OpenWS-Transaction: Enabling Reliable Web Service Transactions

Ivan Vasquez, John Miller, Kunal Verma, and Amit Sheth

Large Scale Distributed Information Systems, Department of Computer Science,
The University of Georgia,
415 Graduate Studies Research Center, Athens, GA 30602-7404 USA
{vasquez, jam, verma, sheth}@cs.uga.edu
http://lsdis.cs.uga.edu

Abstract. OpenWS-Transaction is an open source middleware that enables Web services to participate in a distributed transaction as prescribed by the WS-Coordination and WS-Transaction set of specifications. Central to the framework are the Coordinator and Participant entities, which can be integrated into existing services by introducing minimal changes to application code. OpenWS-Transaction allows transaction members to recover their original state in case of operational failure by leveraging techniques in logical logging and recovery at the application level. Depending on transaction style, system recovery may involve restoring key application variables and replaying uncommitted database activity. Transactions are assumed to be defined in the context of a BPEL process, although other orchestration alternatives can be used.

1 Introduction

OpenWS-Transaction is a middleware framework based on WS-Coordination (WS-C) and WS-Transaction (WS-T) that enables existing services to meet the reliability requirements necessary to take part in a coordinated transaction. For transactions following WS-AtomicTransaction (WS-AT), it features an innovative recovery facility that applies logical logging to restore operations on the underlying data, extending system recovery to include uncommitted database activity. For transactions following WS-BusinessActivity (WS-BA), it presents a straightforward scheme to automate the invocation of user-defined compensating actions. In contrast to existing implementations, OpenWS-Transaction aims to minimize the implementation impact in existing applications with regards to both performance and code changes.

The framework has been implemented as part of the METEOR-S project, which deals with adding semantics to the complete lifecycle of Web services and processes [1]. As a prototype implementation of transactional Web processes, it is particularly focused on integrating BPEL, WS-C and WS-AT/WS-BA [2, 3], which already enjoy wide acceptance.

The next section explains the framework's architecture. Section 3 describes an example scenario where OpenWS-Transaction enables reliable transactional business processes. Section 4 provides implementation and evaluation details, while section 5 summarizes this demonstration.

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 490 – 494, 2005.

2 Architecture

OpenWS-Transaction applies concepts from the reference specifications as well as from existing work on fault tolerant systems [4, 5]. Fig. 1 illustrates the interaction between a BPEL process, the Coordinator, and other services that benefit from the Participant framework entity. Any activities performed within the transactional scope are guaranteed to complete consistently.

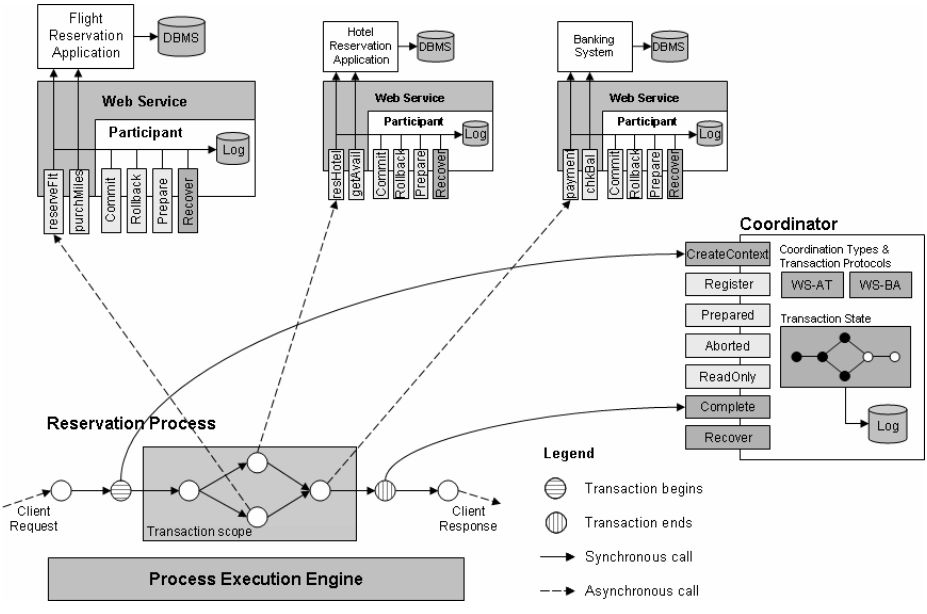

Fig. 1. Entities and their interaction in a transactional business process

Coordinators are dedicated services responsible for delineating new transactions, activating participant services and enforcing transactional behavior according to some *coordination type*. To support recovery, they also record key events throughout the transaction's lifespan using the logging schema shown in Fig. 2. Besides the operations prescribed by WS-C, WS-AT and WS-BA, the *recover* operation restores the state of pending transactions when interrupted by an operational failure.

Many services are the result of evolved applications that have defined an additional layer exposing select functionality to business partners. To take part in a distributed transaction, conventional services can use the features provided by the *Participant* framework entity. Among such features is the ability to intercept and record operation details, guaranteeing a *precommit* behavior regardless of the underlying database system. Using the schema in Fig. 3, their *recover* operation enables transaction participants to go back to the state immediately previous to a failure.

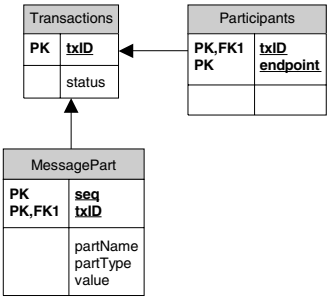

Fig. 2. Coordinator log schema

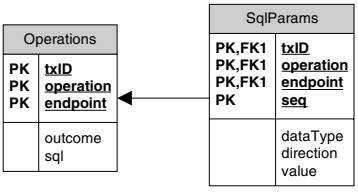

Fig. 3. Participant log schema

3 Example of a Transactional Process

We use a variation of the well-known travel agency use case. The process encompasses three services: A flight reservation system, a hotel reservation system and a banking system. The process is triggered from a Web application in which the user is given options for an immediate purchase (WS-AT) or a long-running process (WS-BA) that increases the chance of finding a suitable itinerary.

In the process definition, service invocations are enclosed by *beginTransaction* and *endTransaction* calls to the coordinator, which delimit the transaction's scope. Before performing any work, participants *register* with the coordinator by providing their endpoint address, which is logged to stable storage to support system recovery.

As soon as participants fulfill their part of the process, the framework logs the operation's name and outcome. For WS-AT, it also logs associated database calls and their parameters, critical to restore uncommitted activity in case of failure. Once operations are recorded, participants report their outcome to the coordinator.

Process execution continues until the *endTransaction* operation is invoked. This causes the coordinator to decide the transaction's final outcome, which depends on participant votes and current coordination type: For WS-AT, all steps of the process must succeed. For WS-BA, we assume that just reserving the flight and processing its payment is enough to consider it successful; however, because of its nature, services must supply an appropriate compensating operation for every business operation.

Following outcome determination, the coordinator updates its transaction log record and confirms or cancels each operation. Participants then forget about the transaction and the process engine communicates its outcome to the client application.

Responding to Operational Failures. Next, we modify the above scenario by introducing an operational failure (Fig. 4) after the transaction outcome has been determined. Assuming a positive outcome and WS-AT coordination type, participants are responsible to commit despite failures. However, these failures cause volatile state information to vanish and, because applications are unaware of the global process, local transactions are implicitly rolled back.

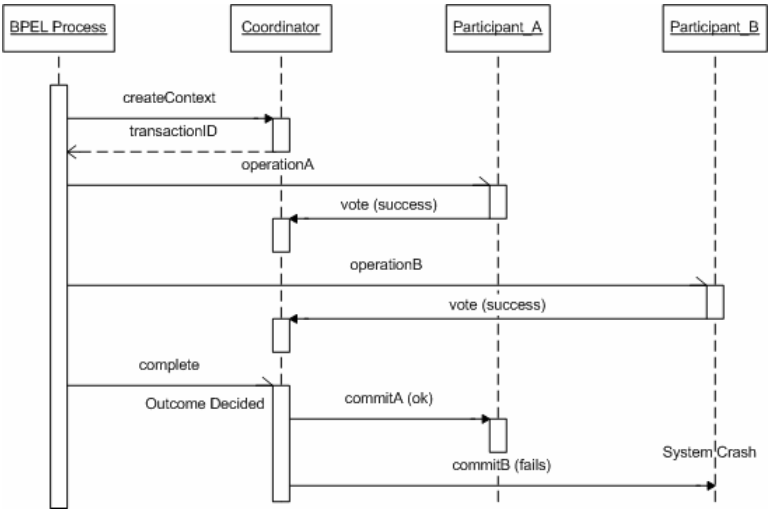

Fig. 4. A failed transactional process where one of its participants crashed

If that is the case, OpenWS-Transaction's coordinator attempts to contact the failed service for a configurable number of times and retry interval. Assuming it becomes available on time, the coordinator first invokes the participant's *recover* operation, which restores key application variables such as transaction identifier, coordination type and operation outcomes. Additionally, recovery also restores the participant's database connection and replays database activity for uncommitted operations (Fig. 5), leaving it ready to accept the final decision.

⚠	edu.uga.cs.lsdis.meteors.wstx.samples.TravelCoordinator	Beginning crash recovery of 2 transactions
	edu.uga.cs.lsdis.meteors.wstx.samples.TravelCoordinator	Restored session for transactionID 192.168.0.2:1112677459590
	edu.uga.cs.lsdis.meteors.wstx.samples.TravelCoordinator	Attempting recovery of http://192.168.0.2:88/axis/services/FlightService in 10 secs.
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	Registration succeeded, context created for http://schemas.xmlsoap.org/ws/2004/10/wsat
⚠	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	********* Business logic starts *********
	edu.uga.cs.lsdis.meteors.wstx.util.DataAccess	Using data source 'jdbc/pgsql'
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	Restored database connection 'jdbc/pgsql'
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	Replaying procedure '{? = call reserve_flight(?, ?, ?)}'
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	stmt.registerOutParameter(1, Types.INTEGER)
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	stmt.setString(2, 'ATL')
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	stmt.setString(3, 'FRA')
	edu.uga.cs.lsdis.meteors.wstx.samples.wsat.FlightService	stmt.setInt(4, 3)
	edu.uga.cs.lsdis.meteors.wstx.samples.TravelCoordinator	Recovery of http://192.168.0.2:88/axis/services/FlightService succeeded

Fig. 5. Participant replaying a database procedure as part of system recovery

Yet another recovery scenario is one in which the coordinator itself goes down in the middle of a process, leaving pending operations at multiple participants. Upon restart, the coordinator scans its log records forward in time, looking for unfinished transactions. State is then restored by polling registered participants on their *prepare* operation. If a participant is not available or does not seem to know about the transaction, it is asked to recover beforehand.

The framework takes into account the effects of network failures. Before performing recovery, participants check whether it is really needed by verifying the local

coordination context. An additional check is done by validating participant registration at the coordinator, so recovery can not occur as the result of erroneous or malicious requests.

4 Implementation and Evaluation

The framework was implemented in Java and relies exclusively on open source projects. Web services run on Apache Axis and Tomcat. Transaction logging is based on BerkeleyDB, an embedded database system. Sample processes are deployed in ActiveBPEL. Web services access data on PostgreSQL and MySQL; other JDBC-accesible sources like Oracle and SQL Server have also been tested successfully.

Evaluating the impact on existing services, we found that the framework can be integrated into existing services by introducing changes to as few as a couple lines of code. Because protocol operations are invariably the same, developers of new applications can remain focused on their business logic.

Experimentation has shown that, even without logging optimizations, the additional overhead results in an average 7.5% increase over the operations' original execution times.

5 Conclusion

OpenWS-Transaction is a framework that facilitates the implementation of Web service-based processes requiring transactional behavior. Example scenarios demonstrate its transactional support under normal and operational failure conditions, achieved by providing the necessary protocol operations and by restoring the state of failed services.

References

1. Sivashanmugam, K., Verma, K., Sheth, A., Miller, J.: Adding Semantics to Web Services Standards. Proceedings of the 1st International Conference on Web Services (2003)
2. Tai, S., Khalaf, R., and Mikalsen, T.: Composition of Coordinated Web Services. Proceedings of the 5th ACM/IFIP/USENIX intl. conf. on Middleware (2004)
3. Papazoglou, M.: Web Services and Business Transactions. World Wide Web: Internet and Web Information Systems, Tilburg University (2003)
4. Lomet, D. and Tuttle, M.: Logical Logging to Extend Recovery to New Domains. Proc. of the 1999 ACM SIGMOD intl. conf. on Management of Data (1999)
5. Salzberg, B. and Tombroff, D.: Durable Scripts Containing Database Transactions. IEEE International Conference on Data Engineering (1996)