Coordination in Service Oriented Architectures Using Transaction Processing Concepts

Peter Hrastnik
EC3 – Electronic Commerce Competence Center
Donau–City–Strasse 1, A–1220 Vienna, Austria
peter.hrastnik@ec3.at

Werner Winiwarter
University of Vienna
Department of Scientific Computing
Universitätsstrasse 5, A–1010 Vienna, Austria
werner.winiwarter@univie.ac.at

Abstract

Service oriented architectures (SOAs) provide an architectural paradigm to develop and evolve enterprise information systems. A key feature of SOAs is compensability of services. Such service assemblies require high coordination efforts to reliably produce a valid result. Transactional processing concepts are widely used to tackle coordination requirements in tightly-coupled distributed systems like J2EE or CORBA. However, in loosely-coupled systems like SOAs, the use of transaction processing systems is uncommon, although proposals for doing transaction processing in Web services systems (an implementation option for SOAs) exist. In this paper, general aspects of a transaction processing system are introduced that can be reasonably used for the coordination of services in SOAs. We present an approach and a corresponding implementation of a transaction processing system for service oriented architectures that adheres to the above-mentioned characteristics.

1. Introduction

Service oriented architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains [12]. By making resources in a distributed system available as independent composable *services*, SOAs reduce complexity and increase flexibility. Composability of services allows organizations to create (new) applications within their enterprise information systems just by aligning existing services. Please note that the SOA paradigm does not mandate an implementation platform. However, characteristics of Web services (loose coupling, statelessness, etc.) match the SOA nature to a high degree and may be a reasonable option to implement an SOA.

According to [7], services in an SOA follow the following essential key principles:

- *Loose coupling*: Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- *Composability*: Collections of services can be assembled and coordinated to form composite services. There is no technology dictated to implement the (process) logic of such service assemblies in SOAs. Web service orchestration technologies like BPEL4WS are possible as well as simple Perl scripts or Java programs.
- *Service contract*: Services adhere to a communications agreement.
- *Autonomy*: Services should stay as far as possible in control over the logic they implement, i.e. nothing else than the service should be able to influence its logic.

Moreover, services in an SOA tend to have a "coarse grained" nature. That is, a service often encapsulates a set of related business functions and consumes considerable computing resources. Its interface accepts and returns complex data in a single invocation [13].

Using a service assembly requires major efforts in order to deliver a coordinated collective result. Such coordination efforts may be addressed solely in the process logic that assembles the services. However, according to [10], *transaction processing concepts* are a superior option. By managing a group of services, transaction processing concepts guarantee that the group of services achieves a coordinated common, consistent, and mutually agreed outcome. For tightly-coupled systems, such an approach for tackling coordination is common and ubiquitous.

The characteristics an outcome (which is common, consistent, and mutually agreed) must have are defined by some kind of transaction logic. For example, the most commonly

known type of transaction logic is called *ACID* [9]. In case the ACID attributes atomicity, consistency, isolation, and durability are honored, a common, consistent, and mutually agreed outcome is guaranteed.

Transaction processing concepts are preferably used in transaction processing systems, where the system takes care of managing transaction specific tasks and achieving an outcome as described above. Thus, the coordination of a service assembly may be achieved by choosing a particular kind of transaction logic and enacting it by using a transaction processing system that manages the involved services as demanded by the transaction logic.

In this paper, we introduce requirements of a transaction processing system that can be reasonably used in SOAs, regardless of the SOA's implementation technique. Moreover, we present an implementation of such an SOA transaction processing system.

The paper is organized as follows. Section 2 presents relevant related work. The basic requirements an SOA transaction processing system should have are described in Sect. 3. Section 4 outlines the approach to achieve the basic requirements and Sect. 5 shortly discusses a realization of this approach.

2. Related Work

Well-known organizations like SUN, IBM, Microsoft, OASIS, etc. have published proposals that deal with transactional processing in the Web service world: Business Transaction Protocol (BTP) [1], WS–Coordination/WS–AtomicTransaction/WS–BusinessActivity (WS–TX family) [11], and WS Composite Framework (WS–CAF) [4]. These proposals are very similar and differ only in details while the basic building blocks are elementary the same. They describe components and their roles in a distributed transaction processing system, and according communication protocols.

These communication protocols are based on the semantics of advanced transaction models (ATMs) [5]: ATMs express arbitrary transaction logics, and not only ACID transaction logic. Several advanced transaction models were presented that differ significantly from ACID style transactions. Mostly, ATMs try to relax ACID style transaction logic because its rigid demands are not appropriate for several application areas. Prominent examples for such ATMs are SAGAs [8] and multilevel transactions [17].

To describe transaction logics, formal models have been developed: Jim Gray and Andreas Reuter developed such a formal model in [9]. ACTA [3] is a very comprehensive model that captures transaction logics formally. ASSET [2], Bourgogne Transactions [15], and TWSO [10] are based on ACTA and enable arbitrary transaction logics in different environments: Imperative programming languages (ASSET), J2EE environments (Bourgogne Transactions), and Web service orchestrations (TWSO).

3. Fundamental Requirements for an SOA Transaction Processing System

3.1. Applicability and Reliability in Distributed Systems

SOAs are distributed systems. According to [6], a distributed system consists of a collection of autonomous computers, connected through a network and distribution middleware, which enables computers to coordinate their activities and to share the resources of the system, so that users perceive the system as a single, integrated computing facility.

Thus, a central requirement for an SOA transaction processing system is that it is designed to be usable in such distributed environments. Mainly, a dedicated component (*"transaction monitor"*, as described in Sect. 4), which offers transaction related services, manages and communicates transaction related matters, and is able to participate in the distributed system's communication, satisfies this demand.

Moreover, since such a central component is essential for many tasks in the distributed system, it has to be reliable to a high degree and virtually always working.

3.2. Advanced Transaction Logic

Common transaction systems found in relational databases or distributed systems platforms like J2EE or CORBA offer (mostly only) ACID transaction logic. ACID works well in tightly-coupled systems and is employed ubiquitously. However, transaction logic that follows ACID principles may not be practical in SOAs because of their inherent *loosely-coupled* and *coarse grained* nature. Potts et al. [14] even assert that "transaction semantics that work in a tightly-coupled single enterprise cannot be successfully used in loosely-coupled and/or multi-enterprise networks ...".

Two aspects of ACID transaction logics are problematic in SOAs. The *atomicity* requirement may not be adequate because it is likely that a service assembly overall succeeds even in case some of the services fail. The *isolation* requirement that prevents concurrency problems can be only realized by using rigid locks. That is, only a single transaction is allowed to access resources that are involved in this transaction as long as it is active. Other transactions have to wait until the transaction is finished. Therefore, the longer a typical transaction takes, the more the performance of the overall system decreases. Because of the coarse grained nature of services in an SOA, service calls tend to be time-

consuming and locking resources to achieve isolation may not be feasible.

Thus, common transaction systems are — generally speaking — inappropriate to handle coordination in SOAs. However, *advanced transaction models* are around for years and many ATMs that relax the rigid demands of ACID transactions have been proposed. In fact, proposals for Web service transaction processing (BTP, WS–TX family, and WS–CAF) make use of SAGAs and multilevel transactions. Such types of advanced transaction models are also appropriate to coordinate service assemblies in an SOA. Therefore, an SOA transaction processing system must support ATMs, i.e. it must enable the enactment of more kinds of transaction logic than just ACID logic.

3.3. Arbitrary Advanced Transaction Logic

In contrast to transaction processing in tightly-coupled systems, which is "omnipresent", transaction processing in loosely-coupled systems is rather uncommon, and transaction proposals for Web services (BTP, WS–TX family, and WS–CAF) had marginal influence on this situation.

A common shortcoming of these proposals is that they offer *de facto* only a fixed set of ATMs: Actually, the usage of arbitrary advanced transaction models is considered by the WS–TX family and WS–CAF. However, to do so it is necessary to introduce a new communication protocol and how this is done is omitted. Moreover, such an approach seems to be unwieldy because, most likely, a large amount of the components participating in the transaction system have to be updated in a non-standardized way in order to be aware of such new communication protocols. Huge efforts would have to be undertaken and would discourage users from using arbitrary transaction semantics.

According to [16], no out-of-the-box set of advanced transaction models can satisfy all requirements of all domains that want to do transactional (Web) service processing. According to this statement, service coordination using transaction processing concepts in SOAs would need the possibility to employ arbitrary transaction logic. We believe that this is reasonable and that arbitrary advanced transaction logic should be inherently supported by SOA transaction processing systems: As well as it is pointless to offer a fixed set of process logics in the assembly, it is ineffective to offer a fixed set of transaction logics.

4. Approach

To realize a transaction processing system that satisfies the requirements stated in Sect. 3, we use an approach based on ACTA [3] and inspired mainly by [10] as follows.

Basically, transaction logic is constructed from several (small-scale) transactions, where each transaction manages one ore more service–activities (see below for details on synchronization issues). In this paper, the term *service–activity* comprises the actions that happen when a service is invoked.

Such transactions are controlled by *transaction primitives*, which are special commands that are used for directing transactions. We use the following set of transaction primitives, which should be sufficient for SOA transaction processing systems. `begin` starts a transaction. In case the outcome of the associated service–activities of a transaction is considered to be successful, the transaction is terminated with a `commit`. Changes that happened in the committed transaction's scope are made permanent. When the outcome of the service–activities is considered to be erroneous, `abort` cancels a transaction. All changes made so far by the service–activities are revoked without any traces and the transaction is terminated. For ACID transaction logic, these three types of transaction primitives would be sufficient. However, advanced transaction logics for SOAs require an additional `compensate` transaction primitive that enables *compensation* of an operation: In case the changes of the service–activities of an already committed transaction should be undone, compensation is used to perform some forward actions that neutralize the already persisted changes. In contrast to an abort, compensation needs no locks on the corresponding resource. Thus, undoing the outcome of an operation will be generally feasible also in loosely-coupled systems by using compensation. However, after compensation, it is visible that the original service–activities took place. Moreover, it should be noted that there may be service–activities that cannot be compensated at all because their outcome is not compensatable by nature. In business terms, compensation resembles the "cancellation" concept.

Because a transaction manages service–activities, transaction primitives indirectly influence the logic of services. For example, let t_1 be a transaction that manages the service–activity a_1 of a service s. An abort on t_1 would stop the execution of s_{a_1} and all changes that were made in the course of s_{a_1} would have to be removed without any trace. It should be noted that this violates the autonomy principle of SOAs to some degree. However — as described below — the services decide how to react on transaction primitives and therefore keep autonomy to a high degree.

To *orchestrate* transactions efficiently, *transaction dependencies* are used. Such a transaction dependency consists of a *source state* and an *effect*. The source state is defined by a particular state of one or more transactions in the transaction processing system. The effect comprises the issuing of transaction primitives. The effect is executed when a particular state emerges. For example, a transaction dependency could state that when transaction t_1 has been committed and transaction t_2 has been aborted, transaction

t_3 should be compensated.

It should be stressed that — in contrast to the existing Web service transaction proposals — a transaction processing system that supports the concepts of transaction primitives and transaction dependencies as described above supports arbitrary transaction logic inherently. No system update is necessary to incorporate new transaction logic.

To realize an approach as described above in an SOA environment, we use an architecture as shown in Fig. 1.

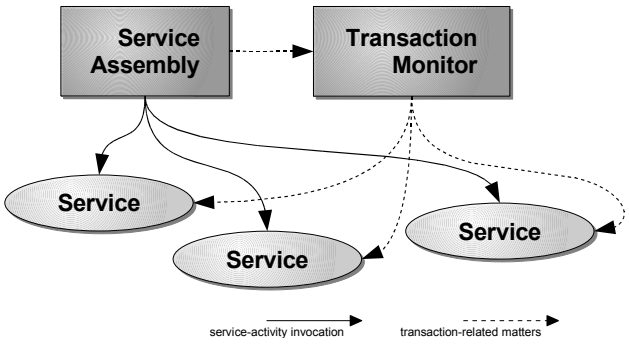

Figure 1. Architecture of an SOA transaction processing system.

The *service assembly* combines a group of services using some process logic. Thus, it invokes service–activities in well-defined sequences and/or concurrencies. To coordinate these services, the assembly employs transaction processing concepts as follows: It sets up transaction dependencies and controls transactions by issuing transaction primitives on transactions. By using these tools, an arbitrary transaction logic is defined.

All these *transaction-related matters* are carried out by the *transaction monitor*, a central highly-available component that manages all transaction-related matters in the SOA transaction processing system. First and foremost, the transaction monitor receives *transaction primitives* and forwards them — if necessary and valid — to affected services. When a transaction involves more than one service–activity, the transaction monitor executes *synchronization protocols* in order to assure a consistent outcome. We use the *two phase commit protocol* [9] to ensure a consistent atomic outcome. It should be noted that in our system coordinating a group of service–activities by associating them to a single transaction is only possible if an atomic outcome is desired. Otherwise, transaction dependencies and/or transaction primitives have to be used as described above to decide what a consistent outcome is. In addition, not only the `commit` primitive but also others, like `compensate`, may be synchronized using the two phase commit protocol for achieving an atomic outcome. Finally, the transaction monitor keeps track of *transaction dependencies* and executes dependency effects when a corresponding state emerges. For example, let dependency d_a state that when transaction t_1 is in state `aborted`, transactions t_2 and t_3 should be compensated. As soon as a commit on t_1 finishes, the transaction monitor will issue `compensate` primitives to t_2 and t_3.

The service assembly states transaction-related matters. According to these matters, the transaction monitor sends appropriate transaction primitives to affected services. These services have to fulfill certain requirements in order to participate reasonably in the transaction processing systems. In order to honor the *service contract* demand in Sect. 1 entirely, the service contract is augmented by transaction specific service capabilities. Thus, the whole system is aware of the transaction primitives and synchronization protocols a service can handle. Consequently, a service has to implement the transaction primitives and synchronization protocols that are stated in the service contract. By leaving the responsibility for the implementation of transaction-related matters to the services, the *autonomy* principle of SOAs is honored to a high degree.

Often, when a service is invocated, it must be aware whether a transaction is involved and — if so — which particular transaction is to be considered. For example, a service that supports `abort` and `commit` primitives will probably store temporary results and lock resources when a transaction is involved — measures that are not necessary when no transaction is involved. We use a *transaction context* to propagate transaction-related information in the distributed system. Every communication action in the SOA that is related to a transaction must be supplemented by this transaction context. Thus, components are aware if a transaction is involved and which particular transaction is to be considered.

5. Prototype

We have implemented the approach described in Sect. 4 using Java technology for SOAs based on Web services. SOAP handles communication in the entire system, including transaction-related matters like transaction primitives or synchronization protocol commands. The *transaction context* is propagated in the SOAP headers of the SOAP messages. We used Apache AXIS running in a Tomcat Servlet container for all participants in the distributed system to carry out the SOAP communication.

Service assemblies are expressed using Java. A library encapsulates transaction-related SOAP communication for the assembly. That is, setting up transaction dependencies and issuing transaction primitives is a matter of using API calls. The *transaction monitor* is implemented as a SOAP Web service. Common techniques for achieving

Web service high-availability may be used to keep the transaction monitor virtually always working. *Services* are implemented as SOAP Web services and receive transaction primitives and synchronization protocol commands through an extra SOAP interface. Transaction-related capabilities of services are expressed using WSDL and may be obtained through an API by concerned components like the assembly or the transaction monitor. This API uses WSDL4J to efficiently explore WSDL descriptions.

The implementation is prototypical and succeeds in enabling experiments with a working SOA transaction processing system. For example, we realized a scenario from the tourism domain as described in [10] in our implementation. However, in the current state, the implementation cannot fulfill all necessary requirements of a production quality SOA transaction processing system like high-availability, comprehensive input verification, and error handling. Nevertheless, since code quality and extensibility is high, the implementation constitutes a base for a production grade transaction processing system for Web service SOAs.

6. Conclusion

In this paper we have described requirements for a transaction processing system in service oriented architectures and have presented an approach that satisfies these requirements. In contrast to prominent proposals for Web service transactions like BTP, WS–TX family, and WS–CAF, which may be used in SOAs based on Web services, the presented approach inherently supports the use of arbitrary advanced transaction logic. With this approach, service assemblies in SOAs can be coordinated efficiently using transaction processing principles, not least because arbitrary transaction logic can be employed.

We implemented the approach prototypically for Web service based SOAs. This working implementation can be used for experimenting with transaction processing in SOAs. Moreover, it can serve as a base for a production-grade SOA transaction processing system.

Future work will focus mainly on the improvement of the prototype to achieve production quality in order to employ it in real life SOAs. Furthermore, it is likely that the experiments and the use of the approach in real life SOAs will provide new insights on the topic. These insights will be considered in the approach as well as in the implementation.

References

[1] Alex Ceponkus et al. Business Transaction Protocol. `http://www.oasis-open.org/committees/download.php/1184/2002-06-03.BTP_cttee_spec_1.0.pdf`, 2002. cited on 2007-05-14.

[2] Alexandros Biliris et al. ASSET: A System for Supporting Extended Transactions. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 44–54, Minneapolis, Minnesota, 1994.

[3] P. K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[4] Doug Bunting et al. Web Services Specifications to Coordinate Business Applications. `http://developers.sun.com/techtopics/webservices/wscaf/primer.pdf`, 2003. cited on 2007-02-26.

[5] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications*. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[6] W. Emmerich. *Engineering Distributed Objects*. Wiley & Sons, 2000.

[7] T. Erl. *Service-Oriented Architecture: Concepts, Technolody, and Design*. Prentice Hall International, 2005.

[8] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 249–259, New York, USA, 1987. ACM Press.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, California, 9th edition, 2002.

[10] P. Hrastnik and W. Winiwarter. TWSO - Transactional Web Service Orchestrations. In *Proceedings of the 2005 International Conference on Next Generation Web Services Practices*, pages 45–50, Los Alamitos, California, 2005. IEEE Computer Society.

[11] Luis Felipe Carbrera et al. Web Services Coordination, Web Services Business Activity Framework, Web Services Atomic Transaction. `http://www.ibm.com/developerworks/library/specification/ws-tx/`, 2005. cited on 2007-05-14.

[12] Matthew C. MacKenzie et al. Reference Model for Service Oriented Architecture 1.0. `http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf`, 2006. cited on 2007-05-14.

[13] E. Newcomer and G. Lomow. *Understanding SOA with Web Services*. Independent Technology Guides, 2005.

[14] M. Potts, B. Cox, and B. Pope. Business Transaction Protocol Primer. `http://www.oasis-open.org/committees/business-transactions/documents/primer/`, 2002. cited on 2005-01-28.

[15] M. Prochazka. *Advanced Transactions in Component-Based Software Architectures*. PhD thesis, Charles University, Faculty of Mathematics and Physics, Department of Software Engineering, Malostranske namest i 25, 118 00 Prague 1, Czech Republic, 2002.

[16] J. Roberts and K. Srinivasan. Tentative Hold Protocol Part 1: White Paper. `http://www.w3.org/TR/2001/NOTE-tenthold-1-20011128/`, 2001. cited on 2007-05-14.

[17] G. Weikum and H. J. Schek. Multi–Level Transactions and Open Nested Transactions. In *Data Engineering*, volume 14, pages 60–64, Los Alamitos, California, March 1991. IEEE Computer Society.