Composite Transactions for SOA

A Technical Whitepaper by: Guy Pardon - Chief Technology Officer for Atomikos | March 2019

Contents

Executive Summary

Strategic SOA-based business services can be a key element in business transformation, allowing businesses to flexibly respond and react quickly to changing business conditions. As the enterprise matures in its use of SOA, and services-based implementations move from prototype technologies to mission critical applications, issues around security, transactions, and reliability begin to dominate the thinking of CTOs, CIOs, and software architects.

In this whitepaper, Atomikos focuses on the topic of transactions for composite applications in a SOA, specifically examining the concept of longer-running transactions and the implications of these transactions for service providers. Specifically, this paper will examine the ways in which transactions and compensation fit into the business model of a service provider, and discusses the architectural support required to ensure these transactions occur successfully and with reliability.

Transactions Defined

A transaction is defined as a task that is composed of one or more operations or related interactions that may need to be cancelled or "rolled back" after the point of execution. A long running transaction is a transaction where the total duration exceeds the duration of an individual interaction by many orders of magnitude. In this situation, the cancellation of an individual interaction within a long transaction may occur long after the interaction has been executed.

Let's use an example to illustrate a long running transaction.

A business transaction consists of scheduling a trip from Brussels to Toronto, booking an initial flight from Brussels to Washington, followed by a connecting flight from Washington to Toronto, the final destination. In this example the second flight from Washington to Toronto is booked with a different airline, and through a different reservation system.

However, after this first transaction is completed, the turns out that the flight for the second leg has been cancelled due to weather. The traveler in question is now stuck with a ticket from Brussels to Washington, and rather than seek alternative connecting flights from Washington to Toronto later that day, he decides to cancel his entire trip.

The reservation has already been made. The airline reservation system has accepted and verified the first reservation, and is holding a seat for the customer.

If there is no cancel event, the business traveler is either booked with a ticket he no longer needs and cannot use, or the airline is going to lose money on an unneeded reservation. Either way somebody loses money – most likely the customer or the reservation agent. Consequently, there are strong drivers for the existence of a cancel event. service providers. Specifically, this paper will examine the ways in which transactions and compensation fit into the business model of a service provider, and discusses the architectural support required to ensure these transactions occur successfully and with reliability.

Part of the Challenge: ACID Based Transactions

Looking at the technology behind this scenario, the concept of cancelling different interactions across different geographic locations is not a new one. This capability has been available for decades by a transaction manager or transaction service. CORBA's Object Transaction Service (OTS), J2EE's Java Transaction API and Java Transaction Service (JTA/JTS) are various examples of these existing technologies.

These transaction management and transaction service technologies are all based on the concept of ACID transactions, which are based on the following properties: atomicity (all changes performed as part of the transaction happen, or none of them happen); consistency (data is transformed from one correct state to another); isolation (multiple concurrent updaters are prevented from interfering with one another); and durability (committed changes remain permanent – even in the face of failure). In the case of ACID transactions, all transactions become locked from other concurrent transactions until the transaction either commits and saves changes or rolls back and cancels the changes.

However, for long running, services based transactions, the ACID approach is far from ideal. Cancellation is only possible if all data remains locked, and maintaining locks on services transactions can cause difficulties for organizations such as:

> Delays due to the wide-area nature of this (Internet-based) system may result in long lock times and long periods of data unavailability

> Denial of service attacks at the database level because parties do not know or trust each other

A Better Way – Compensation-Based Transactions

So if an ACID approach is not ideal for long running services based transactions, what is the alternative? The answer: take a compensation-based approach.

Compensation is a separate ACID transaction local to a service provider that logically cancels the effects of a previous ACID interaction. Rather than implement a long-running transaction as one giant distributed ACID transaction, a compensation-based approach treats each service invocation as one short local ACID transaction, committed as soon as it has executed.

This approach drastically reduces lock duration, and reduces the risk of denial of service attacks - but at the expense of rollback ability. Therefore the cancellation has to be performed differently; by executing a separate local ACID transaction that logically cancels the work after it was committed.

Let's revisit our airline reservation example to see how a compensation-based approach might work.

Booking a reservation for the flight from Brussels to Washington would be an ACID transaction in its own right. It is a transaction that is local to the airline system and it is committed immediately. When there is a need to cancel, a second ACID transaction would logically undo the effect of the reservation. The resulting lock times are much shorter. The method used to cancel the transaction depends partly on the application itself, and also on the application programmers involved. The reservation record could be deleted, leaving no trace of its existence at all. Or it could be explicitly marked as "cancelled." Either approach is fine. The path taken really depends on the business model of the services provider.

In general, there are two distinguishable types of compensation:

1. Perfect compensation – the reverse logic cleans up the effects of the original transaction. This is the scenario where the transaction record is deleted.

2. Imperfect compensation – The reverse logic leaves detectable traces of the original. This is the scenario where the transaction is marked "cancelled.""

Aligning a Compensation-Based Transaction Approach to the Business

So how do compensation-based transactions fit into today's business environment?
The answer varies depending on the business model at hand.

A Simple Compensation-Based System

Let's first look at a very simple compensation-based system, where the service provider is stateless. In this case, the compensating task is nothing special – it could just as well be a regular business interaction. The service- has no requirement to offer an additional service implementation to do the compensation and the process remains a very simple one. Let's look at a simple compensation-based system at a stock brokerage. An investor buys stock, and can compensate by selling it later. Investing-savvy readers will know that this model of compensation is imperfect. The chances are very poor that the stock price will be the same at the point of purchase and at the time of sale. The investor who requests the compensation either gains, or loses money. But the compensation transaction is simple. The services provider does not have to worry about compensating logic.

The Try-Cancel/Confirm (TCC) Approach

The Try-Cancel/Confirm (TCC) approach is an ideal solution for business models that are two-phased, such as the airline reservation example we have used in this paper. Using this TCC approach, the compensation is an explicit second stage of the same business transaction involving custom logic and coding as well as the need for the business transaction state to be maintained over a period of time. TRY represents the normal business logic, such as the reservation of a seat on an airline by a customer. CANCEL represents the possible cancellation of that reservation using a compensation mechanism. If there is no need to cancel, then CONFIRM, as an ACID transaction, completes the transaction and updates the reservation state in the database.

But why add the CONFIRM? To investigate the rationale behind the addition of CONFIRM we need to look at the states of a reservation during its lifetime/lifecycle. When a reservation is first made, its business state is PENDING. This indicates to the airline that the reservation is not final and can still be cancelled at the customer's request. The PENDING status of a reservation is important information for the airline's booking services and financial reporting. This record should not be officially logged until it is final. Some airlines might prefer to build in a timeout function related to the reservation that will automatically CANCEL the reservation. However, as soon as the state of the reservation moves to CONFIRM, the web service knows that the reservation is permanent, the seat booked, and the customer should be billed.

It should be noted that a service provider must expose additional functionality to enable CANCEL and CONFIRM states for transactions. Atomikos® provides this capability to service providers through a patent pending Try-Cancel/Confirm approach within its transaction management system, ExtremeTransactions® that sets it uniquely apart from other competitive solutions on the market today.

Architectural Support for Compensation-Based Systems

Simple Compensation

If compensation can be considered a part of business as usual, as shown in the simple system stock broker example, there is no risk for the stock broker. A purchase of stock is compensated by a later sale, and the stock broker will earn a commission on the purchase, or the sale, or on both activities. This is the most flexible case for compensation with the least associated risk.

Architectural support for this kind of compensation system can be achieved through the use of standard business workflow technology such as the Business Process Execution Language standard (BPEL). A remote workflow engine is used to model and execute the task, and the purchase of stock is just merely one step in the overall process. Should the process (or long-running transaction) need to be canceled, the workflow engine sends a business request to sell the stock it previously acquired. The stock broker does not concern himself with the length of time it takes for the compensation to occur. This is just another separate business transaction and there is no need for a broker-specific state that links the compensation to the original stock purchase.

Try-Confirm/Cancel

Try-confirm/cancel (TCC) processes are more complex and so is the architectural support they require. For instance, there is little chance that the business owner of a TCC service will ever allow a remote workflow to take full responsibility of its compensation, as there is an implicit reservation of resources for the duration of the TCC process. The state management for a TCC transaction is something that naturally belongs with the service provider.

Returning to our flight reservation example, a seat is booked as long as the reservation is still PENDING. During this time, the seat is not available to other potential customers. In this situation it matters very much how long the transaction process takes, and it is unlikely that the airline will allow a remote workflow engine to keep the seat pending for the full length of time. Rather, there will be a scheduled timeout after which the reservation is canceled automatically by the reservation service. A logical consequence is that the reservation service at least needs to know how to cancel, and the cancellation logic is co-located with the service. The cancellation of the reservation would likely be triggered via an event that carries the transaction's token (ID), since the logic is already on the service anyway. The same approach holds for the CONFIRM logic.

Mirroring the business model, architecture for such a system also follows a two-phased protocol. TRY is the first phase, whereas the second phase consist of either a CANCEL request or a CONFIRM request enabled by a two-phased transaction manager for TCC systems, such as Atomikos ExtremeTransactions® to automate the state management of TCC transactions.

A workflow engine could integrate with a two-phased transaction manager to automate the process of sending CANCEL or CONFIRM requests to all relevant sites. For instance, in a BPEL engine, a transaction management plug-in could detect all remote calls and recognize all participants involved in a distributed task. When a cancellation is requested, the transaction management system contacts each participating web

service and asks it to CANCEL its respective part of the work. Keeping in mind that every workflow step may have its own compensation, the use of a transaction management system to coordinate this cancellation effort would reduce workflow-modeling complexity by about 50 per cent. Instead of explicitly modeling each compensation as a step in the workflow logic, the transaction management system maintains an inventory of the sites needed to compensate and automatically handles cancellation. This eliminates the need to build explicit compensation steps into the workflow logic, greatly simplifying the process, as well as reducing associated complexity, time and costs.

In Summary

As business embraces SOA more actively, and complex and long running service-enabled transactions become more prevalent in the business environment, organizations will need to seek out new and innovative ways to manage these transactions to minimize complexity, and maintain the flow of business.

This paper showcases two forms of compensation-based service models, the stockbroker compensation-based transaction that operates stateless, and the airline reservation example that leverages the Try-Confirm/Cancel stateful approach. It also discusses the necessary underlying architecture required for each model. If your business processes are rising in complexity, are long running and are two-phased, you are encouraged to further examine adopting a TCC approach enabled through a transaction management system such as ExtremeTransactions®. The resulting benefit will be more smoothly executed transactions, and greatly reduced workflow complexity.

If you are a service provider developing services-based transaction processes, or an organization engaged in extreme transaction processing, you may benefit from Atomikos' patent-pending Try-Confirm/Cancel approach, and its commercial transaction management system, ExtremeTransactions®.

Next Steps / Update

The ideas presented here are find for really loosely-coupled services across ownership domains. However, today 2019 people are at the stage of doing microservices within their company. Our community told us that they don't want to code compensation logic for that, so instead we built this:
https://www.atomikos.com/Blog/TransactionalRESTMicroservicesWithAtomikos

About Atomikos

Atomikos is a market leader in transaction management for XTP, SOA and open source environments. Atomikos' software safeguards your critical transactions and prevents costly data loss in the event of a system failure or crash by automating the cancellation of failed business transactions. For more information visit our website at www.atomikos.com or contact us at sales@atomikos.com.

Contact Us
Sales contact: sales@atomikos.com
General contact: info@atomikos.com
Phone: +32(0)15613055
Atomikos BVBA Hoveniersstraat 39/1,
2800 Mechelen, Belgium

References

> Composite Systems: Distributed Nested Transactions. PhD Thesis, Guy Pardon, December 2000.

> The OTS specification: http://www.omg.org/technology/documents/formal/transaction_service.htm specifies the interfaces towards a CORBA transaction manager.

> BPEL: Business Process Execution Language is a formal language for expressing workflows over web services, enabling subsequent interpretation and execution by a workflow engine. For more information, check out http://www.oasis-open.org.

> JTA: The Java Transaction API specifies the interfaces towards a transaction manager from within a Java program. More information on http://java.sun.com/products/jta (specifications only). For a working JTA product, check out http://www.atomikos.com