Middleware Mediated Transactions
&
Conditional Messaging

Expert Topic Report
ECE1770 Spring 2003

Submitted by: Tim Chen
John C Wu
To: Prof Jacobsen
Date: Apr 06, 2003

Electrical and Computer Engineering
University of Toronto
I. Introduction

Information systems in large enterprises have undergone a series of changes in the past few decades. At the beginning, most enterprises only have one information system, most likely a mainframe system. As technology evolves rapidly, enterprises have adopted a number of new technologies to provide their customers with services through a number of new channels. Using banking system as an example, at the beginning, people need to go to a branch, and a teller will carry out a transaction through a terminal that’s connected directly to the mainframe system. Later, banks adopted Interactive Voice Response (IVR) technology so that customers can do self-serve banking through telephones. As Internet became more and more popular, customers can now execute transactions online through a web-browser. However, many of these new systems are designed and implemented as solo systems, which not only introduces a lot of redundancy but also makes maintenance a nightmare. Businesses today are more interested in looking for ways to consolidate these solo systems to make the best use of limited resources and to ease the effort in maintaining the systems. This process is referred as Enterprise Applications Integration (EAI). In EAI, each of these independent systems can be looked at as a component, which has its own application process, common data and multiple users [1]. The exercise of EAI becomes a process to connect these components. While explicit Middleware Mediation is often considered to be used to connect these heterogeneous components because of their flexibility, object transaction processing is also attractive because it addresses issues of system reliability and correctness. A common question quickly rises, “Wouldn’t it be great if both flexibility and transactional support can be achieved?” Middleware Mediated Transaction is therefore introduced to address this. Section II of this report presents Middleware Mediated Transactions. The basic concepts of Middle Mediation and Transactions are first reviewed. Then, a sample scenario is used to demonstrate the concept and techniques of Middleware Mediated Transactions. Section III, then presents an implementation of Middleware Mediated Transactions, conditional transaction using D-sphere by IBM research.

II. Middleware Mediated Transactions (MMT)

2.1 Middleware Mediation

“Middleware Mediation refers to the indirection established by middleware for interaction among two or more (distributed) components [1].” This mediation can be implicit or explicit. Components participating in implicit mediation are not aware of any intermediation by middleware. Instead, they each maintain a reference of the other involving components and interact directly with the other involving components. CORBA promotes implicit middleware. Contrarily, with explicit mediation, components are well-aware of intermediation by middleware and make explicit use of a mediator to interact with each other. The mediator is a distinct entity that “decouples” participating components. IBM’s MQ and Sun’s JMS promote explicit mediation.
The connection between two or more components can be described in terms of topology and binding. Topology refers to the number of communication partners. It can be 1:1, 1:n, or n:m [1]. Binding refers to the means by which a relationship is established among components. It can be reference-based (implicit mediation) or mediator-based (explicit mediation) [1]. Component interaction is described through life-cycle dependency and synchronicity [1]. Life-cycle dependency is sometimes referred as time dependency. If the components are time-dependent, they need to be all available throughout the interaction. If they are time-independent, they do not need to be all available at the same time. The interaction among components can be synchronized, in which case the requesting component is blocked after sending out the request until it receives a response from the responding component, or asynchronous, in which case, none of the participating components is blocked. The interaction can also be carried out in a deferred synchronous manner, in which case, the communication is at first asynchronous, but at a defined later point in time, a synchronization is required [1].

The reliability of interaction among components can be considered using two approaches. It can be applied to the guarantee of delivery of a request to one or more final recipients. It can also be applied to the guarantee of processing of such request by recipients. The delivery of a message can be not reliable (best-effort and at-most-once delivery) or reliable (at-least-once and exactly-once). The processing of the message can be not reliable (best-effort) or reliable. When it is reliable, it can either be part of the requester side of transaction or it can form its own transaction and be coupled with the requester side of transaction [1].

2.2 Transactions

The term “transaction” includes many concepts. It first appeared in the database community. It has been defined as any operation that satisfies the four ACID properties, where ACID is an acronym that stands for Atomic, Consistent, Isolation, and Durability.

A database provides integrated transactions. Client requests are sent to database and the transactional execution guarantees are monitored and enforced by the Database Management Server (DBMS). Such transactions are all controlled by the database, and inter-component communication is not included in the transaction. Therefore, the system is considered to be closed [1].

Distributed Object Transaction still needs to support ACID properties. But some of the properties are more difficult to implement in this case. For example, since different components participating in the transaction maybe at different sites, it is difficult to ensure atomicity to ensure either all or none committed. Also, failure may only affect part of a transaction. Commitment must occur “simultaneously” at all sites, and delayed commit is very difficult to implement. A centralized transaction system preserves properties when it commits, while distributed transaction systems
needs to make use certain protocol, such as two phase commit (2PC) protocol, in which a central coordinator is used, to achieve this.

Message Oriented Transaction addresses message-oriented middleware. In this scenario, the enqueing/dequeing of messages and publishing/consumption of notifications is enclosed in a unit-of-work (UOW). And this UOW can be dependent on the overall transaction outcome or vice versa. The UOW can usually be included in the sphere of atomicity of a sender’s transaction, but not in the receiver’s side. Therefore, a transactional interaction in MOT typically spans across more than one transactional context. Even though, all these contexts are logically coupled, there is no standard middleware support to express or enforce such coupling relationship [1].

2.3 Middleware Mediated Transactions

2.3.1 Sample Scenarios

To present Middleware Mediated Transactions (MMT), couple sample scenarios are used for a typical architecture shown in figure 2.3.1.1.

![Figure 2.3.1.1: MMT Scenario]

In the figure, some components, namely object1 through object 3 participates in a distributed object transaction context. In addition, further interaction with more
components, namely recipient1 (and others) and consumer1 (and others) is required through the use of explicit mediation through queue and pub/sub based mediators. Assume that the recipients are autonomous components such that they do not support 2PC protocol, and they carry out separate business functions. In a normal case, the message would eventually be delivered to and be processed by recipients, and the earliest point such message delivery starts would be after the object transaction performed by object1…object3, tx, is committed. However, this may be too weak as reliability. It is often required that some specific recipient will definitely receive the message within a particular time frame, while the transaction is still ongoing. This is not achievable through common messaging transaction. With MMT, however, immediate message visibility is allowed ad a backward dependency of message delivery and/or processing on the outcome of tx, i.e. tx may only commit if the delivery/processing conditions are satisfied.

As another example, assume the OTS context in figure 2.3.1.1 represents a company’s Customer Relationship Management (CRM) system, and the Consumers represent a company’s Enterprise Resource Planning (ERP) system. The transaction tx in the CRM system creates a new account for a new customer. tx also publishes a message to notify the ERP system to create an account for the same new customer. MMT allows immediate visibility of such notification, so that the ERP system can start its transaction without waiting for tx to be committed. However, such immediate visibility of the notification might create a dirty read situation, in which case tx may later abort, while the ERP system already started/possibly committed its transaction in creating the new account. A forward dependency needs to be enforced implicitly such that if tx aborts, this abort needs to be propagated to the Consumers as well so that it can either abort its transaction or carry out compensating operations if the transaction has already committed. Therefore, MMT needs to provide a way to realize the transactional processing reliability between the two coupled transaction contexts.

2.3.2 Coupling Modes

In order to for the application developers to integrate mediated interactions in object middleware transactions in a flexible way, the notion of coupling modes is introduced.

Coupling modes determines the way that the message mediator relays notifications published in a producer’s transaction to the delivery and/or the processing of such notifications at the consumer [1]. The table in Figure 2.3.2.1 summarizes the properties that constitute a coupling mode.
<table>
<thead>
<tr>
<th>Visibility</th>
<th>Immediate, on commit, on abort, deferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>None, shared, separate</td>
</tr>
<tr>
<td>Forward Dependency</td>
<td>None, commit, abort</td>
</tr>
<tr>
<td>Backward Dependency</td>
<td>None, vital, mark-rollback</td>
</tr>
<tr>
<td>Production</td>
<td>Transactional, independent</td>
</tr>
<tr>
<td>Consumption</td>
<td>On delivery, on return, atomic, explicit</td>
</tr>
</tbody>
</table>

**Figure 2.3.2.1: Coupling Modes**

- **Visibility**: This refers to the earliest pint in time with respect to a producer’s transaction at which a message is relayed to a consumer by the mediator. A message is sent to consumers without waiting for the transaction to complete with *immediate* visibility. With *on commit* or *on abort* visibility, a message is sent after the producer’s transaction has committed or aborted. *Deferred* visibility indicates that the message is sent as soon as the producer transaction starts the commit processing.

- **Context**: This refers to the transaction context at the consumer’s side. A *shared* context is established when the consumer joins the producer’s transaction context propagated by the mediator. A consumer may also run in a *separate* transaction context. In some cases, the mediator has no influence to transaction management, then the consumer might not run in any transaction context.

- **Forward Dependency**: This indicates the way the consumer’s reaction is dependent on the producer’s transaction outcome. *Commit* and *abort* forward dependency specifies that the consumer’s transaction may only commit if the producer’s transaction has committed or aborted, respectively.

- **Backward Dependency**: This indicates the way the producer’s reaction is dependent on the outcome of the consumer’s transaction. If *vital* coupling is applied, the producer’s transaction may only commit if the consumer’s transaction has completed successfully. If the consumer is coupled in *mark-rollback* mode, the sender’s transaction is not dependent on the consumer’s
transaction outcome, but the consumer may explicitly mark the producer’s transaction as **rollback-only**

- **Production:** Transactional production implies that the message delivery to the mediator is dependent on the producer’s transaction success and vice versa. With **independent** production, the application decides the reaction in case the delivery of the message fails.

- **Consumption:** The notification may be consumed by accepting the notification (**on delivery**) or when returning from the reaction (**on return**) or it may be bound to the consumer’s transaction atomicity sphere (**atomic**) or it may be **explicitly** indicated by the application at some point.

It is apparent that not all combinations here are meaningful, and some of them overlap. There are also issues related to the definition, semantics, realization and application of these coupling modes. Due to lack of space, this will not be discussed in detail.

### 2.4 Implementation of MMT

Middleware Mediated Transaction is essentially an extension to the conventional distributed object transaction model and it integrates explicit mediation provided by standard messaging middleware [1]. It needs to provide transaction services, explicit messaging mediation, as well as features that realize the coupling modes introduced in section 2.3.2. The integrated service can either introduce an indirection for use of the integrated transaction and messaging services, or expose the underlying service functionality and API directly to the application clients [1]. There are two implementation developed independently. One is the Dependency-Sphere service developed at IBM research. The other is the X²TS developed by Darmstadt University of Technology. Section III will discuss conditional messaging using D-Sphere in detail.

### III. Conditional Messaging using D-Sphere

#### 3.1 Traditional Transaction Middleware

As we discussed above, middleware is application-independent software that is used for the integration of different software component in a heterogeneous environment. As to all other software component, reliability and scalability are also the two main concerns of middleware. For this reason, there are two different approaches in the implementation of the traditional middleware system, the object-oriented system and message-oriented system.

##### 3.1.1 Object-Oriented Middleware

Like the name, OOM takes an object-oriented approach to the problem and offers a well-defined interface and offers synchronous, typed communication between components of a distributed program. As most of the new applications are based on
object-oriented language and design, it's very natural for this type solution. OOM typically consists of a mechanism to allow methods to be invoked on remote objects, plus services to support the naming and location of objects in a system-wide manner. Examples of OOM are Java RMI/JINI and CORBA.

The advantage of OOM is that the object-oriented paradigm is followed in modern system design and development, thus supports consistent development process. It also provides the system with the reliability required.

3.1.2 Message-Oriented Middleware

The idea of MOM is to decouple the software components from synchronous to asynchronous by using generic message to exchange information between distributed applications. The MOM system ensures message delivery by using reliable queues and by providing the directory, security, and administrative services required to support messaging.

The advantage of MOM is that different software components don’t have to be available at the same time. It allows and promotes the decoupling the components in time and in space. MOM increases the flexibility of the architecture by enabling applications to exchange messages with other programs without having to know what platform or proccessor the other application resides on within the network.

3.1.3 Problem of traditional middleware

Existing middle solutions provides with very limited solution with the integration of transaction and messaging. They either restrict the programming flexibility or have no quality-of-service guarantee.

Consider the following example. An application need to create a new account, to complete this transaction, two different accounts has to be created in two different legacy systems, one in the ERP system and one in CRM system. The ERP system was developed using EJB and the interface to the CRM database must through IBM MQ messaging interface. The problem raises as the transaction context can across the messaging system and creation of the two accounts can’t be completed within one transaction.

The current off-the-shelf middleware has no support for such transaction. The current message queue is a kind of one way queue. It guarantees the message to be delivered once but offers no result of the message processing. There are two way to solve this problem, one way is build some management of the message process result module in the application and handle the result by the application or to build some intelligence into the message queue to allow the result of the message processing passed back to the sender.
3.2 Conditional Messaging

As we all know, messaging uses generic messages to send application data. In addition to the regular application data, control information can also be appended to the original message to help the system decide the outcome. The information that’s appended to the original message is defined as condition.

Conditional messaging can be defined as following: Conditional messaging is messaging in which messages are associated with application-defined conditions on message delivery and message processing in order to define and determine a messaging outcome of success or failure. Conditional messaging is an extension of standard, proven messaging middleware, and application should be able to continue to use the standard messaging middleware directly.

By using conditional messaging, the implementation of management of conditions on messages is shifted from the application to the middle. By associate the outcome of a transaction with the condition of a message, we can achieve the result that can’t be done by the standard messaging middleware.

3.2.1 Representation of condition

The characteristic of condition is following:

- multiple intermediary destinations (queues) may exist,
- multiple known or anonymous final receivers may exist,
- final receivers may or may not be required to acknowledge message receipt,
- final receivers may or may not be required to process the message and acknowledge processing success, and
- various time constraints on receipt or on message processing may be defined, specific to a particular receiver or a group of receivers, or independent of any receivers (as default or any receiver).

In this implementation, the composite design pattern is used for the object model. There are three classes in the class diagram, Condition, Destination and DestinationSet.

![Figure 3.2.1 Object Model for Conditions](image)
Condition class is the base class. The destination and DestinationSet class will have all the attribute and method of condition class. This way, one condition can be applied for multiple destinations.

The condition class defines the criteria of success message delivery by using the MsgPickupTime, MsgExpiry attribute. If the MsgProcessingTime attribute is set, the condition also defines the success criteria of message processing. The above is only one implementation of the condition. In real practice, more sophisticated design may be applied to the system.

### 3.2.2 Conditional Messaging Architecture

Figure 3.2.2 shows the overview architecture of conditional messaging:

![Conditional Messaging Architecture Overview](image)

As we can see from the above figure, conditional messaging is an extension of the standard messaging middleware. It’s an extra layer sitting on top of the traditional messaging component. By calling the conditional messaging API, the sender can send the original application data together with the predefined condition to the final recipient, the message receiver can use the condition to decide the successful delivery of the message and send the acknowledgement back to the sender. In case of the fail of the delivery of the message, a compensation message will be sent by the Compensation Engine. (The acknowledgement and compensation will be discussed in 3.3.2). The Evaluation Manager can decide the outcome of the message by reading the message from acknowledge queue and send the result to the outcome queue.

Sender can also use the standard messaging middleware API by calling it directly to bypass the conditional messaging.
3.3 D-Sphere

The concept of Dependency Sphere is to provide a global transaction context to enable transaction processing across object and messaging component. D-Sphere integrates the standard distributed transaction and conditional asynchronous messages in one single unit-of-work. It extends the traditional two-phase-commit protocol with pre-commit message delivery, condition evaluation and support of message compensation and recovery.

3.3.1 A Global Transaction Context for Objects and Messages

The Dependency Sphere is introduced to solve the problem of how to enable transaction object across messaging middleware. The innovation of D-Sphere is to take advantage of both MOM and OOM by integration.

![Figure 3.3.1 D-Spheres](image)

The figure above shows how D-Sphere integrates OOM transactions and MOM messaging and gain the reliability of traditional transaction and the flexibility of messaging. Through the D-Sphere, the transaction can be made to success only if the messaging succeeds and the messaging can be made succeed only if the transaction succeeds.

3.3.2 D-Sphere Service Architecture

From the above discussion we can see the easiest to implement D-Sphere is to add an additional layer of abstraction above the standard MOM and OOM. The add-on layer provides the integrated transaction and messaging services for application development.

The service architecture of D-Sphere is following:
From the above figure we can see that the application that is using D-Sphere uses the D-Sphere API for transaction management and for conditional messaging. The D-Sphere Messaging module is responsible for sending and receiving traditional messages. D-Sphere Management module will take care of evaluation and compensation. The client doesn’t initial transaction using the D-Sphere service, instead, it will manage the transaction resource by itself, but the transaction result will be associated with the result with the D-Sphere result.

3.3.3 D-Sphere Messages
3.3.3.1 Acknowledgments

Acknowledgement is the message sent by the final recipient to acknowledge the receiving of the message. It is also used to determine numbers or identities of the final recipients and different results of the message processing. There are two type of acknowledgment in D-Sphere:

- An Acknowledgment of a successful unconditional read of a message by a final recipient
- An acknowledgment of a successful conditional read (successful processing) of a message by a final recipient.

3.3.3.2 Commit Protocol

To maintain the reliability of the D-Sphere, the delivery of the messages to the destination need to be monitored and the conditions need to be evaluated to determine the success or failure of message delivery and processing. The D-Sphere commit protocol is very similar to the two-phase-commit protocol. In fact, it’s an extension of the two-phase-commit protocol by adding evaluation of the conditions of the messages to the voting phase of the transaction stage.

Figure 3.3.2 D-Sphere Service Architecture
By using this protocol, D-Sphere makes the conventional object transactions depend on the asynchronous messages, and vice versa.

3.3.3.3 Compensation

In the event of failure, recovery of the messages must be performed. The compensation has to be sent to all the destinations to which the original message has been delivered.

There are two approaches to the implementation of compensation message. One is to send a system-generated that contains no specific data, just simply tell the recipients that the conditional message failed and the recipient has to undo the original message, the other way is to send application-defined message that contains any data that is needed to undo condition message.

IV. Conclusion

One of the major challenges of the current middleware system is to provide reliable and flexible interaction between heterogeneous and autonomous components. Current MOM system OOM provides either the flexibility or the reliability but not both. In this paper, we discussed the concept of MMT, which is an extension of the conventional distributed object transaction model. MMT integrates the standard message interaction into the standard distributed transaction and thus provide the flexibility for object transaction. MMT also defines the sphere of reliable message delivery to and of processing by. With MMT, dependencies between multiple sender and receiver transaction contexts can be established. MMT also support recovery of messages through compensation for messages sent with immediate visibility.

The later part of the paper discussed one implementation MMT, the Dependency Sphere and Conditional Messaging. D-Sphere implements the transaction context by using the combination of an object-oriented architecture with persistent queues and MOM messaging transaction. Limited by the space, this paper only gives the brief overview of the implementation. For details, please refer the reference [2] and [3].
References

