

# The SOA Magazine

## Feature Article



### Service Transaction Handling Without WS-AtomicTransaction

by Kanu Tripathi

Published: February 23, 2009 (SOA Magazine Issue XXVI: February 2009)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmarks](#)

*Abstract: This article explores options available to service-oriented architecture implementations to address distributed service transaction requirements when support for specifications like WS-Coordination [REF-1] and WS-AtomicTransaction [REF-2] are not available. An example is discussed to explain this problem state and four separate alternative solutions.*

#### Introduction

As more and more organizations are realizing the benefits of creating an ever growing inventory of services, many of these services may have to be used in situations where the functionality they provide can be included in single, short-lived atomic transactions.

Although there are ratified industry standards that address this requirement (e.g. WS-Coordination and WS-AtomicTransaction), your platform may not support them. As a result, you may need to explore alternative mechanisms capable of simulating transaction control.

In this article defines we explore such mechanisms along with their pros and cons. Each solution discusses impacts on service design principles like Service Reusability, Service Autonomy, Service Abstraction, and Service Composability [REF-3]. Even though the examples focus on services implemented as Web services, the solutions covered here can be applied to other types of service implementation technologies.

#### An example

Assume that we have two simple services and operations as follows:

Service 1: Customer Management Service:

- CreateCustomer: Creates a new customer
- UpdateCustomer: Update an existing customer
- GetCustomer: Returns customer information

Service 2: Order Management Service:

- CreateOrder: Creates a new order
- UpdateOrder: Updates an order
- GetOrder: Returns order information

These are typical entity services that provide basic data access capabilities for customer and order entities. Since both entities are logically separate, it makes sense to represent them with two services. Because these two services are managing different entities, each can be made to be autonomous, reusable and decoupled.

Over a period of time, a new requirement comes along: We should now be able to audit all changes to both customer and order data. For example, if a customer's address changes, we need to record that in the audit log. Similarly, if there is a change in an order shipment address, we need to record this also.

Because there are many possible changes, we decide to create a separate Auditing Service, as follows:

Service 3: Auditing Service:

- CreateAuditRecord: Creates a new audit record
- GetAuditRecord: Returns audit records

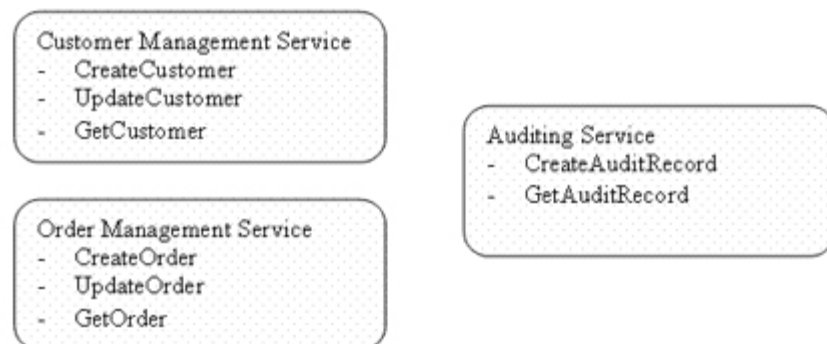


Figure 1: Three services are identified.

As shown in Figure 1, we have three services with clear responsibilities, but how can we wrap the data access functions and audit record creation within a single atomic transaction?

### The Ideal Solution

Ideally, we would use WS-Coordination and WS-AtomicTransaction, which define technologies that address cross-service transactions by guaranteeing that they all succeed or rollback changes if they any one participant fails.

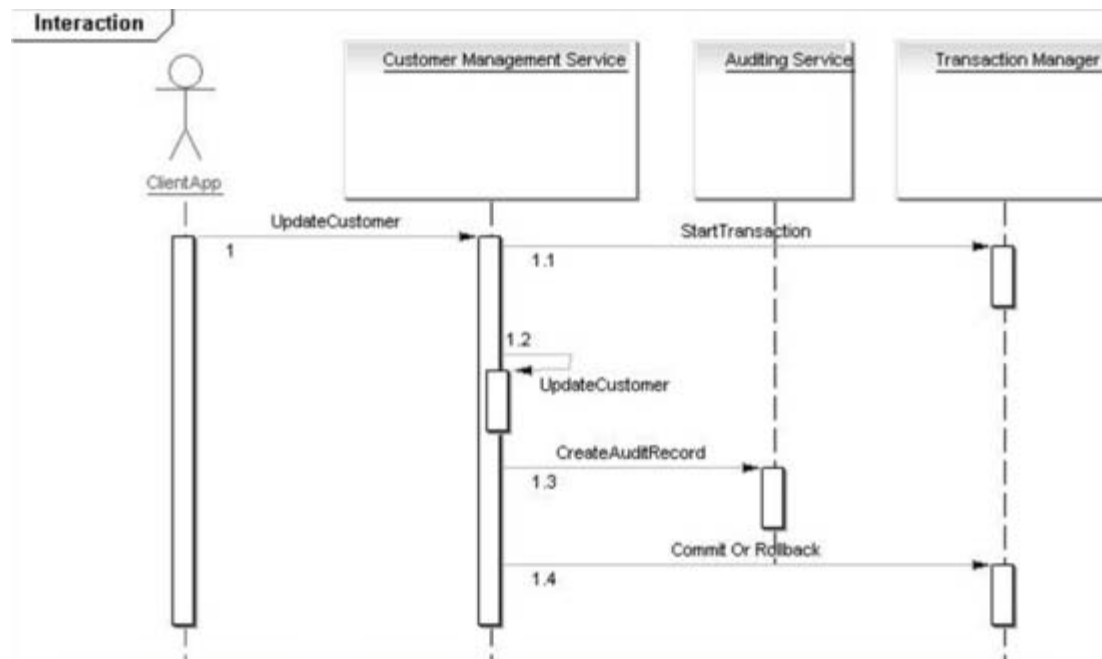


Figure 2: Distributed transaction using WS-AtomicTransaction.

As shown in Figure 2, when customer information is updated, a new transaction is started and then the update happens along with the audit record creation. If both operations succeed, then the transaction is committed; otherwise it fails. This solution is also described in Atomic Service Transaction [REF-4] design pattern, which identifies the problem of transactional context propagation when invoking multiple service operations. This solution requires that a transaction management system is made a part of service inventory architecture and is used by composed services that require rollback features.

Let's now look at the alternatives.

### Alternative Solution 1: Merge Auditing Functionality into Each of the Other Services

This solution simply suggests that instead of creating a separate Auditing Service, you can simply keep auditing logic within the entity services (Figure 3).

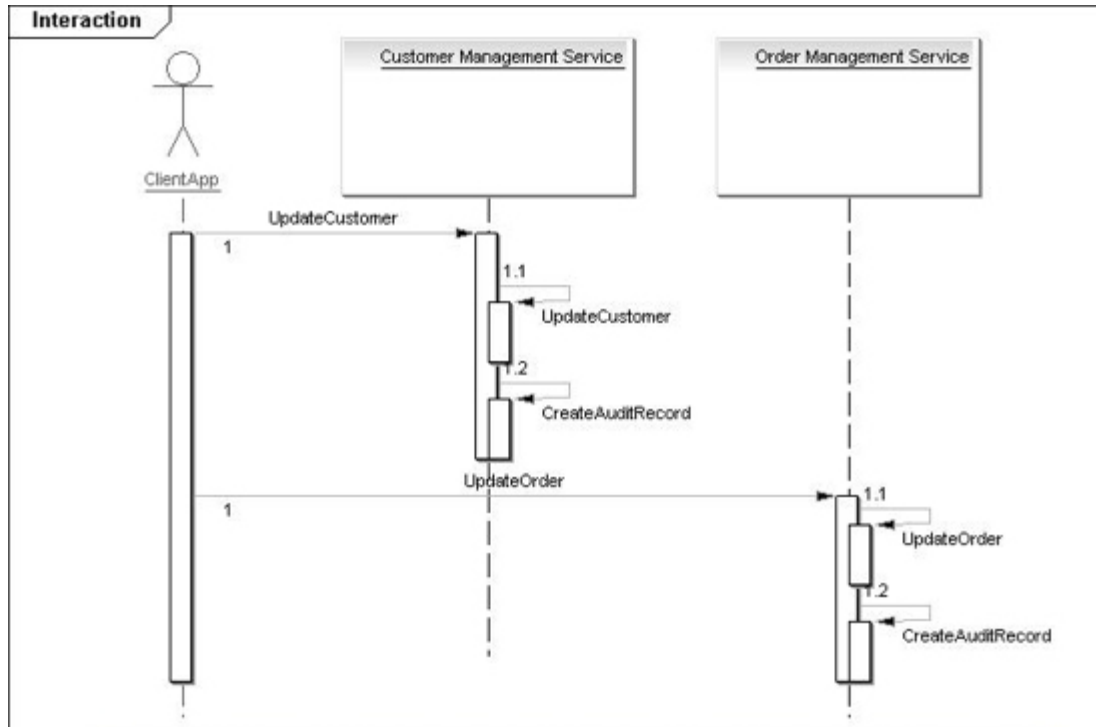


Figure 3:  
Alternative  
Solution #1.

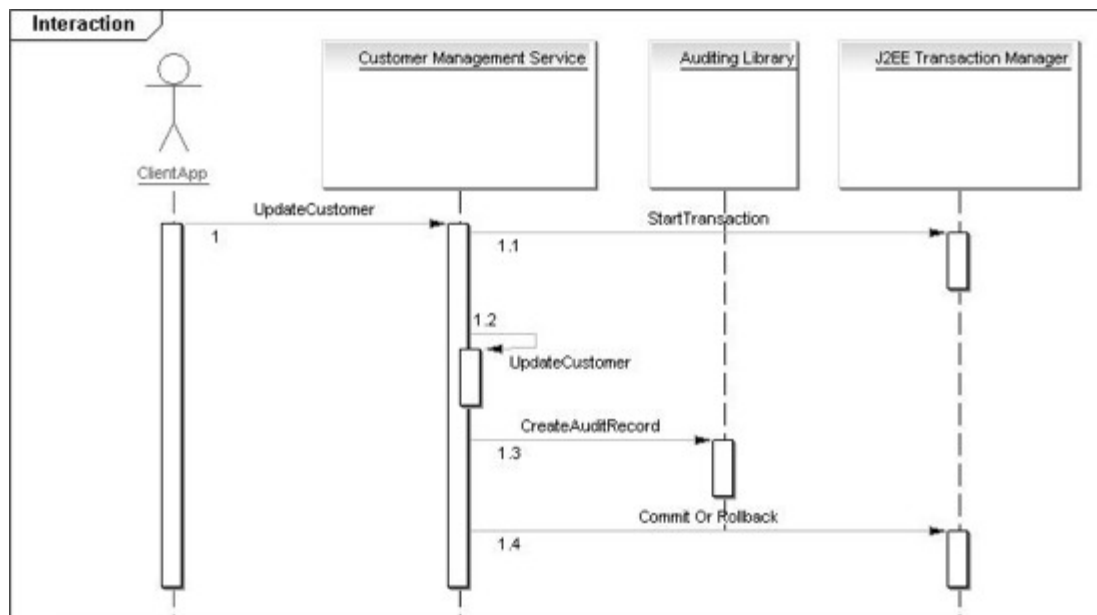
Although the simplicity of this solution may be attractive, it does lead to more work since most likely these services have their own database and thus the entire implementation of auditing will have to be duplicated.

Also, if more services need auditing functionality as well, they will all have to implement their own auditing capabilities, leading to duplication and redundancy and further violating the Service Normalization pattern [REF-4]. Further, we lose centralized control of auditing functions, which can negatively impact long-term governance. It is also worth noting that this solution goes against the Logic Centralization pattern [REF-4] as well, which recommends limiting the options of accessing logic to one.

### Solution 2: Create a Shared Auditing Library for use by Each Service

This solution (Figure 4) proposes creating a shared library (e.g. a JAR file or a DLL) that each service uses. This effectively establishes a reusable component that must be written in such a way that it can participate in pre-existing transactions. For example, it may need to be invoked from a Spring container so that it can execute its functions within transactional boundaries. In this case, the transaction support has to be provided by the underlying platform, but it does not need to be based on WS-AtomicTransaction.

Figure 4: Solution  
#2



A proper implementation of this approach would require appropriate development life-cycle management of the shared library. That means there must be suitable version control management allowing multiple versions to remain available. This will enable each service to independently choose which version of the library it needs to use. Thus the library can grow at its own pace without affecting services that use it and also the services can evolve on their own without becoming dependent on the lifecycle of the library.

Being a shared library with its own lifecycle, the impact on autonomy of the services is reduced. Although services become dependent on the library which is controlled externally, they retain the ability to select the library version. By letting the library be managed as an independent enterprise resource and by allowing each service to decide which version to use and when to upgrade, the impact on service autonomy is minimized.

There can, however, be some practical problems with this solution. For example, if the library uses its own database, then we need to make sure that the underlying platform supports distributed transactions so that different databases can be updated as a part of a single transaction.

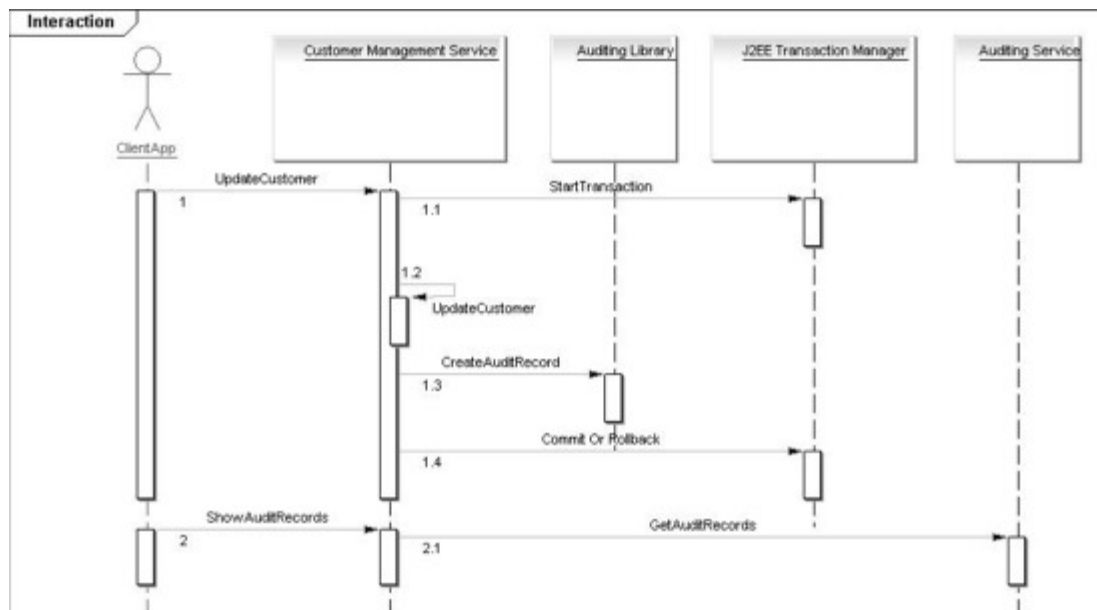
Secondly, if the database schema for the library has to be changed, it may not be a backwards compatible change. In this case, all services using the library may have to be upgraded to a new version of the library. Another problem is related to heterogeneous platforms used to host services; for example, if the Order service runs on Java and the Customer service runs on .Net, then we will have to create multiple implementations of the library to support different platforms.

### Solution 3: Create a Shared Library and Keep the Auditing Service

This solution (Figure 5) recommends that a shared library is created together with the Auditing Service. The rationale behind this approach is that although the creation of the audit log may need to be part of a transaction, there are other capabilities (like read audit log) that do not need transactional invocation. So it is beneficial to keep these non-transactional capabilities as part of a service and only provide transactional operations within a shared library.

Furthermore, the service must also provide a capability to create the audit log, even if it cannot participate in a transaction. This is again for situations where transactional propagation may not be necessary. Whenever possible the service would be used by default and the library would be used in those situations where service is not suitable.

Figure 5: Solution #3



This solution should be implemented in such a way that the service and the library both uses the same code base. It may be best to create the library and use it as part of the service logic implementation. To follow this approach, care must be taken to ensure that newer versions are rolled out for the library and service together. To maximize backwards compatibility, previous versions of the service may need to be kept alive for a reasonable amount of time and proper service versioning strategies will need to be defined and followed [REF-5].

The primary benefit of this approach is that the library creation is minimized and the push towards the service is not adversely impacted. This can lead to better governance of services and components. Also, since we are providing only a limited set of functions in the library, it may not require much effort to provide implementations for other platforms (i.e. a JAR file for Java-based clients and a DLL for Windows based clients).

The challenges associated with this approach are not trivial by any means. You may have to provide multiple versions of the library as well as the service as time progresses and more capabilities are added. The versions of the service and the library have to be kept in synch and so as a matter of practice, whenever a modifications occur, you may need to release a new version of both and publish proper release notes. This can become an added release management burden.

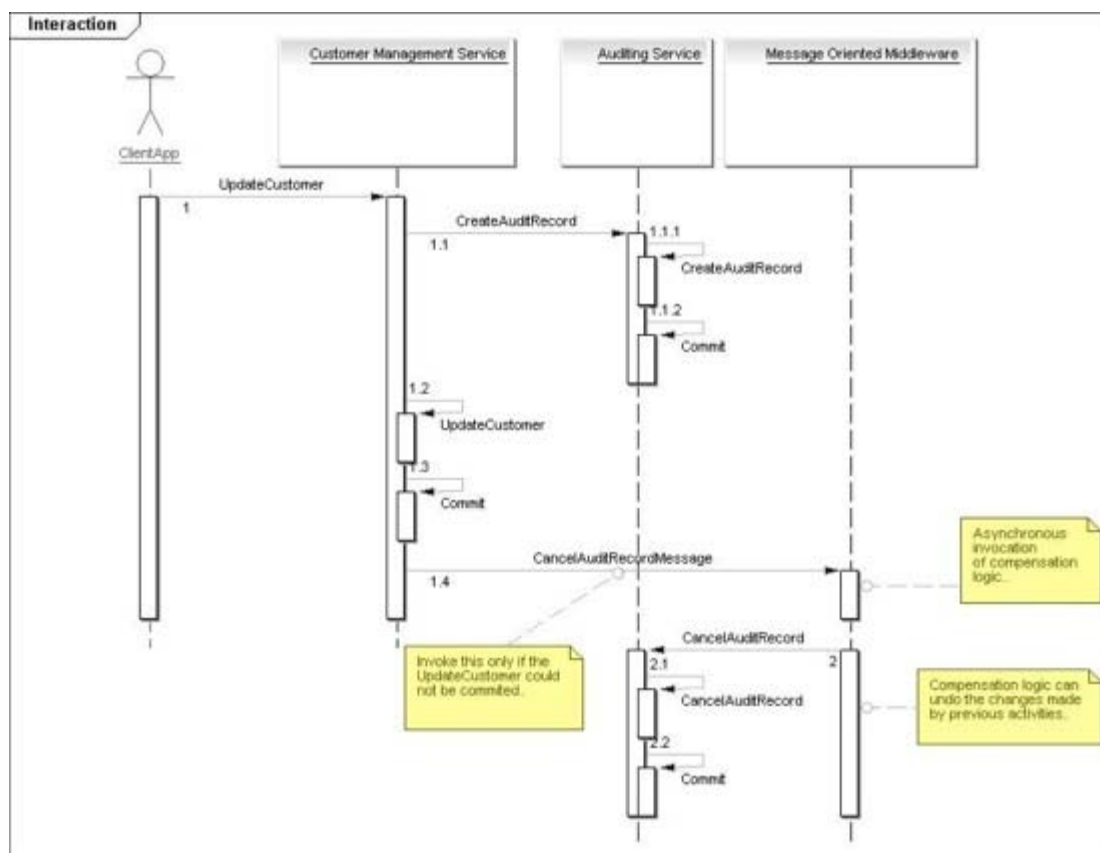
#### Solution 4: Create Auditing Service with Compensation Logic

This last solution (Figure 6) advocates providing a mechanism to rollback changes via compensation logic. This approach is appropriate for long running transactions where a process might span multiple days and may have several heterogeneous disconnected systems processing data.

For our example, it could be applied as follows:

- The Audit Service provides another operation to cancel audit records by setting a flag.
- All services that need to create an audit log, do so as the first activity. After successful creation, they perform the update of their own entity.
- If the update of the entity fails, they invoke the Cancel operation of the Audit Service.

Figure 6: Solution 4



Because our example has only two services, this solution may look trivial to implement. However, the complexity can grow tremendously if there are more services and operations participating in the transaction.

Care must be taken to design and implement compensation logic. For example, it may be a good idea to use Reliable Messaging [REF-4] mechanisms to support compensations (e.g. by using JMS transports) in case a Cancel operation fails. If it were carried out synchronously, then once it fails, it may not be possible to cancel the audit record anymore. If it is implemented asynchronously, the cancellation can be retried until it succeeds.

The benefit of this solution is that it maintains the separation of concerns among participating services. Each service remains autonomous and there is no duplication of logic. It also does not have the additional burden of creating, maintaining and governing the shared library. The solution remains available for heterogeneous platforms and the logic to manage the audit records remains abstracted from other services.

But, here come the cons. First, we lose real time updates. The cancellation of an audit record in this example is carried out asynchronously and thus there may be a brief period where the audit record may exist in the system as a valid record. This problem is not insurmountable and may be acceptable in many cases. Since the compensation logic may require an asynchronous mechanism, this solution can also increase the complexity.

## Conclusion

As a service inventory grows, we will likely need to wrap service compositions into transactions. If your platform does not provide support WS-Coordination and WS-AtomicTransaction, or if these standards are insufficient for your requirements, you are left having to explore alternatives. This article has documented several alternative solutions and compared their impacts.

## References

[REF-1] WS-Coordination specification

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-Coordination.pdf>

[REF-2] WS-AtomicTransaction specification

<http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-AtomicTransaction.pdf>

[REF-3] "SOA Principles of Service Design", by *Thomas Erl (Prentice Hall)*

[REF-4] "SOA Design Patterns", by *Thomas Erl (Prentice Hall)*

[REF-5] "Web Service Contract Design and versioning", by *Erl et al (Prentice Hall)*

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL



[Home](#) [Past Issues](#) [Contributors](#) [What is SOA?](#) [SOA Glossary](#) [SOA School](#) [SOA Books](#) [About](#) [Legal](#)

Copyright © 2006-2009  
SOA Systems Inc.