaf nodes are inferiors, the root is a superior and intermediate nodes are superiors of their children and inferiors of their parent.

If the top-level superior coordinates an atom, it plays the role of *atom coordinator*. If it coordinates a cohesion, it is a *cohesion composer*. If a superior is also an inferior, in the transaction where it is a superior is termed *sub-coordinator* or *sub-composer* depending on whether all inferiors of this superior should confirm (or cancel) or can choose which inferiors can confirm.

When a transaction begins a *context* is made available. The context identifies the coordinator or composer, and whether this superior will behave atomically or cohesively. The context is propagated in association with application messages. The receiving application creates an inferior and informs the superior (its address is in the context) about this new inferior.

BTP delivers a consistent decision even if messages are lost (messages are resent) or some sites fail (state information is made persistent).

3.2.2 WSCoordination/WSTransaction

The WSCoordination and WSTransaction specifications provide a WSDL definition, including its HTTP and SOAP binding styles, for coordination protocols of web services which does assume ACID properties, unlike BTP. The coordination protocols include the 2PC protocol used in traditional distributed transactions and more flexible protocols to coordinate the results from multiple services in a more flexible way.

WSCoordination

The WSCoordination sets up a generic foundation for web services coordination. It describes an extensible framework for providing protocols that coordinate the actions of distributed applications in a heterogeneous environment. The protocols are not restricted to transaction processing. Each coordination model is represented as a *coordination type* supporting a set of coordination protocols. This coordination framework consists of a coordination service which is an aggregation of three services:

- Activation service. This service enables the creation of an activity and a coordination context.
- Registration service. This service allows an application to select a protocol and register to it.
- A set of coordination protocol services for each supported coordination type.

A coordination type is specified when an activity starts through the Activation service. The Activation service returns a coordination context, which includes information about the type of activity, a global identifier and a port reference for the Registration service. When an invocation is done on some web service, the coordination context is propagated along with the message. The web service uses the port reference to find out the Registration service and register for the coordination protocol.

WSTransaction

A WSTransaction specification defines two coordination types to be used with WSCoordination for transaction coordination: *atomic transaction* and *business activity*. Atomic transaction is used to coordinate short duration activities which occur completely or not at all, while business activity coordinate long duration activities.

The atomic transaction coordination type defines the *Completion*, *CompletionWithAck*, *PhaseZero*, *2PC* (two phase commit), and *OutcomeNotification* protocols. An application uses the Completion protocol to tell the coordinator to either try to commit the transaction or abort it. The coordinator can forget about the outcome immediately. The CompletionWithAck has the same meaning but, the coordinator cannot forget about the outcome until it is told explicitly that the application has received the response. about the transaction when an acknowledgement is received. Applications that need to prepare themselves for the completion of a transaction (prior to 2PC) use the PhaseZero protocol. The 2PC makes a presumed abort assumption, which means that no knowledge of a transaction implies it is aborted. The OutcomeNotification allows a participant to find out when a transaction has completed and its outcome.

The business activity protocols differ from the previous one in that they handle long-lived activities. Due to the duration of the activities, the results of the constituent activities are released before the overall activity completes. There are two business coordination protocols: *BusinessAgreeement* and *BusinessAgreeementWithComplete*. In the former protocol a child must know when it has completed all the work for a business activity. In the latter protocol, the parent informs a child when it has received all requests. As we have seen in the first section, when isolation is not guaranteed, mechanisms to reverse the effects of previously completed activities (e.g. compensating transactions, state reconciliation) are introduced. Business activity and atomic transaction protocols can be used in combination. Actions of a long running activity can be short ACID transactions, for instance to implement open nested transactions. Subtransactions would be atomic transactions, whilst the top-level open nested transaction would be a business transaction that in case of failure should compensate the effects of the committed atomic subtransactions.

In any WS-Transaction protocol, parties can receive duplicate notifications. The parties should be prepared to respond back in manner consistent with their current state.

4 Dynamically Adaptable Transactional Storage

Transactions in J2EE are supported by two components: the JTS/JTA and the database. As explained in the previous section, JTS is embedded in the application server and plays the role of transaction manager for BSs. Databases are used to store the transactional persistent state of applications, such as entity beans. The replication of the transaction manager of BSs is described in deliverable D1, since the TM is tightly integrated with the application server. In this document we will deal with the replication of the database and how dynamic adaptability is introduced within the database.

4.1 Implementation of Database Replication Protocols

There are several aspects to be considered in order to implement a database replication protocol. These aspects will have a considerable impact in the scalability of the protocol and other efficiency measurements like number of messages per transaction, response time and abort rate. In this section we will consider some basic implementation issues that must be considered in order to provide an efficient implementation of a database replication protocol.

The type of communication among sites plays a crucial role in the overhead of a replication protocol. The number of messages per transaction varies depending on whether the protocol uses point-to-point communication or multicast messages (group communication primitives) [JPAK01]. The number of messages has an impact in the use of computing power. Therefore, the more messages per transaction, the higher response time and lower throughput. [JPAK01] show, for a set of replication protocols, that the use of multicast reduces on average to the half the number of messages per transaction.

The simple read-one-write-all-available (ROWAA) protocol outperforms quorum protocols in a wide variety of scenarios [JPAK01]. Under this protocol, queries (read operations) are executed at one site. However, write operations are executed at all available sites. A ROWAA protocol does not impose that update transactions must be executed completely at all available sites. According to the number of sites that process an update transaction completely, transaction processing can be symmetric or asymmetric. Under symmetric processing all sites process every single transaction completely. This approach is also called *active replication* in the context of fault-tolerance. In contrast, under asymmetric processing an update transaction is processed at a single site and at the end of the transaction the resulting updates (write set) are propagated to the rest of sites. Experimental results and simulations show that asymmetric processing is essential to achieve scalability when transactions contain updates [KA00a, JPAK01, JPAK02].

The implementation of a replication protocol can be done inside a non-replicated database management system (DBMS) or on top of it. The implementation takes a white, grey or black box approach depending on the modifications required to the DBMS. The white box approach implements replication within the database. It takes a non-replicated database management system (DBMS) and implements a replication protocol within the DBMS. Postgres-R [KA00a, KA00b] is the result of implementing a suite of replication protocols [KA98] within PostgreSQL. The grey and black box approaches implement replication on top of the database at a middleware level. The black box approach, as its name indicates, just uses the standard DBMS interfaces to implement a replication protocol on top of it. It does not require any modification of the DBMS, providing seamless replication of a database. Its main advantage is that once a protocol is implemented it can run on top of any non-replicated DBMS. This approach was taken in [AT02]. On the other hand, the grey box approach assumes that the DBMS provides a couple of services in order to implement a replication protocol. The grey box approach has been implemented in [PJKA00, JPAK02].

Symmetric and asymmetric processing can either use the white and grey box implementations. However, the black box approach requires symmetric processing. Seamless replication pays a price. Since the database is a black box, transactions can only be processed symmetrically, what reduces drastically the

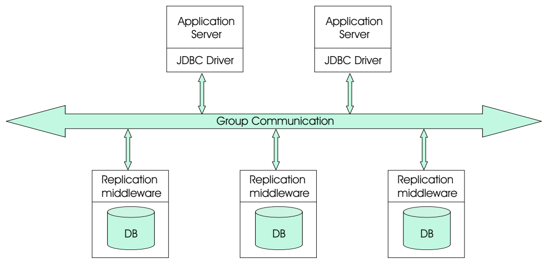

Figure 2: Database Replication Middleware

scalability when transactions include update operations, and additionally, transactions must be executed sequentially in order to guarantee consistency among the sites, what results in a poor performance.

The white box approach provides good scalability when adopting asymmetric processing. Additionally, being implemented within the database it enables a highly efficient implementation. However, the approach is inherently complex, since it requires modifying a DBMS. Finally, the grey box approach tries to combine the best of both, the simplicity of the implementation at a middleware level (black box) and the scalability of the asymmetric processing (white box). The price to be paid is the knowledge needed about which data a transaction will access (to avoid sequential execution of transactions), and the requirement that the database exports two services to get and set the updates of a transaction (for asymmetric processing). The two aforementioned services reduce the seamlessness of the approach.

One of the objectives of ADAPT is to provide scalable replication, what discards the black box approach. The white box approach would require modifying a non-replicated DBMS. This solution is inherently complex and therefore, costly. However, the implementation of the two services needed of the grey box is not as complex as introducing a replication protocol within a non-replicated DBMS. This leaves the grey box approach as the only option, and therefore, the one followed in ADAPT. The replication protocol will be ROWAA, using asymmetric processing and multicast to propagate the write set to all the replicas.

4.2 Replicated Clients

In the architecture for BSs (deliverable D1), the database is always accessed from the application server. The application server will interface the database through JDBC. Given that the application server can be replicated for availability and performance reasons, the replicated database should take into account that its potential clients (e.g., the application server) can be replicated as well. This architecture is depicted in Fig. 2.

We will consider three different replication models for the application server:

- Active replicated application server, that is, all the replicas of the application server submit the same sequence of requests to the database.

- Passive replicated application server. Only one replica of the application server, the primary, executes the requests from all clients, and accesses the database. The primary periodically checkpoints its state and multicast it to the remaining replicas, or backups. If the primary fails, another replica takes over. In case of failover, the new primary might repeat part of the work (i.e., submit duplicate JDBC operations) already performed by the failed primary.

Both models require the handling of duplicate JDBC requests. In the active replication model, a mechanism will be required to identify identical requests from different replicas of the application server. On the database side, these duplicates must be detected and removed. In order to guarantee consistency, the database will require from the replicated application server to send exactly the same sequence of JDBC requests from each replica. In the passive replication model, duplicates can only happen during the fail-over. The requirement imposed by the replicated database in this case is that the replica taking over should identify duplicate requests in the same way the failed replica did. This means that independently of the assumed replication model a duplicate removal mechanism is needed.

Without loss of generality, from now on, we will assume a passive replicated application server (the active model can be seen as a special case). Some of these requests might be duplicated (during failover), but since duplicate requests are filtered out, they do not pose any inconsistency problem.

In the passive replicated approach, a transaction is only tracked by the transaction manager in the replicated application server that executes the transaction. In order to enable fail-over, it is necessary to checkpoint the transaction manager state to the rest of the replicas of the application server. The state corresponding to each transaction can be checkpointed at the end of each method invocation together with the state of the stateful session beans and the set of accessed entity beans. Clients accessing the application manager should be able to perform fail-over in a transactional context without aborting the transaction. This might require providing a modified JTA to the client.

4.3 Dynamic Adaptability

Our concept of dynamic adaptability for replicated databases includes different features:

- Online recovery. The ability to recover new or failed replicas of the database without stopping request processing and disrupting service as minimally as possible.
- Dynamic load balancing. The ability to balance the load at run time, without disrupting service processing.
- Dynamic control of concurrency. The ability to track dynamically the optimal level of concurrency to achieve the maximum throughput.

Online Recovery

The goal of only recovery for replicated databases in the context of ADAPT is to introduce recovery facilities to the replicated database so that the inclusion of failed or new replicas disrupts minimally normal processing. Several protocols with a similar goal have been proposed recently [KBB01, JPA02]. Given that in ADAPT the implementation of database replication will follow a gray box approach (replication protocols are implemented on top of the database), we will use the protocols presented in [JPA02].

Our online recovery protocol basically recovers each table in a more or less independent fashion. This feature is very convenient to prevent the disruption of service processing in the recoverer replica (the one that sends the state to the new replica). Additionally, the protocol uses the logged updates for recovery instead of the database itself, what uncouples the recovery processing from the request processing. This is very beneficial since the recoverer replica can continue applying updates even to the table being recovered. Recovery is also uncoupled from the underlying group communication state transfer. This is important, since during the state transfer group communication is blocked, what would result in stopping servicing requests during this period.

Another aspect worth to mention is that of simultaneous and cascading recoveries. When recovering a set of sites in a cluster, they can restart simultaneously (very unlikely) or in a cascading fashion. In the case of simultaneous recovery the algorithm takes advantage of the group communication primitives and perform recovery of all the replicas simultaneously. Cascading recoveries are more difficult to deal with. In this case, the protocol takes advantage of the fact that recovery is performed on a per-table basis. Thereby, when a replica starts recovery, whilst a recovery is underway, it joins the recovery process in the next table that is recovered. For instance, let us assume that there is a database with 10 tables. Let us also assume that one replica has recovered tables 1-3 and it is recovering the fourth table when the new replica starts recovery. The new recovering replica will join the recovery process when the recovery of the fifth table starts and continue it till the former replica ends it. Then, the new recovering replica will continue alone the recovery of the first four tables. This approach to recovery prevents redundancies that would take place if cascading recoveries are performed independently. Additionally, the recovery process is fault-tolerant. If the recoverer replica fails, another working replica takes over the recoverer role and continues the recovery process with the existing recovering replicas.

Finally, the protocol for online recovery is adaptable in the sense that it can adapt the resources devoted to recovery according to the spare computing capacity in the replicated database. We will later discuss the importance of this issue in the context of load balancing.

Dynamic control of concurrency

A database, as any other software with finite resources, is able to increase its throughput with an increasing load till a threshold. Once this threshold is reached, the lack of resources yields to a trashing behavior, reducing the potential throughput. Unfortunately, this threshold changes dynamically with the workload, that is, it cannot be set at configuration time.

The proposed middleware for database replication will keep a pool of connections with the underlying database, PostgreSQL. The number of active transactions in the database can be controlled dynamically by increasing or reducing the number of connections used in parallel. The question is how to determine the optimal degree of concurrency since it is a moving target. Our plan consists in running a set of experiments to determine the database behavior under different workload conditions. From these experiments an analytical model will be synthesized. The middleware will track the behavior of the system and determine, by applying the analytical model, at which region of the throughput curve the system is. If the trashing threshold has not been reached, the degree of concurrency can be increased. Otherwise, if that threshold has been surpassed, the number of active transactions should be reduced. This means that the replication middleware will adapt itself dynamically to reach the optimal level of concurrency at each replica.

Load Balancing

The database load balancing will be dynamic. The replication middleware should detect which replicas become overloaded and distribute their load to replicas with spare computing resources. Since each replica is dynamically adapting its optimal level of concurrency, overload cannot be detected by a trashing behavior. The number of pending requests will give an estimation of the load at a given replica. Therefore, the load balancing protocol will monitor the number of pending reques and the average response time to detect the overload of replicas. A replica whose response time increases, it is becoming overloaded, what will result in an increase of the number of pending request. The load balancing protocol will decide that some pending request at an overloaded replica can be executed at other replica to reduce the load of that overloaded replica.

Since all the request to the replication middleware are sent to all replicas, and update transactions must sent their write set to all the replicas, every replica has a precise knowledge of the behavior of the other replicas without resorting to extra messages for load balancing purposes.

5 Transactional Composite Services

Our initial goal was to propose a new advanced transaction model or identify a suitable existing one. However, from our study of advanced transaction models and CSs the conclusions we have drawn are the following:

- Some of the available advanced transaction models are suitable for specific composition scenarios.
- It seems that the different needs for transactional CSs cannot be covered by a single advanced transaction model. Current specifications for transaction coordination protocols consider that there is not a single transaction model suitable for CS.

In the light of these conclusions, it seems more adequate to provide a generic framework able to express the different kinds of transactional dependencies among the components of a CS. This approach would have the advantage of both being able to express a wide variety of existing advanced transaction models and expressing new models if required.

The support for transactional CSs includes a visual tool for composing CSs and the addition of transactional semantics. This tool will provide an integrated view of transactions and CSs. The tool

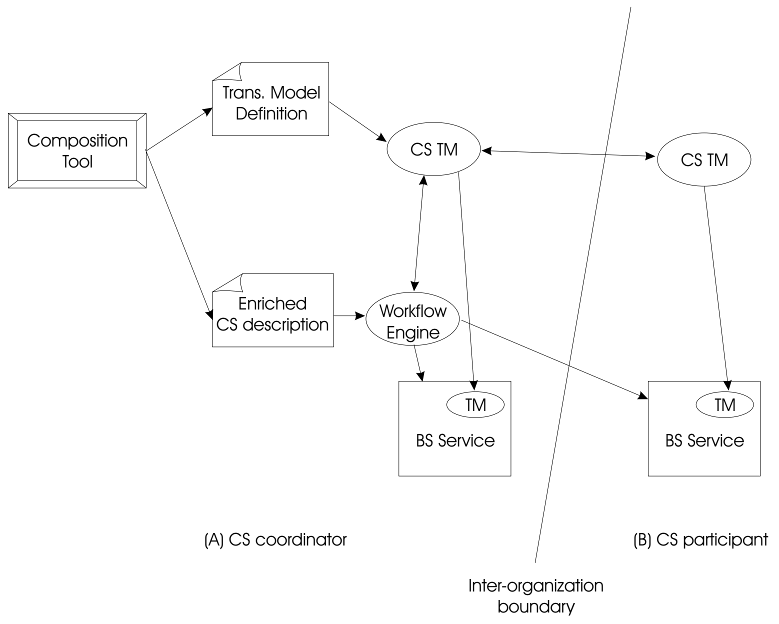

Figure 3: Transactional Composite Services

will be in charge of generating code for transactional CSs and enforcing the corresponding transactional semantics. The run-time support is provided by a transaction manager (TM) for CSs which implements the transaction coordination protocols run among the web services visited by a transactional CS.

5.1 Our Vision for Transactional CSs

ADAPT will provide a visual tool (part of deliverable D14) to ease the development of CSs. This tool will use a workflow-like visual language to create CSs. Since CSs can be transactional, the environment will support the notion of transactions as well. As argued before the transactional support should be flexible enough so that, a CS can choose the most adequate transaction model.

Despite a good deal of advanced transaction models can be found in the literature, only a few implementations of them exist. What is more, most of these implementations only support a single advanced transaction model. A few implementations have tackled with the problem of providing flexible support for advanced transaction models [BP97, BDGR94]. In all these flexible approaches transactional semantics must be added programmatically. The disadvantage of programmatic advanced transactions is that due to their complexity, they require skills beyond the ones of an application programmer. For this reason, we have decided to take a different path and adopt a higher level approach based on declarative transactions in ADAPT. We use the term declarative transactions in the same sense as in container managed transactions for EJBs in the J2EE platform. In this platform, it is possible to describe the transactional semantics separated from the EJB code, in the bean deployment descriptor. By adopting a declarative approach to add transactional semantics to CSs, the task will be highly simplified and automated.

However, declarative advanced transactions pose several difficulties:

1. Some advanced transactional models impose a sequencing of the activities to be executed (e.g. a Saga imposes a sequential execution of its activities). Since a CS already has its own structure, structural compatibility issues might raise.

2. In order to provide flexible support, the programming environment should provide a means to define new advanced transaction models, as well as to apply them to a particular CS.

3. Some transaction models impose requirements on their components, such as that an activity should be compensatable, retriable, or testable (can be tested whether the transaction was executed). Again, this fact might raise compatibility issues.

4. Finally, many workflow engines (the engine supporting the CS execution) do not provide any transactional support or just support a particular advanced transaction model.

We believe that the above problems should be addressed in order to provide a realistic approach to declarative transactions in CSs. What is more, we think that the visual composition tool should have an analysis component to determine the compatibility of a particular transaction model with the CS to which is applied.

Finally, it seems unrealistic to assume that workflow engines will provide the required flexible support for advanced transactions. Therefore, it should be possible to enrich CSs with advanced transactions based on non-transactional workflow engines. Our approach to solve that problem is to generate an enriched CS which will include all the code needed to enforce the advanced transaction model, such as interactions with the CS TM (e.g. to notify about transaction life cycle events).

5.2 Description of Advanced Transaction Models

A programming language is required in order to enable the description of advanced transaction models. This language should be able to express the different kinds of dependencies that characterize the different advanced transaction models. A visual version of this language will be provided by the visual integrated programming environment. From the visual description of the advanced transaction model, the visual tool will generate a textual representation to feed the CS transaction manager.

Some research has been done in order to identify the dependencies that appear in the different advanced transaction models. The Acta framework [CR90, Chr91, CR91] is an example of such a research. Acta is a first-order logic-based specification language for the description of advanced transaction models. Acta identifies the important events in a transaction life cycle (such as begin, abort, and commit) and allows establishing dependencies among these events. By establishing dependencies among these events, it becomes possible to express structural aspects such as serial execution, to determine which executions are successful (e.g., commit and abort dependencies), to specify when compensation is required (e.g. force-commit-on-abort), or to describe alternative execution (e.g., exclusion dependency). We will borrow from Acta those kinds of dependencies that are useful for describing transactional CSs.

Acta is also capable of expressing isolation requirements. However, since CSs deal with long running transactions and autonomous entities, isolation cannot span beyond the scope of a single organization. For this reason, we will focus solely on atomicity properties.

5.3 Flexible Transaction Management Support for CSs

In order to provide a flexible transaction support, we will develop a transaction manager (TM) for CSs capable of expressing arbitrary dependencies among transactions. The TM will be fed with a transaction model description file generated by the visual composition tool. This description will enable the TM to coordinate the different component activities according to the semantics of the advanced transaction model.

The TM will interact with different entities, as shown in Fig. 3, and coordinate transactional CSs. The coordination of transactional CSs accessing either BSs or other CSs requires a standard for transactional context propagation and protocols for transaction coordination (such as WSTransaction or BTP). Currently there is no clear trend about which of these specifications will become widely accepted. On one hand, BTP is a standard, on the other hand, WS-Transaction is supported by a large enterprise consortium, and therefore, it might become a (de facto) standard. We will postpone this decision until a clear trend is observed.

For CSs started locally, the TM will provide an interface to the workflow engine to express these dependencies. Additionally, the TM will coordinate the global transaction enclosing the CS. For CSs started externally, the TM captures the transactional context, coordinates the local transaction with the local BSs, and participates in the coordination of the global transaction with the caller. In Fig.3, the two roles a TM can play in a CS are depicted: the role of the coordinator of the CS, that is, the site running the CS, is shown on the left side, the role of a participant in the CS shown on the right side.

References

[AAA$^+$96] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced Transaction Models in Workflow Contexts. In *IEEE 12th Int. Conf. in Data Engineering*, March 1996.

[AM97] G. Alonso and C. Mohan. Workflow Management: The Next Generation of Distributed Processing Tools. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*, chapter 2, pages 35–59. Kluwer Academic Publishers, 1997.

[AT02] Y. Amir and C. Tutu. From Total Order to Database Replication. In *Proc. of Int. Conf. on Distr. Comp. Systems (ICDCS)*, July 2002.

[BDGR94] A. Biliris, S. Dar, N. Gehani, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 44–54, 1994.

[BP97] R. Barga and C. Pu. A Reflective Framework for Implementing Extended Transaction Models. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[BzHG92] A. Buchmann, M. T. Özsu, M. Hornick, and D. Georgakopaulus. A Transaction Model for Active Distributed Object Systems. In A. K. Elmagarmid, editor, *Database Transaction Models*, chapter 5, pages 123–158. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[CCC$^+$02] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, and S. Thatte. Web Services Transaction (WS-Transaction), 2002. http://www.ibm.com/developerworks/library/ws-transpec/.

[CCF$^+$02] F. Cabrera, G. Copeland, T. Freund, J. Klein, D. Langworthy, D. Orchard, J. Shewchuk, and T. Storey. Web Services Coordination (WS-Coordination), 2002. http://www.ibm.com/developerworks/library/ws-coor/.

[Chr91] P. K. Chrysanthis. *ACTA: A Framework for Modelling and Reasoning about Extended Transactions*. PhD thesis, Dept. of computer and Information Science, Univ. Of Massachusetts, 1991.

[CR90] P. K. Chrysanthis and K. Ramamritham. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 194–203, 1990.

[CR91] P. K. Chrysanthis and K. Ramamritham. A Formalism for Extended Transaction Models. In *Proc. of the 17th Int. Conf. On Very Large Data Bases*, pages 103–112, Barcelona, Spain, 1991. Morgan Kaufmann Publishers.

[DA93] P. Dasgupta and R. Ananthanarayanan. Special Issue on Workflow and Extended Transaction Systems. *IEEE Data Engineering Bulletin*, 16(2):3–56, June 1993.

[ELLR90] A. K. Elmagarmid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proc. of the 16 th Int. Conf. On Very Large Data Bases*, pages 507–518, Brisbane, Australia, 1990. Morgan Kaufmann Publishers.

[Elm92] A. K. Elmagarmid, editor. *Database Transaction Models*. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[EV88] F. Eliassen and J. Veijalainen. A Functional Approach to Information System Interoperability. In *Proc. of EUTECO 88. Research into Networks and Distributed Applications*, pages 1121–1135. North-Holland, 1988.

[GMS87] H. García-Molina and K. Salem. SAGAS. In *Proc. of the ACM SIGMOD Int. Conf. On Management of Data*, pages 249–259, 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[HLR$^+$01] Iain Houston, Mark C. Little, Ian Robinson, Santosh K. Shrivastava, and Stuart M. Wheater. The CORBA activity service framework for supporting extended transactions. *Lecture Notes in Computer Science*, 2218, 2001.

[JK97] S. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[JPA02] R. Jiménez-Peris, M. Patiño-Martínez, and G. Alonso. Non-Intrusive, Parallel Recovery of Replicated Data. In *IEEE Symp. on Reliable Distributed Systems*, pages 150–159, 2002.

[JPAK01] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. How to Select a Replication Protocol According to Scalability, Availability, and Communication Overhead. In *Proc. of the Int. Symp. on Reliable Distributed Systems (SRDS)*, pages 24–33, New Orleans, Louisiana, October 2001. IEEE Computer Society Press.

[JPAK02] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme. Scalable Database Replication Middleware. In *Proc. of 22nd IEEE Int. Conf. on Distributed Computing Systems, 2002*, pages 477–484, Vienna, Austria, July 2002.

[KA98] B. Kemme and G. Alonso. A Suite of Database Replication Protocols based on Group Communication Primitives. In *Proc. of IEEE ICDCS*, pages 156–163, 1998.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, A new way to implement Database Replication. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, Cairo, Egypt, September 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, September 2000.

[KBB01] B. Kemme, A. Bartoli, and O. Babaoglu. Online Reconfiguration in Replicated Databases Based on Group Communication. In *Proc. of the Int. Conf. on Dependable Systems and Networks (DSN 2001)*, Goteborg, Sweden, June 2001.

[KS95] N. Krishnakumar and A. P. Sheth. Managing heterogeneous multi-system tasks to support enterprise-wide operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, Cambridge, MA, 1985.

[MRSK92] S. Mehrotra, R. Rastogi, A. Silberschatz, and H. F. Korth. A Transaction Model for Multidatabase Systems. In *Proc. of the 12th Int. Conf. on Distributed Systems*, pages 56–63, June 1992.

[OAS02] OASIS. Business Transaction Protocol, 2002. http://www.oasis-open.org/committees/business-transactions.

[OMG] OMG. *CORBA services: Object Transaction Service (OTS)*. OMG.

[OMG02] OMG. *Additional Structuring Mechanisms for the OTS. Version 1.0*. OMG, September 2002.

[PJKA00] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, October 2000.

[RS92] A. Reuter and K. Schneider. The ConTract Model. In A. K. Elmagarmid, editor, *Database Transaction Models*, chapter 7, pages 219–263. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[RS97] A. Reuter and K. Schneider. Contracts Revisited. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*, chapter 5, pages 127–151. Kluwer Academic Publishers, 1997.

[SR93] A. Sheth and M. Rusinkiewicz. On Transactional Workflows. *IEEE Data Engineering Bulletin. Special Issue on Workflow and Extended Transaction Systems*, 16(2):37–40, 1993.

[Sun] Sun. *Java Transaction Service*. http://java.sun.com/products/jts/.

[SW90] S. K. Shrivastava and S.M. Wheater. Implementing Fault-Tolerant Distributed Applications Using Objects and Multi-Coloured Actions. In *Proc. 10th Int. Conf. on Distributed Computing Systems*, pages 203–210, 1990.

[WS91] G. Weikum and H. Schek. Multi-level Transactions and Open Nested Transactions. *IEEE Data Engineering Bulletin*, pages 60–64, March 1991.

[WS92] G. Weikum and H. J. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions. In A. K. Elmagarmid, editor, *Database Transaction Models*, chapter 13, pages 515–554. Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[WS97] D. Worah and A. P. Sheth. Transactions in transactional workflows. In S. Jajodia and L. Kerschberg, editors, *Advanced Transaction Models and Architectures*, chapter 1, pages 3–34. Kluwer Academic Publishers, 1997.

[XOp96] XOpen. *Distributed Transaction Processing: Reference Model*. X/Open Company, 1996.

[ZNBB94] A. Zhang, M. Nodine, B. Bhargava, and O. Bukhres. Ensuring Relaxed Atomicity for Flexible Transactions in Multidatabase Systems. In *Proc. of Int. Conf. on Management of Data*, pages 67–78, 1994.