TRANSACTION SERVICE COMPOSITION
A Study of Compatibility Related Issues

Anna-Brith Arntsen and Randi Karlsen
Computer Science Department, University of Tromsoe, 9037 Tromsoe, Norway

Keywords: Flexible Transaction Processing Environment, Dynamic service composition, Compatibility, Integration.

Abstract: Different application domains have varying transactional requirements. Such requirements must be met by applying adaptability and flexibility within transaction processing environments. `ReflecTS` is such an environment providing flexible transaction processing by exposing the ability to select and dynamically compose a transaction service suitable for each particular transaction execution. A transaction service (*TS*) can be seen as a composition of a transaction manager (*TM*) and a number of involved resource managers (*RMs*). Dynamic transaction service composition raises a need to examine issues regarding `Vertical Compatibility` between the components in a *TS*. In this work, we present a novel approach to service composition by evaluating `Vertical Compatibility` between a *TM* and *RMs* - which includes `Property` and `Communication` compatibility.

1 INTRODUCTION

New application domains and execution environments have transactional requirements that may exceed the traditional ACID properties. Such domains, including workflow, cooperative work, medical information systems and e-commerce, are constantly evolving and possess varying and non-ACID requirements. The travel arrangement scenario is a well-known example of a long-running transaction with requirements that goes beyond the ACID properties. Such a transaction consists of a number of subtasks (booking flights, hotel rooms, theater tickets, etc), possible with adjacent contingent transactions and of dissimilar importance (vital vs. non-vital). Resources cannot be locked for the entire duration of the transaction. So, to increase performance and concurrency, this transaction must be structured as a non-ACID transaction with relaxed (i.e. semantic) atomicity based on the use of compensating activities in case of failure.

Varying transactional requirements demand a flexible transaction execution environment. Such requirements are not met by current transaction processing solutions where merely ACID transactions are supported. Thus, there is a gap between offered and required support for varying transactional requirements.

A number of advanced transaction models (Elmagarmid, 1992; Garcia-Molina and Salem, 1987) have been proposed to meet different transactional requirements. Many advanced models where suggested with specific applications in mind, and with fixed transactional semantics and correctness criteria. Consequently, they do not provide sufficient support for wide areas of applications.

The characteristics of the proposed transaction models support our conviction that the ”one-size fits all” paradigm is not sufficient and that a single approach to extended transaction execution will not suit all applications. To close the gap between offered and required support for varying requirements, we designed the flexible transaction processing platform `ReflecTS` (Arntsen and Karlsen, 2005). `ReflecTS` is a highly adaptable platform offering an extensible number of concurrently running transaction services, where each service supports different transactional guarantees.

Generally, a transaction service (*TS*) can be viewed as a composition of a transaction manager (*TM*) and a number of resource managers (*RMs*),

Arntsen A. and Karlsen R. (2007).
TRANSACTION SERVICE COMPOSITION - A Study of Compatibility Related Issues.
In *Proceedings of the Ninth International Conference on Enterprise Information Systems - DISI*, pages 239-245
DOI: 10.5220/0002392602390245
Copyright © SciTePress

one for each involved source. Today's systems keeps mainly one *TM*, giving a predefined and static *TS* composition. ReflecTS, on the other hand, exposes the ability to dynamically select a *TM* and subsequently compose a *TS* suiting particular transactional requirements. Dynamic service composition raises a need to evaluate Vertical Compatibility between each *TM* - *RM* pair. This must be done both with respect to Property and Communication compatibility. The goal of this paper is to investigate these issues, with a particular focus on Property compatibility and problems related to the integration of heterogeneous commit and recovery protocols.

In the remainder of this paper we first, in section 2, present the architecture of ReflecTS. Section 3 presents the service composition procedure and compatibility related issues. Section 4 follows with related work, and section 5 draws conclusions and presents future work.

2 REFLECTS

ReflecTS (Arntsen and Karlsen, 2005) is a flexible and adaptable transaction processing system suiting varying transactional requirements by providing an extensible number of transaction managers (*TMs*).

The main functionalities of ReflecTS are transaction manager (*TM*) selection, transaction service (*TS*) composition and transaction activation. We present the architecture of ReflecTS and the specifications involved in *TS* composition and compatibility evaluation.

2.1 Architecture

ReflecTS, shown in Figure 1, is a composition of components. TSInstall handles requests for *TM* configurations and reconfigurations, and TSactivate handles requests for transaction executions. The TM-Framework hosts the *TM* implementations, and the InfoBase keeps *TM* and resource manager (*RM*) descriptors and results from the compatibility evaluation procedures.

The IReflecTS interface (Arntsen and Karlsen, 2005) defines the interaction between the application program (AP) and ReflecTS, and is called to demarcate global transactions and to control the direction of their completion. Its design has been influenced by the TX-interface defined in the X/Open Distributed Transaction Processing (DTP) model (Group, 1996), but differ from it to conform to varying transactional requirements. This is, among others, done by including the transactional requirements and information about requested *RMs* in the *TransBegin()* request. The interaction between a *TM* and a *RM* is generally determined by the X/Open standard and the XA-interface (Group, 1996).

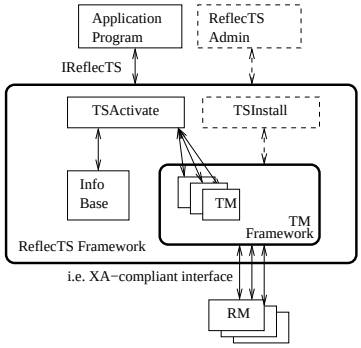

Figure 1: Overview of ReflecTS.

Applications initiating *TransBegin()* embeds a XML-document describing the transactional requirements and a list of requested *RM* identifications. Based on the transactional requirements and descriptors of available *TMs*, a suitable *TM* is **selected** for the transaction execution. The mapping of requirements to a *TM* need not be one-to-one. A specific set of requirements can be mapped to different *TMs*, in which case a list is stored in the system for use if incompatibility arises. Consecutively, the *TM* is **composed** together with required *RMs* into a *TS*. This *TS* is responsible for coordinating the execution of the particular transaction while preserving the requested transactional requirements.

TSactivate performs *TS* composition based on the descriptor of the selected *TM*, TM_Descriptor, and the descriptors associated with involved *RMs*, RM_Descriptors. For each pair of *TM* and *RM*, compatibility is evaluated. When this compatibility, which includes Property and Communication compatibility, is fulfilled between the involved parties - composition takes place, and eventually, the transaction is started.

Transaction **activation** presupposes successful evaluation of Horizontal Compatibility, which is compatibility between concurrently active transaction services. This is part of future work.

2.2 Transaction Service Specifications

ReflecTS introduces two specifications sustaining service composition and compatibility evaluation: the TM_Descriptor and the RM_Descriptor.

2.2.1 TM_Descriptor

The TM_Descriptor describes a *TM* and includes information about: 1) a TM_ID 2) transactional properties (ACID or non-ACID), 2) transactional mechanisms (commit/recovery, global concurrency control), and 3) compatibility with a standard (i.e. XA) or not.

The following is an example of a *TM* with ACID guarantees running a 2PC protocol with presumed abort, supporting the XA-interface:

```
<TM_Descriptor>
   <ServiceID>TM_ID</ServiceID>
   <Properties>ACID</Properties>
   <Standard>XA</Standard>
   <TaskList>
     <Task>
       <TaskId>2PC</TaskId>
       <TaskParameters>
         <Parameter>PrA</Parameter>
       </TaskParameters>
     </Task>
   </TaskList>
</TM_Descriptor>
```

2.2.2 RM_Descriptor

A RM_Descriptor holds information about a registered *RM* and includes the following: 1) a resource identification RM_ID, 2) ResourceID with the DNS name of the resource, 3) whether the *RM* is XA-compliant or not, and 4), information about the *RMs* transactional mechanisms.

The specification of ResourceID corresponds to the content of the *RMlist* following the *Start_Trans()* request. An example of a XA-compatible *RM* running a two-phase commit protocol with presumed abort (PrA) follows.

```
<RM_Descriptor>
   <RM_ID>Resource_ID</RM_ID>
   <ResourceID>mypc.mydom.edu</ResouceID>
   <Standard>XA</Standard>
   <TaskList>
     <Task>
       <TaskId>commit</TaskId>
       <TaskParameters>
         <Parameter>PrA</Parameter>
       </TaskParameters>
     </Task>
   </TaskList>
</RM_Descriptor>
```

3 SERVICE COMPOSITION AND COMPATIBILITY

3.1 Introduction

A transaction service *TS* is a composition of a transaction manager *TM* and a set of n compatible resource managers $TS = (TM, \mathcal{RM})$ where $\mathcal{RM} = \{RM_1, \ldots, RM_n\}$. To compose a *TS*, Vertical Compatibility must be successfully evaluated with respect to the following:

- Each pair of *TM* and *RM* must match with respect to local and global transaction control to assure the requested transactional requirements. This we refer to as Property compatibility
- Each pair of *TM* and *RM* must be able to communicate through some common interface while assuring requested transactional requirements. This we refer to as Communication compatibility

Results from these evaluations are kept in the InfoBase. This means that it is unnecessary to repeatedly evaluate the same pair of *TM* and *RM*.

3.2 Composition Procedure

This section presents the service composition procedure, TS_Composition(), which follows *TM* selection.

Code regarding selection, service composition and incompatibility is presented below. R_T represents the transactional requirements of a transaction *T*, and *RMlist*, the list of *RMs* requested by *T*. The procedure SelectTM() takes as input the TM_Descriptors of the deployed *TMs* and the transactional requirements R_T, and returns a list of *TMs* (*TMlist*) assuring R_T. In the case of unsuccessful composition, another *TM* may be selected, and TS_Composition() restarted. This sequence is repeated either until the composition completes or there are no available *TMs* in the list. If the composition does not complete, incompatibility is managed by the ResolveIncomp() procedure (see 3.5). ResolveIncomp() takes as input a list of compatible *RMs* (*Complist*), returned from the TS_Composition() procedure. If there is no solution to the incompatibility problem, the procedure stops.

```
TMlist = SelectTM(TM_Descriptor[]* TM, R_T)
while TS not composed {
   if TS_Composition(TM, RMlist, R_T, Complist)
     {
       Compose and return TS
     } else {
       if More TM's available {
         TM = ChooseNewTM(TMlist)
       } else {
         if ResolveIncomp(TM, Complist, R_T)
         {
           Compose and return TS
         } else STOP
     }
```

```
    }
  }
```

In the `TS_Composition()` procedure following below, the procedures `Comm()` and `Property()` are initiated for each involved *RM*. Based on the results of these calls, `InfoBase` and `CompList` are updated with information about compatible *TM-RM* pairs. The `Complist` contains information regarding the transaction service composition for use while managing incompatibility.

```
boolean TS_Composition(TM_Descriptor*
 TM, RM_ID[]* RMlist, R_T) {
  for (each RM_i in RMlist) {
    if Comm(TM,RM_i,R_T) {
      if Property(TM,RM_i) {
        Update Infobase
        Update Complist
        return true
      }
    } else return false
  }
}
```

3.3 Property Compatibility

Formally, a composition of a transaction service TS for the execution of a transaction T over a set of compatible resource managers $\mathcal{RM}^T$ is denoted $TS^T = (TM_i, \mathcal{RM}^T)$, where $\mathcal{RM}^T = \{RM_1, \ldots, RM_M\}$ and where the transaction manager TM_i is selected for the specific transaction T. TS^T has the properties $\mathcal{P}(TS^T)$. The principal goal of TS^T is to coordinate the execution of transaction T while assuring the requested transactional requirements R_T.

Evaluating property compatibility involves examining the **transactional mechanisms** of the TM_i with the corresponding mechanisms of each involved RM_j with the aim to assure the requested transactional requirements.

3.3.1 Transactional Mechanisms

The main transactional mechanisms of *TMs* and *RMs* are commit/recovery and concurrency control. Generally, a *TM* controls global commit/recovery and global concurrency control, and a *RM* performs local transaction control (logging, concurrency control, persistency, commit/recovery). When heterogeneous commit protocols cooperate for the commitment of a distributed transaction, problems and incompatibility may arise. In this work, we focus on integration of heterogeneous commit protocols, and leaves concurrency control issues for future work.

Traditionally, two-phase commit (2PC) is the protocol used to ensure the ACID properties of distributed transactions, and the X/Open Distributed Transaction Processing (DTP) model (Group, 1996) is the most widely used standard implementing the 2PC protocol.

Some optimizations of the 2PC protocol are proposed. For instance presumed abort (PrA), presumed commit (PrC), presumed nothing (PrN), presumed any (PrAny), and three phase commit (3PC) (Gupta et al., 1997; Tamer and Valduriez, 1999; Al-Houmaily and Chrysanthis, 1999). These 2PC optimizations ensures atomic commit of global transactions. Other optimizations ensures weaker notions of atomicity, e.g. semantic atomicity. These include for instance the Optimistic 2PC (O2PC) protocol (Levy et al., 1991), the OPT (Gupta et al., 1997) protocol, and one-phase commit (1PC). Compared with 2PC, 1PC omits the first phase, thereby permitting immediate commit of subtransactions. Consequently, one-phase commitment is feasible in a X/Open environment. A combination of 1PC and 2PC protocols is realized in the dynamic 1-2PC protocol (Al-Houmaily and Chrysanthis, 2004). The 1-2PC protocol switches from 1PC to 2PC when necessary.

A prerequisite for a *RM* participating in a 2PC protocol is to support a visible prepare-to-commit state. *RMs* not supporting prepare-to-commit are not able to participate in a 2PC protocol and can thus not contribute in assuring global atomicity. Integrating the different commit protocols may cause problems and a visible prepare-to-commit may not be enough for a practical integration of commit protocols. For instance, in (Al-Houmaily and Chrysanthis, 1999) they show that it is impossible to ensure global atomicity of distributed transactions executed at both PrA and PrC participants if PrA, PrC or PrN is running at the transaction manager. Consequently, they presented PrAny, which is a protocol that successfully integrates PrN, PrA and PrC.

3.3.2 Evaluating Property Compatibility

Assume a transaction T with the set of transactional requirements R_T. If each RM_j requested by the transaction can participate in the selected TM_i's commit and abort protocols so that the requirements R_T of T are assured, the TS^T with the properties $\mathcal{P}(TS^T)$ will be composed. The requirements R_T and the properties $\mathcal{P}(TS^T)$ need not be equivalent.

The following definition must be sustained.

Definition 1: (Assured transactional requirements). The set of transactional requirements R_T of a transaction T is assured if and only if

R_T is semantically a subset of the set of transactional properties $\mathcal{P}(TS)$ of the transaction service TS^T, where TS^T=$(TM_i, \mathcal{RM}^T)$ and TM_i is the manager selected for this particular transaction T:

$$R_T \subseteq^s \mathcal{P}(TS^T)$$

The definition states that the set of transactional requirements R_T of the transaction T must be a *semantic subset* of the set of properties $\mathcal{P}$ of TS^T such that every element of the set R_T is semantically contained in the set $\mathcal{P}(TS^T)$. This definition is used during evaluation of property compatibility. If the equation does not hold and $R_T \nsubseteq^s \mathcal{P}(TS^T)$, the composition will not take place.

Definition 1 must hold for each *TM - RM* combination, and it must hold even though the *TM* changes or *RMs* are added. Consider TS^T=$(TM_i, \mathcal{RM}^T)$ where $\mathcal{RM}^T = \{RM_1, \ldots, RM_{i-1}\}$ and $R_T \subseteq^s \mathcal{P}(TS^T)$. Assume adding the resource manager RM_i so that $\mathcal{RM}^T = \mathcal{RM}^T \bigcup \{RM_i\}$. Then, according to definition 1, transaction manager TM_i and the resource manager RM_i are property compatible for the execution of T if and only if $R_T \subseteq^s \mathcal{P}(TS^T)$.

According to definition 1 and deduced from our perception, a *semantic subset* refers to a set of transactional properties belonging a transaction service that are powerful enough to assure a specific set of transactional requirements.

To illustrate the *semantic subset* relationship, consider a transaction service (TS_1) having the following set of properties: $\mathcal{P}(TS_1)$ = (A), where A refers to the atomicity property. Assume a set of requirements deduced from a particular transaction specification: R_T = (SA^{saga}), which refers to semantic atomicity as supported by Sagas (Garcia-Molina and Salem, 1987). TS_1 assures atomicity by implementing a variant of 2PC or a 3PC protocol. Since these protocols are able to commit individual transactions one-phase as is required to assure semantic atomicity, TS_i guarantees R_T, $(SA^{saga}) \subseteq^s (A)$, and definition 1 is fulfilled. If the transaction require full atomicity, R_T = (A), definition 1 still holds as TS_1 assures atomicity, and $(A) \subseteq^s (A)$.

Next, consider a service TS_2 with the properties $\mathcal{P}(TS_2) = (SA^{saga})$ - semantic atomicity. This service implements a Sagas-like commit protocol supporting compensation. The resource managers of the composition may implement either a 2PC variant, 1PC, or just committing transactions as soon as they are finished (like in for instance a web service). If a transaction requires semantic atomicity $R_T = (SA^{saga})$, the equation $(SA^{saga}) \subseteq^s (SA^{saga})$ is fulfilled and the requirements guaranteed. If a transaction requires ACID, the service TS_2 will not be composed for the transaction unless incompatibility can be solved (see section 3.5).

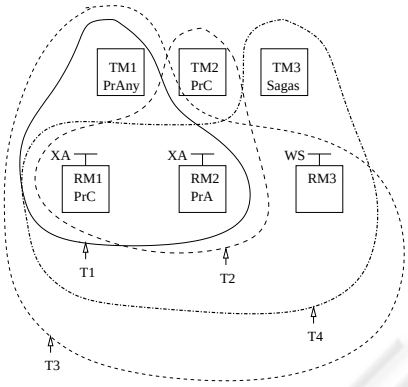

Figure 2: Transaction Examples.

3.3.3 Exemplifying Property Compatibility

Assume an environment with three transaction managers, TM_1, TM_2 and TM_3. TM_1 assures ACID by implementing PrC, TM_2 implements PrAny, and TM_3 assures relaxed atomicity as required by Sagas. The environment also includes three resource managers, RM_1 running PrC, RM_2 running PrA, and a *web service*, RM_3, not supporting prepare-to-commit. Figure 2 illustrates this environment with four proposed transaction services composed for four different transactions. These are denoted `T1` to `T4`, and are surrounded with drawn lines.

First, consider a transaction `T1` that requests ACID and the resources RM_1 and RM_2. For `T1`, TM_1 implementing PrAny is selected. RM_1 implements PrC and RM_2 PrA. As seen in (Al-Houmaily and Chrysanthis, 1999), this combination implies compatibility as PrAny successfully integrates both PrC and PrA.

Next, a transaction `T2` requests the same properties and resources as `T1`, namely ACID and RM_1 and RM_2. However, for `T2`, TM_2 implementing PrC is selected. We know from (Al-Houmaily and Chrysanthis, 1999) that atomicity cannot be guaranteed when a PrC protocol controls the execution of transactions over PrA and PrC protocols. In fact, *Definition 1* will prevent this service from being composed. Instead, the procedure managing incompatibility will be initiated, and the problem can be solved by for instance reconsidering the choice of *TM* (see 3.5).

Transaction `T3` requests ACID and the resources RM_1, RM_2 and RM_3. TM_1 is selected for the execution. RM_3 does not support prepare-to-commit, so PrAny implemented by TM_1 cannot control the execution of `T3`. Consequently, global atomicity cannot be assured, incompatibility exists and the procedure managing incompatibility will be initiated.

The fourth transaction, T4, is a Sagas requesting semantic atomicity and the execution over RM_1, RM_2, and RM_3. For T4, TM_3 is selected. The Sagas-like commit protocol implemented by TM_3 require the underlying resources to respond to immediate commit of individual transactions. This is assured by the involved *RMs*, compatibility is present and the requirements are assured.

3.4 Communication Compatibility

Communication compatibility evaluates the communication capabilities of a specific *TM* - *RM* pair. The ultimate goal is to assure requested transactional properties.

The interface implemented by the involved parties determines the ability to communicate. At present, the XA-standard (Group, 1996) defines the most widely used interface. *XA-compatibility* and *non-XA compatibility* is a natural classification of communication capabilities for *TMs* and *RMs*. XA-compatibility in our sense means conformance to the XA-interface, not necessarily assuring specific transactional requirements. In our definition, a XA-compatible *TM* or *RM* can assure either ACID or non-ACID. Non-XA compatible participants may conform to any other standard (or interface), or none at all.

The XA-interface provides the methods necessary for transaction coordination, commitment, and recovery between a *TM* and one or more *RMs*. The XA-interface supports both atomic commit by the use of a 2PC variant and relaxed atomicity by having the ability to perform 1PC.

The `Comm()` procedure (see 3.2) handling communication compatibility, takes as input the particular *TM* and *RM*, and a set of transactional requirements R_T. The `Comm()` procedure discovers XA-compatibility by investigating the *standard* tag of the *TM* and the *RM* descriptor. Then, the transactional requirements, R_T are used in the process of evaluating communication compatibility. Based on R_T, the requirements regarding communication can be deduced. We will see that a specific *TM* - *RM* combination may satisfy a particular R_T set, but not another one.

If both the *TM* and the *RM* are XA-compatible, communication compatibility exists irrespective of the content of R_T. In this case, the *TM* most likely implements a 2PC variant, and the XA-compatible *RM* supports a visible prepare-to-commit state. Consequently, within this communication, both ACID and relaxed (i.e. semantic) atomicity are provided.

In the combination of a XA-compatible *TM* and a non XA-compatible *RM* and when R_T demand relaxed (i.e. semantic) atomicity, communication is most likely satisfied. However, independent of the content of R_T, the descriptors are consulted ahead of the evaluation. If the R_T claims ACID, the communication might be fulfilled even though the *RM* is non-XA compatible. For instance, a non-XA compatible *RM* may be able to support prepare-to-commit.

Consider a non-XA *TM* in combination with a XA *RM*. If the *TM* implements a Saga-like commit protocol and R_T requests semantic atomicity, communication is satisfied.

3.5 Managing Incompatibility

Incompatibility may be detected either during evaluating property or communication compatibility. In each case, the following actions are considered:

- Communication incompatibility: 1) add an adapter (or wrapper) to either *RM* or *TM* to make them conform to each other's interfaces, or 2) choose another TM with different communication characteristics, but with the same transactional guarantees.
- Property incompatibility: 1) choose another TM with different transactional mechanisms, but with the same transactional guarantees, or 2) negotiate to find an alternative way to execute the transaction by either modifying the transactional requirements or the list of involved resources.

4 RELATED WORK

Today's transaction processing platforms supports the execution of distributed transactions, but with limited flexibility. Present platforms, like for instance Microsoft Transaction Server (MTS) (Corporation, 2000), Sun's Java Transaction Server (JTS) (Subhramanyam, 1999) provide merely one transaction service with ACID guarantees.

Other approaches support more than one transaction service, although not concurrently. One is given by the CORBA Activity Service Framework (Houston et al., 2001), where various extended transaction models are supported. Others are the WS-transactions (Group, 2004), the OASIS BTP (Little, 2003) specification and the Arjuna XML Transaction Service (Ltd, 2003) describing solutions providing two different transaction services, one for atomic transactions and the other for long-running business transactions.

Flexibility within transactional systems can be found in the works of Barga (Barga and Pu, 1996) and Wu (Wu, 1998), implementing flexible transaction services. Related work on dynamic combination

and configuration of transactional and middleware systems can for instance be found in Zarras (Zarras and Issarny, 1998). References to other works can be found in (Arntsen and Karlsen, 2005). These works recognizes the diversity of systems and their different transactional requirements, and describes approaches to how these needs can be supported.

Our work on the flexible transaction processing environment `ReflecTS`, contrasts previous work in several matters. First, by supporting an extensible number of concurrently running services, and next, by providing dynamic transaction service selection and composition according to the needs of applications.

5 CONCLUSION AND FUTURE WORK

The transactional requirements of advanced application domains and web services environments are varying and evolving, demanding flexible transaction processing. On the basis of the flexible transaction processing platform `ReflecTS`, this work presents a novel approach to dynamic transaction service composition and compatibility related issues. From `ReflecTS` a suitable transaction manager can be selected for a particular transaction execution, and dynamically composed together with requested resource managers into a complete transaction service. To complete transaction service composition, this work evaluates `Property` and `Communication` compatibility between a transaction manager and resource managers. The main contributions of this work are the procedures and the formalisms related to these compatibility issues.

Ongoing and future work includes an in-depth evaluation of local and global transactional mechanisms (including concurrency control) with respect to transaction service composition. Further, ongoing work includes developing rules for managing incompatibility, and future work includes an examination of compatibility related to service activation, `Horizontal Compatibility`.

REFERENCES

Al-Houmaily, Y. J. and Chrysanthis, P. K. (1999). Atomicity with incompatible presumptions. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 306–315, New York, NY, USA. ACM Press.

Al-Houmaily, Y. J. and Chrysanthis, P. K. (2004). 1-2pc: the one-two phase atomic commit protocol. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 684–691, New York, NY, USA. ACM Press.

Arntsen, A.-B. and Karlsen, R. (2005). Reflects: a flexible transaction service framework. In *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*, pages 1–6, New York, NY, USA. ACM Press.

Barga, R. and Pu, C. (1996). Reflection on a legacy transaction processing monitor.

Corporation, M. (2000). The .net framework.

Elmagarmid, A. K., editor (1992). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers.

Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259. ACM Press.

Group, O. (1996). X/open distributed transaction processing: Reference model, version 3.

Group, W. W. (2004). Web services architecture, working draft.

Gupta, R., Haritsa, J., and Ramamritham, K. (1997). Revisiting commit processing in distributed database systems. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 486–497, New York, NY, USA. ACM Press.

Houston, I., Little, M. C., Robinson, I., Shrivastava, S. K., and Wheater, S. M. (2001). The corba activity service framework for supporting extended transactions. *Lecture Notes in Computer Science*, 2218.

Levy, E., Korth, H. F., and Silberschatz, A. (1991). An optimistic commit protocol for distributed transaction management. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 88–97, New York, NY, USA. ACM Press.

Little, M. (2003). Transactions and web services. *Commun. ACM*, 46(10):49–54.

Ltd, A. T. (2003). Web services transaction management (ws-txm) ver1.0.

Subhramanyam, A. (1999). Java transaction service.

Tamer, . M. and Valduriez, P. (1999). *Principles of Distributed Database Systems*. Prentice Hall.

Wu, Z. (1998). Reflective java and a reflective component-based transaction architecture. In *OOPSLA workshop*.

Zarras, A. and Issarny, V. (1998). A framework for systematic synthesis of transactional middleware.