Conditional Messaging: Extending Reliable Messaging with Application Conditions

Stefan Tai, Thomas Mikalsen, Isabelle Rouvellou, Stanley M. Sutton Jr.

IBM T.J. Watson Research Center, New York, USA
{stai | tommi | rouvellou | suttonsm@us.ibm.com}

Abstract

Standard messaging middleware guarantees the delivery of messages to intermediary destinations like message queues, but does not guarantee the receipt or the processing of a message by final recipients. Conditional messaging is an extension to standard messaging middleware that addresses this shortcoming by allowing an application to define, monitor, and evaluate various conditions on messages, such as time constraints on the receipt or the processing of a message by a set of final recipients. In this paper, we introduce the notion of conditional messaging, and present the design and implementation of a flexible and reliable system that supports conditional messaging for use in Java 2 Enterprise Edition and message queuing environments. Our solution uniquely shifts the responsibilities for implementing the management of conditions on messages from the application to the middleware. We further discuss the grouping of multiple conditional messages into atomic units-of-work, which can also integrate requests to transactional resources like distributed objects using object middleware. Conditional messaging serves to implement various kinds of backward dependencies for distributed object transactions that integrate messaging.

1 Introduction

The development of enterprise systems frequently entails the integration of diverse existing applications. Middleware is application-independent connectivity software that is commonly used for purposes of Enterprise Application Integration (EAI). Middleware enables EAI by providing services that mediate between applications, including basic services of data translation and conversion mechanisms and more complex services of business process definition and management.

Some form of messaging and messaging middleware is often used for EAI [11]. Applications create, manipulate, store, and (typically asynchronously) communicate messages using the services of messaging middleware. A message can be any application data combined with some control information. Messaging middleware is exemplified by products such as IBM's MQSeries [5] and messaging services to object middleware such as implementations of Sun's Java Message Service (JMS) [12,6].

Messaging is considered beneficial to EAI as it does not require applications to be tightly coupled. Messaging applications do not directly communicate with each other, but are mediated by the messaging middleware in the form of message queues and/or publish/subscribe message brokers. Mediation and asynchrony support resilience and robustness in cases of partial failure.

Messaging applications rely on the messaging middleware to distribute messages. Standard messaging middleware typically guarantees eventual message delivery to intermediary destinations like queues [1,7], but does not guarantee that an enqueued message will also be read from the queue and be processed by some application.

From the message sender's perspective, however, message receipt and message processing by final recipients often are important criteria that represent a condition on further processing by the sender. For example, in a coordinated workflow management system, a message representing the notification of a group meeting is sent to a set of participants, some of which may be required to acknowledge the receipt and accept the meeting before the meeting can be scheduled and databases (such as for room reservation and other purposes) can be updated. Or, in an air traffic control scenario, a message representing an incoming flight is delivered to a queue and must be picked up from the queue by a controller within a certain time frame, otherwise, some exception handling must be started.

With current middleware, applications themselves are forced to implement the management of such conditions on messages as part of the application. The sender application itself must in some manner define the conditions on message delivery and/or processing and must implement some observation and evaluation mechanism to determine the satisfaction (respective violation) of the conditions. The receiver applications must send explicit acknowledgments of receipt and/or processing back to the sender, and these must conform to the sender's expectations, that is, the sender's particular implementation mechanism for working with conditions. There is no defined middleware support available today to aid in the development of applications that require the management of conditions on messages.

In this paper we address this shortcoming of current middleware. We define and introduce the notion of *conditional messaging* and present a middleware service that shifts the responsibilities for implementing the management of conditions on messages from the application to the middleware. Our solution is compliant to, and built on top of, standard messaging middleware,

which is an important requirement for its application to EAI scenarios. We also present the use of conditional messaging in the context of an extended transaction processing middleware service, the *Dependency-Spheres* service [14], which allows conditional messages to be integrated with distributed transaction processing.

The paper is structured as follows. Section 2 defines conditional messaging and addresses all aspects of supporting the management of conditions on messages; it is structured into seven subsections that successively explain the concepts and the design and realization of our middleware service, using the same recurring motivating application examples in each subsection. Section 3 presents Dependency-Spheres, a novel approach to integrating distributed transactions and messaging that employs conditional messaging as a fundamental element. Section 4 concludes with a summary and discussion.

2 Conditional Messaging

We define conditional messaging as follows: 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 a general notion. Specific models of conditional messaging can be defined with respect to

- specific models of messaging, such as message queuing and publish/subscribe systems,
- particular participant roles, such as sender, publisher, receiver, and subscriber,
- action kinds, such as sending out notifications or request messages, reading messages, subscribing to messages, and processing messages.

For example, conditions can be specified by which the sender of a message may define delivery failure of a notification message in the context of message queuing, or, conditions can be specified by which a subscriber may define processing success of a request message in the context of publish/subscribe messaging.

In this paper, we focus on conditional messaging for message senders in the context of message queuing. We introduce conditional messaging step-wise, as follows:

- the definition and representation of conditions (Section 2.2.),
- the delivery of messages that are associated with conditions (Section 2.3),
- the monitoring of conditional message delivery and processing (Section 2.4),
- the evaluation of condition satisfaction (Section 2.5), and
- the support for evaluation outcome actions of compensation and success notifications (Section 2.6).

Each section covers general concepts as well as design and implementation aspects. We present the conditional messaging system that we have developed, and discuss application examples alongside. The system is summarized with an architectural overview in Section 2.7.

2.1 Running Examples

Consider again the two examples described in Section 1 Introduction. We will use these two examples as running examples in this paper.

In the first example, a message representing a group meeting notification is sent to a set of particular queues, where each queue is designated to a specific final recipient. Figure 1 illustrates this example.

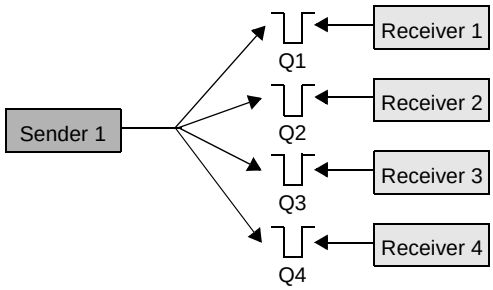

Figure 1: Example 1

Sample conditions for message success are:

- All four recipients reading from the four different queues must acknowledge message receipt within two days after the message has been sent out.
- `Receiver3` must successfully process the message (must update his calendar database) one week ahead of the meeting time, and at least any other two receivers must successfully process the message three days ahead of the meeting time. An acknowledgment of processing success of these receivers is required.

In the second example, a message representing an incoming flight is sent to one central queue from which multiple controllers can read messages. Figure 2 illustrates this example.

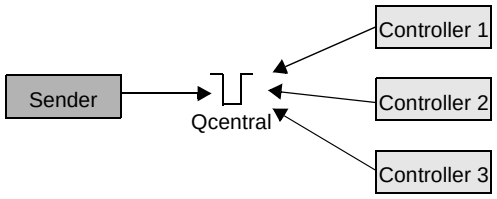

Figure 2: Example 2

A sample condition for message success here is:

- Any one of the controllers must read the message from the shared queue within 20 seconds after the message has been sent out.

These two message queuing examples and the sample conditions already exhibit some of the variety of messages and conditions that are possible. We can observe two purposes of an outgoing message, as

- a one-way event notification (distribution of some data), or as
- a request for processing by receivers.

For both kinds of messages,

- 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 for any receiver).

These kinds of conditions are commonly encountered in EAI scenarios, and are typically implemented through various application artifacts. This paper introduces the concept of conditional messaging and describes the design and implementation of a conditional messaging system that covers these kinds of conditions. Our objective is to explore and motivate the idea of conditional messaging. We expect that new (and more sophisticated) versions of conditional messaging systems will emerge in the future, which in addition may support other kinds of conditions as well.

2.2 Definition and Representation of Conditions

Conditions such as those described above can be modeled in various ways. We have chosen an object representation of conditions where conditions are modeled using three classes, the `Condition` class, the `Destination` class, and the `DestinationSet` class. These are depicted in Figure 3.

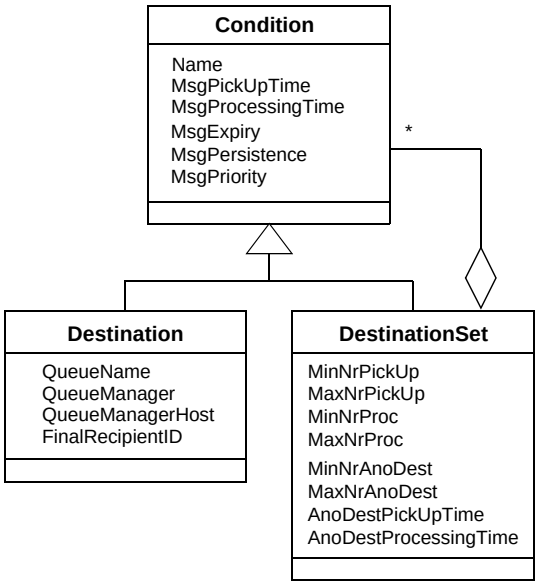

Figure 3: Object Model for Conditions

The classes follow the *Composite* design pattern [4] for defining conditions on individual destinations and on (hierarchies of) sets of destinations. The base class is the `Condition` class. The `Condition` class defines as its interface the methods to set and get the values of its attributes, and methods for accessing and managing child components according to the Composite design pattern. The `Destination` class is the leaf class that represents conditions on a particular destination. The `DestinationSet` class is the composite class that represents conditions that apply to a set of destinations.

The attribute of `MsgPickUpTime` specifies the time during which a message read is required by a final recipient, and the attribute of `MsgProcessingTime` specifies the time during which a successful processing of the message is required. Both time values are set in milliseconds and are interpreted relative to the sender's time clock and the timestamp of sending the message.

A `Destination` must specify a unique queue, and may specify an identification string for a final recipient (for example, a defined name such as a userid in a namespace). If time conditions such as those described above are specified for a `Destination` object, the destination is a *required destination* (a message read and/or processing is required). If no time conditions are specified on a `Destination` object, but are specified on its parent `DestinationSet` object, the destination is an *optional destination* (time conditions specified on a set may be satisfied without a read or processing by the particular destination).

Time conditions on a `DestinationSet` object apply per default to all members of the set, unless a subset of minimum and maximum numbers of destinations of the set for message pick up or message processing is defined. The `MsgPickUpTime` condition on a `DestinationSet` object, for example, is a condition for all members of the set unless a `MinNrPickUp` and/or `MaxNrPickUp` value is specified. A `DestinationSet` object may also specify minimum and maximum numbers for anonymous destinations.

Other attributes, including `MsgExpiry`, `MsgPersistence`, and `MsgPriority`, are common properties of standard messaging middleware that can be used to set the (general, destination set-specific, or destination-specific) expiration time of a message, the persistence property of a message, or the priority for delivery and placement of the message on a queue.

The condition objects define the *success* criteria for *message delivery*, or, in case that the attribute of `MsgProcessingTime` is set, the success criteria for *message processing*. They serve for defining and representing the conditions of the two examples described. Figure 4 and Figure 5 depict the object instances as they would be defined for the two examples.

Figure 4 for Example 1 shows four `Destination` objects, and two `DestinationSet` objects. The `destSetRoot` object specifies a `MsgPickUpTime` as a condition on all the destinations. For the destination `qr3`, a `MsgProcessingTime` value is set in addition. The `destSet1` object groups the three other destination objects and specifies that at least two of these must process the message in the `MsgProcessingTime` specified.

Figure 5 for Example 2 shows one `Destination` object only. The `Destination` object specifies a

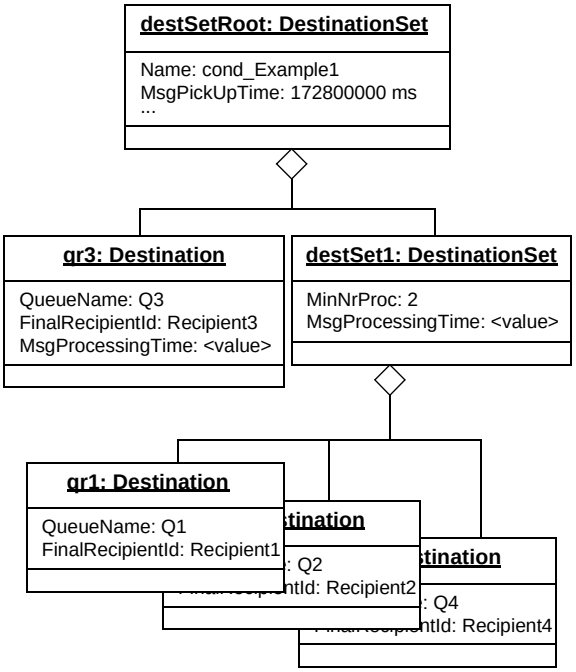

Figure 4: Conditions for Example 1

`QueueName`, but no particular final recipient. The object also specifies a `MsgPickUpTime`, but not a `MsgProcessingTime`.

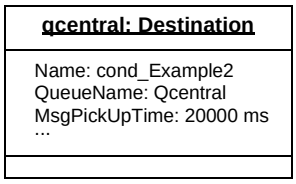

Figure 5: Conditions for Example 2

2.3 Association of Conditions to Messages and Message Delivery

Conditions can be defined and represented independently of a message, as described above. The separation of condition definition and condition representation from message creation allows conditions to be reused for different messages. Specific conditions may apply to all messages processed by a messaging application, to groups of messages processed by the application, or (most generally) to individual messages processed by the application.

We have implemented the condition classes as a set of Java classes that can be used in combination with the JMS standard for messaging in Java systems (as well as with IBM's MQSeries Java classes). In order to associate specific condition objects with specific JMS or MQSeries messages, a dedicated API is needed. However, the use of a special conditional messaging API should not compromise the use of standard messaging middleware like JMS and MQSeries, as these are often required in an EAI setting for integration of different legacy systems. Conditional messaging should be implemented as an extension to standard, proven messaging middleware, and applications should also be able to continue to use the standard messaging middleware directly.

In order to associate conditions with messages, we define a method `sendMessage (Object, Condition)` on a Java interface for conditional messaging that takes as parameters an instance of `Java.lang.Object` and an instance of the `Condition` class. The Java object parameter is any application data that is to be represented and exchanged as a message. (An additional method for sending conditional messages that have application-defined compensation support is also provided, as will be described in Section 2.5).

Depending on the condition, multiple JMS or MQSeries messages may be generated by the conditional messaging system for the conditional message. For example, if the Java object is a text string, and the condition specifies four different queues as required destinations, then four JMS messages of JMS type `TextMessage` are generated (as JMS does not support multiple queue distribution lists for a single message). Conditional messaging thus introduces two levels of messages: the conditional message level, and the level of standard (JMS or MQSeries) messages that are used to implement the conditional message.

The generated standard messages contain the application data (as provided and understood by the application), and are attributed by the conditional messaging system with control information required for purposes of monitoring and evaluating the conditional message. For example, each generated JMS message will carry as a property the unique id that represents the conditional message. It will also encode whether or not the processing of the message has been required for the particular destination. Further, it contains information about the sender, such as the sender's queue manager, in order for the recipient's conditional messaging system to be able to send an acknowledgment and reply back to the sender. The conditional messaging system in addition creates a log entry for the outgoing messages and stores the log entry persistently on a local message queue (`DS.SLOG.Q`).

An application uses the conditional message id to later associate outcome notification messages of condition evaluation results with a conditional message. Outcome notifications are sent by the conditional messaging system to the sender's `DS.OUTCOME.Q` as soon as a condition evaluation process (see Section 2.5) has completed.

The conditional messaging API is a simple indirection to standard messaging middleware for purposes of conditional messaging. However, an application can continue to use JMS/MQSeries directly for sending standard (non-conditional) messages. Figure 6 illustrates this approach.

In case of Example 1, the sender application sends a conditional message with the group meeting notification as object data and the `destSetRoot` condition

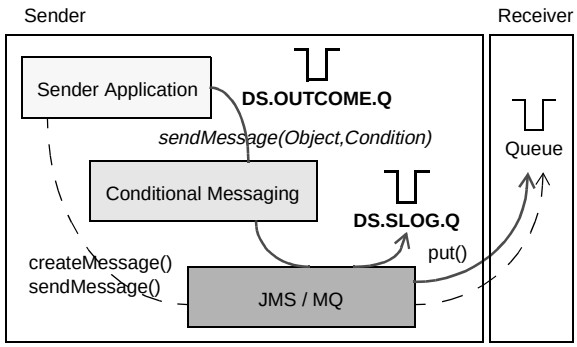

Figure 6: Application Use of Conditional Messaging and Standard Messaging

object of Figure 4. The conditional messaging system then generates four standard JMS messages and distributes these to the four recipient queues.

In case of Example 2, the sender application sends a conditional message with an object representing the flight and the `qcentral` condition object of Figure 5. A single JMS message is generated and sent to the queue that is used by the various controllers.

2.4 Monitoring of Conditional Message Delivery and Processing

Conditional messaging requires more than support for the definition of conditions and their association to messages. It also requires a system for monitoring message delivery and message processing over a given time, in order to be able to evaluate the satisfaction (respective violation) of the conditions.

Our conditional messaging system implements special internal acknowledgment messages for this purpose. A final recipient implicitly initiates the sending of such acknowledgments when successfully reading a message from a queue or completing the processing of a message. Two types of internal acknowledgments exist:

- an acknowledgment of a successful non-transactional read of a message by a final recipient, and
- an acknowledgment of a successful transactional read (and therefore, successful processing) of a message by a final recipient.

The first kind of acknowledgment is generated if a recipient has successfully performed a non-transactional read of a message from a queue. That is, the read did not occur within a recipient's transaction, and the message cannot be put back to the queue due to a recipient's transaction failure.

The second kind of acknowledgment is generated if a recipient has successfully performed a transactional read of a message from a queue. That is, the message read occurred within a recipient's transaction and the transaction committed successfully. (In case that the recipient's transaction failed, no acknowledgment is generated and the message is put back to the queue by the messaging middleware according to the semantics of messaging transactions [1].)

In order for the conditional messaging system to be able to generate these acknowledgments automatically, the final recipients need to use a dedicated API for reading conditional messages. Similarly to the API for sending conditional messages, the use of such an additional API should not compromise the use of standard messaging middleware. As such, we provide additional Java methods that only serve as an indirection to the use of JMS and MQSeries. These include the method `readMessage(String)` and methods that serve as facade methods to the messaging transaction demarcation API of `begin_tx()` and `commit_tx()`. The sending of a reply message by a receiver, however, does not need to be performed using the conditional messaging API. An application can use JMS/MQ directly.

A final recipient attempts to read a message from a queue by calling the method `readMessage (String)` with the queue name as the parameter. The conditional messaging system then checks if the read occurs within an ongoing transaction or not. If no transaction context exists, the first kind of acknowledgment is generated once the message get call to the queue is completed. If a transaction context exists (`begin_tx` has been called), the generation of the second kind of acknowledgment is bound to the successful commit of the receiver's transaction.

In messaging systems, it is common practice to perform the processing of a message in a transaction. That is, the read of the message from a queue, some processing, and possibly the sending of a result message are all executed in an all-or-nothing manner. If the transaction fails for some reason, the message is put back on the queue. An acknowledgment of a successful transactional read therefore corresponds to an acknowledgment of successful processing, if the transaction commits. (An acknowledgment of a successful processing that is non-transactional cannot automatically be generated.)

At the time that an acknowledgment of either kind is generated, information about the final recipient can be encoded in the acknowledgment message. This includes the timestamp of message read or transaction commit. The acknowledgment message further includes management information such as the id of the conditional message that is acknowledged. The acknowledgments thus allow the sender of the message to determine numbers and identities of, and timestamps and other data about final recipients of a conditional message. In this way, the state-of-the-art model of message delivery of conventional messaging middleware is uniquely extended beyond intermediate destinations like queues to include final recipients and their actions.

A designated queue to store the acknowledgments needs to be set up on the sender side (per default, a queue named `DS.ACK.Q` is used for this purpose). This acknowledgment queue must be known to the recipient-side conditional messaging system. Information about the queue is therefore propagated to each recipient as a property on the generated outgoing messages. The conditional messaging system on the receiver side retrieves the information and directs the acknowledgments properly to the right sender and acknowledgment queue. The conditional messaging

system further creates a log entry for each consumed message and puts the log entry on the persistent receiver log queue (`DS.RLOG.Q`). Figure 7 illustrates this model for reading conditional messages.

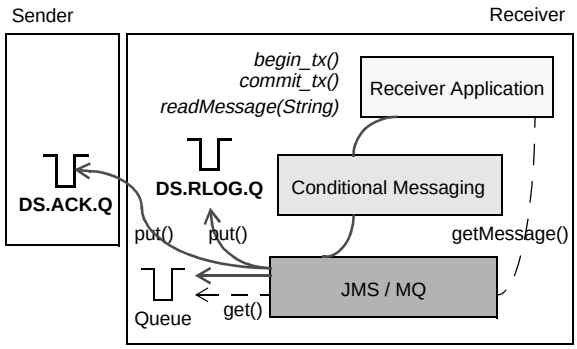

Figure 7: Implicit Acknowledgments for Reading Conditional Messages

The architecture of our conditional messaging system, as indicated with the presentation so far, is a distributed architecture. Responsibilities of conditional messaging are distributed between the sender side and the various receiver sides, with message communication taking place in both directions (for the primary messages and for internal acknowledgment messages). The architecture complies to the architecture of common messaging middleware, where sender and receiver applications each need to explicitly establish a connection to the messaging middleware (typically, a queue manager) in order to send and receive messages. The conditional messaging functionality in this way could be regarded as an extension to existing queue manager functionality.

Consider again Example 1. Each receiver application will implicitly initiate the sending of an acknowledgment message when it successfully reads a message using the conditional messaging API. If the receiver performs a non-transactional read, an acknowledgment of the first kind with a timestamp of the actual message read is sent. If the receiver performs a read inside the scope of a transaction, an acknowledgment of the second kind is sent in case that the transaction commits. This acknowledgment comprises two timestamps: the timestamp of the actual message read from the queue, and the timestamp of transaction commit. Note that there will never be two acknowledgments generated for one receiver reading one message (such as one acknowledgment for receipt, and one acknowledgments for transaction processing commit). A receiver either consumes a message non-transactionally, or consumes the message transactionally. A transactional read of a message cannot be acknowledged if and unless the transaction commits.

For Example 2, an acknowledgment is generated for the one receiver that reads the incoming flight message. The acknowledgment comprises, in both cases of transactional read or non-transactional read, the timestamp of the actual message read from the queue.

2.5 Evaluation of Condition Satisfaction

The monitoring of conditional messages that are delivered to and processed by final recipients is needed in order to evaluate the message conditions so that the success or failure of the message can be determined.

The evaluation of conditions can be started immediately after the message has been sent out, and can be ended when an evaluation result of success or failure is determined. The conditional messaging system further allows a sender to specify a timeout relative to the timestamp of the sending of the conditional message to ultimately terminate an evaluation.

For example, the message sender of Example 2 may specify an evaluation timeout of 21 seconds, as the requirement for message receipt by any controller is 20 seconds. If no positive evaluation result is determined after 20 seconds, the conditional message can be declared to have failed.

The conditional messaging system comprises an evaluation manager that reads incoming acknowledgment messages of the designated acknowledgment queue and interprets them accordingly. Incoming acknowledgment messages must be sorted with respect to the conditional message they address (using the conditional message ids), as the single acknowledgment queue collects acknowledgment messages for an arbitrary number of different conditional messages.

Consider again Example 1. The evaluation of the satisfaction of the conditions can start after the message has been sent out. Four acknowledgment messages are expected, of which at least three must be an acknowledgment of message processing success. Once all four acknowledgments have been received, the individual time values need to be checked. All four acknowledgment messages must carry a read timestamp that is less than the `MsgPickUpTime` specified on the `destSetRoot:DestinationSet` object. The acknowledgment from `Recipient3` must carry a processing timestamp that is less than the value specified on the `qr3:Destination` object, and at least two of the other three acknowledgments must carry a processing timestamp that is less than the value specified on the `destSet1:DestinationSet` object. If any single condition is violated, the overall outcome of the conditional message is declared to be a failure.

When the evaluation process is completed, an outcome notification of success or failure is sent to the sender's `DS.OUTCOME.Q`.

2.6 Evaluation Outcome Actions: Compensation and Success Notifications

Conditional messaging further provides system support for taking some appropriate action in the event that a condition is found to be satisfied or violated.

In the event that a message succeeds (with respect to its conditions), the system can send out a notification message of evaluation success to all destinations. Success notifications serve as confirmations to final receivers that all sender-side conditions were met. In case of our Example 1, for instance, a success notification confirms that the meeting is scheduled to take place.

In the event that a message fails (with respect to its conditions), the conditional messaging system can send out a *compensation* message to all destinations to which the original message has been delivered.

The compensation message can be a system-generated message that contains no specific data, but simply tells the receiving applications that the conditional message failed and that any effects caused by the receipt of the original message need to be undone. This type of compensation message is generated in case `sendMessage(Object,Condition)` was called, and the receiving application needs to understand this simple notification. In case of the Example 1, such a compensation message indicates the cancellation of the meeting, and the receiving applications can be designed to understand such simple compensation even if no additional application-specific compensation data is provided.

The compensation message can alternatively be an application-defined message that contains any data that the application provides to undo effects caused by the receipt of the original message. For this type of compensation message, the conditional messaging API provides the method `sendMessage(Object, Object,Condition)`, where the second object parameter is the compensation data.

Both kinds of compensation messages are generated by the conditional messaging system at the time the original messages are created and sent out, and they are stored on a local persistent queue for compensating messages (`DS.COMP.Q`), as shown in Figure 9. The compensation messages are correlated to the id of the original message, and are only sent out if the conditional message fails.

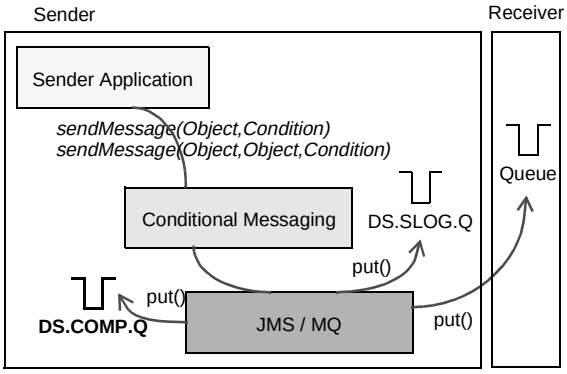

Figure 8: Compensation Queue

On the receiver side, the conditional messaging system correspondingly implements some special behavior for reading messages from queues. In case that both the original message and the compensation message are in the queue (the original message has not been read from the queue), both messages cancel each other out and will be deleted from the queue. The compensation message is only delivered to the receiver application in case that the original message has been read from the queue and a log entry for consumption exists in the `DS.RLOG.Q`.

Compensation introduces some special issues, as the process of compensation must be guaranteed for an application even in the presence of system failures. In [16], we describe how guaranteed compensation can be implemented using three specific message queuing patterns.

2.7 Architecture Overview and Summary

Figure 9 depicts the overall conditional messaging architecture. A sender application uses the conditional messaging service to define conditions, and to send messages associated with conditions. The conditional messaging system provides the respective functionality, and implements an evaluation manager and a compensation manager. Three specific, persistent queues are used by the conditional messaging system for these purposes: the `DS.SLOG.Q` for logging of sender-side actions, the `DS.ACK.Q` for storing incoming acknowledgments, and the `DS.COMP.Q` for storing compensation messages. A sender application can in addition continue to use the underlying messaging middleware directly, for example, to send unconditional messages, or to read incoming standard messages (like reply messages) that were not created by the conditional messaging system.

A receiver application uses the conditional messaging service to read messages that represent conditional messages (messages that were created by the conditional messaging system). It also uses the service to demarcate a transaction, if the transaction comprises the receipt of a conditional message. A receiver-side persistent queue `DS.RLOG.Q` is used to log a receiver's actions. A receiver application can also continue to use the underlying messaging middleware directly, for example, to send an unconditional reply message. Note that any receiver can also be a sender of a conditional message, in which case all queues for sending conditional messages would also exist in addition to the `DS.RLOG.Q` queue.

3 Dependency-Spheres

Conditional messages typically are part of a larger business processing context. For example, the group meeting notification example of Section 2 may be part of a larger coordinated workflow process for some contract negotiation and signing. Or, the flight distribution example of Section 2 may be part of a larger business process for handing over responsibilities for flights leaving one air sector and entering another one.

Such larger business processes may comprise multiple conditional messages, and may also comprise other kinds of actions such as invocations on distributed object applications and updates of distributed databases. The various conditional messages and other actions often need to constitute a single unit-of-work, that is, should be executed in an all-or-nothing manner.

In this section, we present how the *Dependency-Spheres* middleware service [14] can be used to group multiple conditional messages and other actions such as distributed object requests into a single atomic unit-of-work.

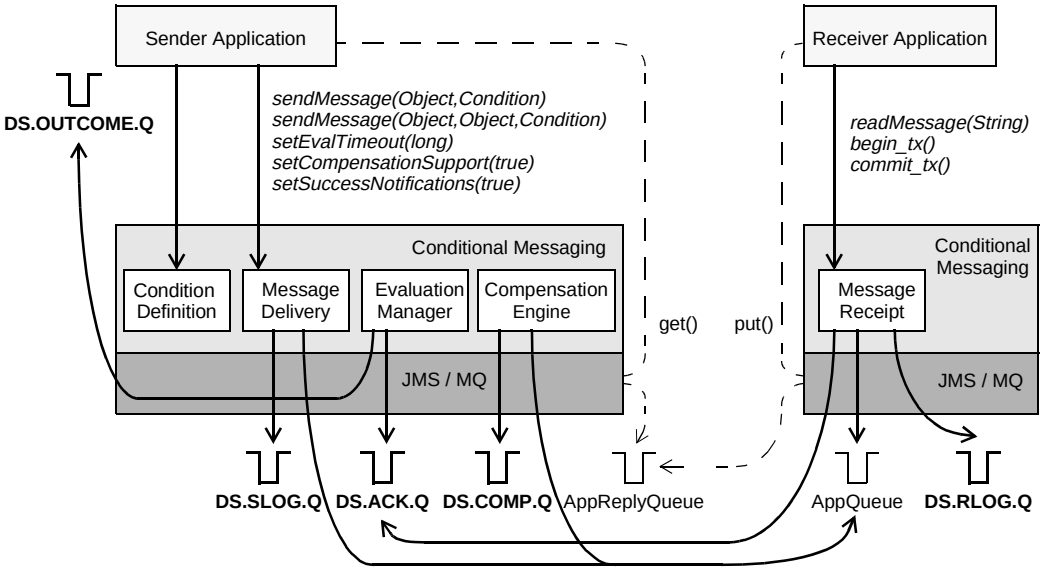

Figure 9: Conditional Messaging Architecture Overview

3.1 Atomic Groups of Conditional Messages

Conditional messaging extends conventional messaging and standard messaging middleware in that an application can define and evaluate a message outcome of success or failure on a per-message basis. A group of conditional messages can also have an *overall group outcome*, based on the individual outcomes of the messages grouped. Similar to distributed transactions that atomically group distributed requests on transactional resources [1], a group of conditional messages can be an atomic unit-of-work that either succeeds as a whole, or fails as a whole. The success of the group is dependent on the success of all constituting messages; if a single conditional message of the group fails, the whole group will have a failure as its outcome.

A *Dependency-Sphere (D-Sphere)* [14] describes such a group of conditional messages. A D-Sphere is a global context inside of which various conditional messages may occur. The D-Sphere is demarcated by the sender of the conditional messages using the verbs of `begin_DS`, `commit_DS`, and `abort_DS`, which compare to the verbs of common transaction services.

With respect to message delivery, conditional messages that are part of a D-Sphere have the same behavior as conditional messages that are outside of a D-Sphere. That is, the messages are sent immediately to all distributed destinations required, and are not bound to the D-Sphere commit. A D-Sphere thus is unlike common messaging transactions that group standard messages and make their publication dependent on a successful commit. D-Sphere conditional messages are sent out immediately, and are then subject to monitoring, condition evaluation, and further outcome actions.

When an evaluation outcome for an individual message is determined, however, no immediate outcome action will be taken for the message, if the message is part of a D-Sphere. Only when the D-Sphere terminates as a whole (through `commit_DS`, `abort_DS`, or a D-Sphere timeout), outcome actions for all individual messages that are part of the D-Sphere will be initiated based on the overall D-Sphere outcome. If a D-Sphere succeeds as a whole, success notifications can be send for the D-Sphere and its individual messages to all destinations involved. Likewise, if the D-Sphere fails as a whole, compensation messages that are correlated to the original messages can be send, as described in Section 2.6.

3.2 Atomic Groups of Conditional Messages and Distributed Object Requests

A D-Sphere not only allows the grouping of a set of conditional messages but also supports the integration of distributed object requests and conventional distributed object transactions into the atomic unit-of-work. D-Spheres follow the model of "message delivery transactions" and "message processing transactions" as introduced in [15] to integrate distributed object transaction processing with messaging, and thus, to better support enterprise applications that use object middleware (like CORBA, EJB) and messaging middleware (like MQSeries, JMS) in combination.

The sender of one or more conditional messages may also invoke transactional resources like distributed objects and databases using the standard invocation mechanism of the transaction object middleware used (such as CORBA OTS [10] and JTS [13]). These transactional requests become part of the D-Sphere, that is, they are associated to the D-Sphere context as established when beginning a D-Sphere. The outcomes of the object requests affect the outcome of the D-Sphere, and the outcome of the D-Sphere affects the individual object actions. In case that a transactional object

request fails, the D-Sphere as a whole fails. In case that the D-Sphere fails, all object requests need to be rolled back. If the D-Sphere succeeds, all object changes become persistent. Figure 10 illustrates the D-Sphere service architecture.

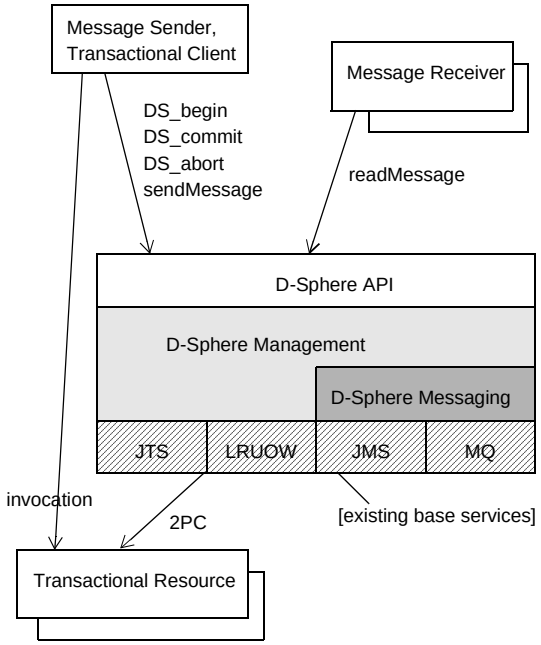

Figure 10: D-Sphere Service Architecture

The D-Spheres transaction service employs conditional messaging, but conditional messaging, as introduced in this paper, is a concept and middleware service that can be used independently of D-Spheres.

4 Summary and Discussion

Messaging and messaging middleware are often used for purposes of enterprise application integration. Messaging middleware guarantees eventual message delivery to intermediary destinations like queues, but does not support the management of application-defined conditions on messages, for example, to determine the timely receipt or processing of a message by a set of final recipients.

In this paper, we addressed this shortcoming of current middleware. We defined the notion of conditional messaging and presented a middleware system supporting the management of application-defined conditions on messages. Conditional messaging is a new middleware solution

- to define diverse conditions on message delivery and message processing in a structured and flexible way, and to represent these conditions independently of messages as distinct condition objects
- to send messages associated with conditions using a simple indirection API to standard messaging middleware
- to monitor the delivery to and the processing by final recipients using automatically generated internal acknowledgment messages of receipt and of (transactional) processing
- to evaluate conditions to determine a message outcome of success or failure
- to perform actions based on the outcome of a message, including the sending of success notifications to all destinations in case of a message success, or, the sending of compensation messages in case of a message failure.

We presented the design of a conditional messaging system which employs both objects and reliable messaging of an underlying MOM. Various persistent message queues are used for purposes of logging, message monitoring and evaluation, and message compensation. The architecture proposed implements conditional messaging reliably on top of standard middleware, and can easily be integrated in J2EE and MQ environments.

The infrastructure used and the messages created to support conditional messaging are those that are needed to achieve the desired qualities of service. If no conditional messaging system were available, the application would have to create similar messages for purposes such as logging or acknowledgments. Therefore, conditional messaging relieves the application developer of this burden.

We further described how the grouping of a set of conditional messages into a single atomic unit-of-work is supported with the Dependency-Sphere transactional middleware service. The Dependency-Sphere service also allows conditional messages to be dependent on distributed object requests that are included in the same atomic unit-of-work.

Conditional messaging can be regarded as a functionality extension to standard messaging middleware like MQSeries and JMS, and could also be supported by middleware products directly. For example, a message queue manager could be enhanced to support conditional messaging.

Conditional messaging uniquely shifts the responsibilities for implementing the management of conditions on messages from an application to the middleware. The definition of conditions remains, as it should, the responsibility of the application.

Conditional messaging extends reliable messaging of MOM to deal with application conditions on messages in a structured and middleware-supported way. The form of conditions supported by our system is appropriate for many applications. Other forms of conditions, for example, conditions expressed in Java or conditions encoded in message bodies, may also be useful for many applications and we expect to support them in the future.

4.1 Related Work

We are not aware of a middleware system that is comparable to the features and architecture of the conditional messaging system described.

However, the importance of message processing assurance beyond the delivery guarantees of current middleware has been pointed out in the literature. The Coyote approach [2], for example, suggests to imple-

ment a timeout-constrained message exchange protocol of acknowledgments and compensation/cancellation messages for a single server. Conditional messaging supports such processing assurance, and further supports the assurance of conditions other than a timeout for message processing by a single server. Conditional messages may have multiple recipients (servers) that can be required or optional, with time constraints set on the individual recipient or on sets and subsets of recipients and with respect to message processing or message receipt only.

The aspect of processing dependencies between messaging partners is also addressed by [8,9] through the definition of "coupling modes." The notion of a coupling mode was originally introduced in the context of active database management systems [3]. A coupling mode as defined in [8,9] includes the specification of forward dependencies and backward dependencies in the context of distributed transaction processing using messaging. For example, if a sender publishes a message as part of a transaction, and the transaction commit depends on the successful commit-processing by a recipient of the message, one kind of backward-dependency is described. Conditional messaging as introduced in this paper allows for a flexible way in specifying different kinds of backward dependencies.

4.2 Future Work

Conditional messaging as defined in Section 2 is a general notion. In this paper, we focused on message queuing as the messaging model, and on sender-side conditions for publishing notifications and sending out processing requests. In our future work, we plan to extend the model for Web environments. This includes more flexible representation of conditions, use of XML in messaging, and message delivery through standards such as the Simple Object Access Protocol (SOAP).

5 References

[1] P. Bernstein, E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufmann, San Francisco, CA, 1997.

[2] A. Dan, F. Parr. The Coyote approach for Network Centric Service Applications: Conversational Service Transactions, a Monitor, and an Application Style. *High Performance Transaction Processing Workshop*, Asilomar, CA, 1997.

[3] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D.R. McCarthy, A. Rosenthal, S.K. Karin, M.J. Carey, M. Livny, R.Jauhari. The HiPAC Project: Combining Active Databases and Timing Constraints. In *SIGMOD Record, Volume 17(1)*, March 1988.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, Massachusetts, 1994.

[5] IBM Corp. MQSeries Application Programming Guide, 10th Ed., 1999.

[6] IBM Corp. MQSeries Using Java, 5th Ed., 2000.

[7] R. Lewis. *Advanced Messaging Applications with MSMQ and MQSeries.* Que Professional, 2000.

[8] C. Liebig, M. Malva, A. Buchmann. Integrating Notifications and Transactions: Concepts and X^2TS Prototype. In *Proceedings 2nd International Workshop on Engineering Distributed Objects* (EDO 2000, Davis, CA, USA, 11/2000), Springer-Verlag LNCS 1999, pp. 194-214, 2001.

[9] C. Liebig, S. Tai. Middleware-Mediated Transactions. In *Proceedings 3rd IEEE International Symposium on Distributed Objects and Applications* (DOA 2001, Rome, Italy), IEEE Computer Society, September 2001.

[10] OMG. Transaction Service v1.1, TR OMG Document formal/2000-06-28, OMG, 2000.

[11] OMG. UML Profile for Event-based Architectures in Enterprise Application Integration. OMG EAI SIG Joint Submission, OMG Document Number ad/2000-8-05, August 2000.

[12] Sun Microsystems. Java Message Service API Specification v1.02. Sun, 1999.

[13] Sun Microsystems. Java Transaction Service (JTS). http://java.sun.com/j2ee/transactions.html

[14] S. Tai, T. Mikalsen, I. Rouvellou, S. Sutton. Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages. In *Proceedings 5th International Enterprise Distributed Object Computing Conference* (EDOC 2001, Seattle, Washington, USA), IEEE Computer Society, September 2001.

[15] S. Tai, I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In Proceedings IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2000, New York, NY, USA), Springer-Verlag LNCS 1795, pp. 308-330, April 2000.

[16] S. Tai, A. Totok, T. Mikalsen, I. Rouvellou. Message Queuing Patterns for Middleware-Mediated Transactions. February 2002. *in submission.*