Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages

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

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

Abstract

Many enterprise systems employ both object-oriented middleware (OOM) and message-oriented middleware (MOM). However, support for the integration of object and messaging services, in particular for transaction processing across object and messaging components, is very limited. In this paper, we introduce the concept of Dependency-Spheres (D-Spheres), a global transaction context for distributed objects and messages. A D-Sphere integrates standard distributed object transactions and conditional asynchronous messages in one single unit-of-work. It is a new service for transaction processing that enhances two-phase-commit ACID transactions with pre-commit message delivery, concurrent evaluations of transaction-critical message conditions, and message compensation support for recovery. D-Spheres enrich standard OOM and MOM services, uniquely providing for an increased level of reliability for their use in combination in enterprise systems.

1. Introduction

Middleware is application-independent connectivity software that is commonly used to integrate diverse software components in distributed and heterogeneous system environments [4]. Middleware defines a communication infrastructure, providing services that mediate between applications and address software quality concerns like reliability or scalability. Two important kinds of middleware are object-oriented middleware (OOM) and message-oriented middleware (MOM). As each kind of middleware may be required for different reasons, such as integration of different legacy systems [17], many enterprise software systems employ both OOM and MOM. In this context, the integration and interoperation of their services becomes a concern.

System reliability, for example, is addressed by services such as object transactions with OOM or guaranteed message delivery with MOM. Object transactions address reliability (and correctness) of object-oriented systems in that the execution of a series of object requests is atomic, isolated, and leads to a consistent and durable end-state [2,18]. Guaranteed message delivery addresses reliability of messaging systems in that the delivery of messages to distributed destinations is guaranteed to take place even in the presence of system failures [2,7]. But how object transactions can be combined with guaranteed message delivery and which forms of those combinations can guarantee which definitions of reliability is still largely an open issue.

Existing middleware solutions to integrating transactions and messaging are very limited, either restricting the programming flexibility drastically or allowing arbitrary, unstructured integration without any quality-of-service guarantees. However, modern enterprise systems demand integration solutions that are structured, yet flexible, and that attain defined levels of quality-of-service. Consequently, application developers today are forced to build their own, complex integration solutions.

In this paper, we address this problem of integrating transactions and messaging for combined OOM and MOM system environments. We introduce the concept and middleware service of *Dependency-Spheres (D-Spheres)* as a structured and reliable yet flexible approach to integrating transactions and messaging. D-Spheres define a new type of a global transaction context inside of which conventional object transactions and distributed messages may occur. D-Spheres make transactional synchronous object requests dependent on asynchronous messages, and vice versa.

The paper is structured as follows. In Section 2, we review background on OOM, MOM, and conventional transaction processing. In Section 3, we introduce the concept of Dependency-Spheres, a new type of a global transaction context for distributed objects and messages. Section 4 describes the D-Sphere prototype middleware service that we have built for use in Java2 Enterprise Edition (J2EE) and message queueing environments. We present the service architecture, the object model, and the internal message queues and MOM transactions implemented. An application example is explored in Section 5. In Section 6, we discuss related work. Finally, in Section 7, we conclude the paper with a summary and a discussion of remaining challenges and future work.

2. Background

OOM is exemplified by object request brokers (ORB) like the OMG's CORBA [14] and by component technology like Sun's Enterprise JavaBeans (EJB) [19]. With OOM, software components that are to be integrated are rendered as distributed objects that offer well-defined interfaces. The standard communication

model between objects is a synchronous client/server model.

MOM is exemplified by message queueing (MQ) systems like IBM's MQSeries [7] and implementations of the Java Message Service (JMS) [20,8]. MOM uses messages as the method of integration. Components create, manipulate, store, and (typically asynchronously) communicate messages, for example using message queues as intermediators. The reliable distribution of the messages in the network is the responsibility of the MOM. The components do not directly interact with each other, but are decoupled and may be anonymous to each other.

OOM and MOM each have their advantages. The object-oriented model of OOM is consistent with the object paradigm commonly followed in modern system development and thus helps to support a consistent development process. OOM advocates the creation of object-oriented interfaces to new and existing applications, and because most new applications are based on object-oriented languages, OOM often is a natural solution. MOM, on the other hand, allows and promotes the decoupling of components in time (i.e., the interacting components do not need to be available at the same time, and thus partial system failures can easily be tolerated), and in space (i.e., the interacting components do not need to know each other), and allows for arbitrary multiplicity of communication partners (1-to-n arity). MOM is generally considered less complex compared to OOM and beneficial whenever a strong coupling of components as required by OOM is problematic.

Consequently, OOM and MOM are often used in combination. For example, message queuing may be used to communicate with diverse legacy backend servers, while distributed objects may be used as web frontends. OOM and MOM together promise to be able to solve a broad set of application integration problems.

2.1. Transaction Processing

An important approach to building reliable middleware-based systems is the use of transactions. A transaction executes a set of actions as one atomic and isolated unit-of-work [2,6]. Transaction processing (TP) technology allows a program to demarcate a transaction and complete its actions in the right order, and provides features for failure handling such as persistent logging and rollback.

OOM support transactions with distributed object transaction services like the CORBA Object Transaction Service (OTS) [15], or its Java binding JTS [21]. Distributed object transactions resemble database transactions; the actions that constitute a transaction are synchronous requests on objects.

MOM also has a notion of a transaction. However, MOM transactions ("unit-of-work" with MQSeries [7,11], or "transacted session" with JMS [20,8]) are different from object transactions. It refers to grouping a set of produced or consumed messages as one atomic unit of work. Either all messages that constitute a MOM transaction are sent out and read, or none of the messages are sent out and read.

Object transactions do not integrate (asynchronous) messages and, conversely, MOM transactions do not include (synchronous) object requests. The only transaction model for combined OOM and MOM environments is the model of "transactional enqueue/ dequeue", where local message enqueue and dequeue operations can be associated with an object transaction (messages may also be enqueued to remote queues, and become visible in the case that the transaction commits successfully) [11,2]. This model builds on the X/Open Distributed Transaction Processing standard [23]. The local message queues essentially function as XA resource managers, comparable to other transactional resources like relational databases.

In [22], we described the shortcomings of this model and identified a number of features that would support better integration. These include, in particular, delivery of messages to *final recipients* beyond intermediary destinations like queues, *remote*, *pre-commit* message delivery during on ongoing transaction, and *transactional dependency* between object requests and message communication. This describes a common demand in modern enterprise systems that use OOM and MOM in combination.

2.2. Motivating Example

Consider the following application example. A website visitor opens a new account, which must then be created in two different distributed legacy systems: an ERP database for accounting, and a CRM database for customer support. The interface to the ERP database is implemented as an Entity EJB, but access to the CRM database must be through a JMS/MQ messaging interface. The challenge is on how to complete the creation of the two accounts in the ERP and the CRM databases in a single distributed transaction.

Unfortunately, there is no support for such transactions available with existing middleware. The model of transactional enqueue/dequeue is insufficient and inappropriate, as it does not allow us to couple remote messages with the object transaction. That is, no single transactional context can be established that comprises the remote request message for creation of the CRM database account, a corresponding reply message of creation success (or failure), and the EJB transaction that creates the ERP database account. Furthermore, no system support for recovery in case of a transaction failure is provided.

We believe that a better programming model for combined transactional synchronous object processing with asynchronous messaging is needed to address such problems. This model should be structured, reliable, and flexible. A distributed object transaction should be able to include asynchronous messages as constituent parts of the transaction. It should be possible to send messages to remote destinations from within an ongoing object transaction, and to receive replies from these destinations. The success of the transaction should be dependent on the success of the messages, and vice versa. In this way, the use of OOM and MOM can be enriched, and limitations on their ad-hoc combination can be overcome.

3. A Global Transaction Context for Objects and Messages

We introduce Dependency-Spheres (D-Spheres) as a new kind of global transaction context that is intended to address the kinds of problems seen in our motivating example. The innovation of Dependency-Spheres is the integration of OOM distributed object transactions and MOM guaranteed messages. This innovation is made with the objective of providing a level of reliability for OOM transactions and MOM messaging in combination that is comparable to that achieved for each kind of middleware separately, at the same time allowing a high degree of flexibility in the ways in which they can be used together. The integration of OOM and MOM occurs through a D-Sphere, which is a new kind of transaction context that logically and operationally groups distributed object transactions, consisting of synchronous object invocations, with guaranteed messaging, consisting of asynchronous messaging operations (Figure 1).

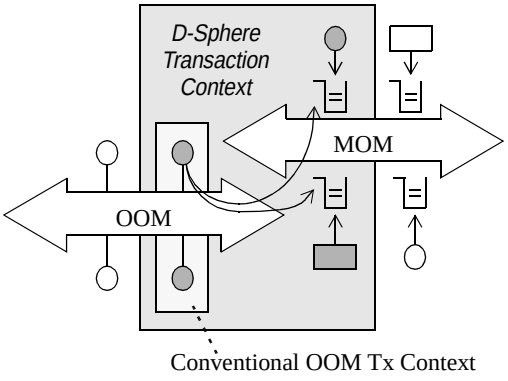

Figure 1. D-Spheres

The effect of integration through a D-Sphere is to make the final outcomes of a transaction and messages mutually dependent. Within a D-Sphere, the synchronous object invocations and the asynchronous messaging operations can be mixed arbitrarily. However, the final success of the transaction and messages in combination depends on the success of the D-Sphere, and the success of the D-Sphere depends on the initial success of the transaction and messages individually. Thus, through the D-Sphere, the transaction can be made to succeed only if the messaging succeeds, and the messaging can be made to succeed only if the transaction succeeds.

The coupling achieved by D-Spheres is desirable when long-running transactions need to include participants that are decoupled in time. The coupling is also desirable to increase parallelism associated with short-running transactions.

In sections 3.1 and 3.2 we discuss the sorts of the transaction and messaging models that can be integrated through D-Spheres, and in section 3.3 we describe the properties attained with the D-Sphere.

3.1. Transaction Model

The purpose of D-Spheres is to enable the integration of existing OOM and MOM. Ideally, D-Spheres would allow existing OOM systems to be used directly. However, this would not support the required level of integration of transactions with messaging. For that reason, D-Spheres must provide an intervening layer of transaction management that enables coordination with messaging. However, to minimize the impact on applications and to allow for the integration of as wide a range of OOM as possible, the constraints imposed by D-Spheres are simple, general, and minimal. Thus, the D-Sphere transaction model is a combination of imposed elements that are independent of the particular transactional OOM being integrated and adopted elements that are dependent on the OOM being integrated.

The imposed elements take the form of a general transaction API that is used in place of the OOM transaction API. D-Spheres use explicit transaction demarcation, including operations such as `begin_dsphere`, `end_dsphere`, and `abort_dsphere`, that parallel typical transaction management operations. This provides familiarity and flexibility to applications and facilitates coordination with messaging operations. It also preserves the transparency of transaction management that is usually provided by OOM. Beneath its transaction API, D-Spheres rely on the underlying OOM to implement management of object transactions. D-Spheres also presume that transactions observe the ACID properties of atomicity, consistency, isolation, and durability. However, D-Spheres rely on the OOM to assure these properties with regard to object transactions. The D-Sphere model is independent of aspects of the OOM transaction model such as optimistic versus pessimistic locking and short versus long transactions.

Within these general requirements, D-Spheres can accommodate a variety of kinds of OOM transaction models. For example, these requirements are satisfied by the typical transaction frameworks and protocols used by existing OOM, such as the OTS [15], the X/Open Distributed Transaction Processing (DTP) architecture [23], or the Enterprise JavaBeans architecture [19]. These systems typically provide support for traditional, short-lived transaction models, such as pessimistic and optimistic ACID transactions [6]; OOM supporting long-lived transactions, such as the Long Running Unit-of-Work (LRUOW) transaction service [1], can be incorporated within D-Spheres as well.

3.2. Messaging Model

Applications use message-oriented middleware to establish reliable, generally asynchronous communication with other applications. Reliability is supported by MOM by means of reliable transport, persistent message storage, and transactional mechanisms with the MOM.

As for the transaction model, our goals for the D-

Spheres messaging model are to impose just the minimum requirements needed to assure effective integration with OOM while preserving the basic level of reliability already provided by MOM. In this way we also hope to accommodate as wide a range of MOM systems as possible.

The main assumption that D-Spheres makes in its messaging model is that it is possible to determine an outcome for a message, that is, the success or failure for significant messaging actions such as the delivery or processing of a message. More specifically, we require the ability to determine the success of message delivery to final recipients and the success of message processing by a set of final recipients. With respect to specific MOM systems, this means that they must support (possibly by means of an additional messaging layer as provided by the D-Sphere middleware service, see Section 4) the addressing and delivery of messages to final recipients (as well as to queues and/or topics) and also provide acknowledgments of deliveries to, or acknowledgments of processing by, these recipients.

The D-Sphere messaging model builds on these basic capabilities to support the definition and evaluation of application-specific conditions for messaging success. For example, applications can define messaging success in terms such as delivery to a minimum number of final recipients or completion of processing by a final recipient by a certain deadline. The D-Sphere middleware, in turn, has responsibility for evaluating these conditions.

The evaluation of message outcome success by means of acknowledgments describes a "worrying-parent-model" for messages: If no acknowledgment by a recipient is received after a predefined scope in time for a message, then that message is declared to have failed, the D-Sphere aborts, and recovery is initiated.

Message outcomes in D-Spheres are treated analogously to the commit or abort outcomes of OOM transactions. This is necessary in order to achieve the interdependence of outcomes between messages and transactions. When a D-Sphere succeeds, then the transaction and messaging operations it contains have been successful, and these are simply allowed to stand. When a D-Sphere fails, though, the transaction is aborted (if it has not already failed) and some additional action must be taken to negate the effects of any messages that have been sent. The D-Sphere can handle the transaction abort (if necessary) simply by instructing the transaction manager to abort the transaction. However, MOM offers no operation analogous to abort for messages. Instead, the D-Sphere must execute some compensating actions. These may include the retracting of messages that have not been delivered to their final recipients, and the sending of compensating messages for messages that have been delivered to their final recipients.

These assumptions made by D-Spheres can be supported for standard MOM like implementations of the Java Message Service JMS [20,8] or IBM's MQSeries messaging middleware [7], if not directly, then by using an additional messaging layer such as provided by the D-Sphere service. As with transactions, a D-Sphere application performs messaging using a D-Sphere messaging API (see Section 4.1). Additionally, D-Spheres offers APIs for defining message conditions and compensating messages. The basic reliability that MOM systems provide for message delivery is preserved, as it is the MOM that still effects the delivery. The additional capabilities that D-Spheres provides, including condition evaluation and compensation, are handled by the D-Sphere middleware.

3.3. Integration Properties Attained

A Dependency-Sphere is a new type of a transaction context inside of which object processing and message exchange takes place. The D-Sphere context is a "global" context in that it contains, or is aware of, other "lower-level" contexts. These lower-level contexts include the conventional object transaction context established by the transaction service used for the transactional client, and the contexts inside of which message recipients may process messages. The D-Sphere context creates and manages the conventional object transaction context, and it establishes a dependency to message recipients' contexts by associating all message recipients of published messages as transaction participants. Message recipients can be both final recipients and intermediary destinations like queues.

As a D-Sphere forms a new kind of transaction context, we can consider how and to what extent it addresses the ACID properties (atomicity, consistency, isolation, and durability) of standard transactions, especially as they may be affected by the integration of two types of middleware.

A D-Sphere is atomic as it either succeeds or fails as a whole. Success is defined as

- all synchronous object requests are completed successfully, and
- all asynchronous message deliveries and message requests for processing are successfully acknowledged.

Failure is defined as

- none of the synchronous object requests has any durable effects and
- for each asynchronous message that has been successfully delivered to or processed by a recipient, a compensating action (for example, the sending of a compensation message) has been initiated.

These definitions of success and failure imply that if any individual synchronous request or asynchronous message that is part of the D-Sphere is not successfully completed, the D-Sphere as a whole fails.

To evaluate the success of a message that is part of a transaction requires the message to be sent prior to the transaction commit, i.e., the message is published with immediate visibility. The immediate visibility of a message is an alternative to on-commit or on-abort visibility, where a message is only visible after the transaction has committed or aborted. Immediate visibility is required for integrating messaging into a transaction, if message success should be able to affect the transaction outcome. Immediate visibility allows computations to be externalized, for example for

purposes of parallelizing transactional work. The externalized computations might later be revoked or compensated [12].

Immediately visible messages are a means to relax the strict isolation properties of conventional ACID transactions. Strict isolation is only required if the non-isolation of an action would lead to conflicts with other transactions. Dependency-Spheres support both strict isolation and relaxed isolation. Transactional object requests are always strictly isolated. Messages are, per default, immediately visible. Messages may, however, also be programmed to observe on-commit or on-abort visibility (in which case they are isolated). Immediately visible messages can be used to model those actions for which isolation is not required or not desired.

The D-Spheres approach retains the notion and model of consistency and durability as established with common OOM transaction services. Consistency and durability must be sustained, which is a shared responsibility of the resource managers involved in the transaction and the application itself [18]. The Dependency-Sphere approach further aids to this process by providing compensation support as a middleware service feature.

4. D-Sphere Service Support

The idea of Dependency-Spheres is supported by a middleware service prototype. The service introduces an additional layer of abstraction above standard transactional OOM and MOM, presenting an integrated view of the base transaction and messaging services used for transactional application development.

The service advances the current state-of-the-art and state-of-the-practice in transactional messaging in that

- messages can be sent to remote queues at any point in time during the ongoing object transaction,
- messages can be associated with application-specific conditions for delivery, such as a particular required set of destinations,
- message delivery is observed to the level of final recipients reading from queues,
- message delivery and processing are evaluated according to the specified message conditions to determine a message outcome, and
- the outcome of messages and the outcome of transactions are interdependent in accordance with the D-Sphere approach.

4.1. Architecture

The D-Sphere service has been implemented in Java using IBM's WebSphere application server [9]. The prototype has been designed for applications running in a J2EE and message queueing environment. D-Sphere clients and servers can use OOM such as EJBs, Java/CORBA, or Java/RMI. MOM supported include JMS providers, or proprietary MOM like IBM's MQSeries. The transaction services supported include JTS and the LRUOW service.

An application (a D-Sphere client) uses the D-Sphere service for transaction management and for conditional messaging. The coupling of the transaction with messages sent as part of the D-Sphere is done by the service. A client does not use the D-Sphere service for invocations on transactional resources, as previously described. Invocations on transactional objects are performed by the client in the standard way of the transactional OOM selected. Figure 2 illustrates the service architecture.

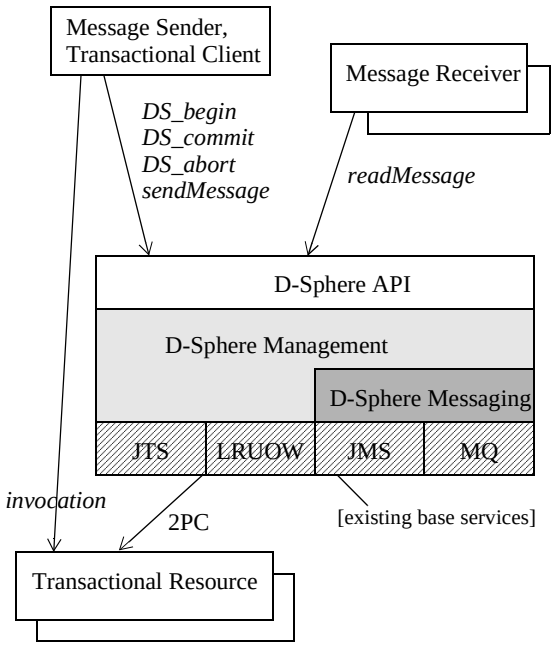

Figure 2. D-Sphere Service Architecture

In addition to the transaction demarcation API (as previously described), the D-Sphere API further comprises a set of messaging operations to send and to read messages. The `sendMessage` operation of the D-Sphere API allows a client to send any data (any Java object) as a message. The operation takes an instance of `java.lang.Object` and an instance of the `Condition` object (the object created using the D-Sphere API for representing message conditions), as described below.

The `readMessage` operation of the D-Sphere API reads a message from a message queue. Any valid queue in the MOM network can be specified.

4.1.1. Object Model

Figure 3 depicts the D-Sphere object model. The core of the model is the `DSObject`, which represents the global context for object transactions and messages. A D-Sphere client creates an instance of the `DSObject` and invokes operations on it through the `DS` interface representing the D-Sphere API. The `Condition` object represents message outcome conditions, which comprise the specification of

message destinations through associated `Destination` objects (objects that specify particular destinations).

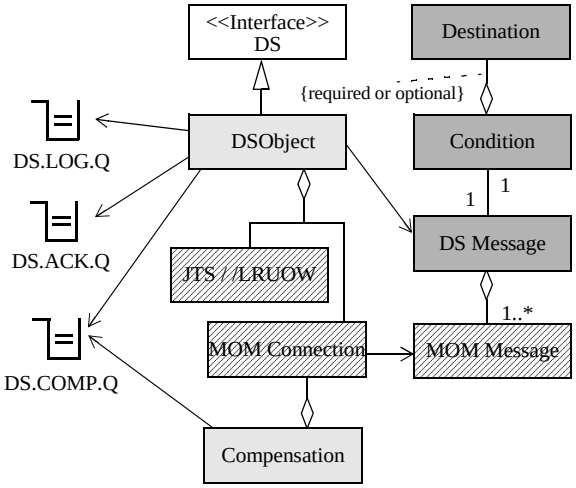

Figure 3. D-Sphere Object Model

The D-Sphere management subsystem implements the transaction and messaging D-Sphere API. It creates and maintains the transaction contexts, keeps a persistent log using queues, and drives the D-Sphere commit protocol that includes the two-phase commit protocol for JTS transactions, or the LRUOW commit protocols for LRUOW transactions. Similarly, it creates and sends all MOM messages for the particular underlying MOM selected (JMS messages or MQ messages). A single D-Sphere message may be mapped to multiple MOM messages. Further, it implements message delivery observation, evaluation of messaging conditions, and management of D-Sphere failures (enforcing transaction rollback and running message compensation).

Issues of the model are elaborated below as follows: Section 4.1.2. describes the internal use of D-Sphere acknowledgments for message observation and message evaluation. Section 4.1.3. presents the D-Sphere commit protocol that uses message evaluation outcomes prior to starting the 2PC protocol for transactional object resources. Finally, compensation is discussed in Section 4.1.4. Section 4.2. then presents the D-Sphere internal MOM queues and MOM transactions implemented.

4.1.2. Acknowledgments

D-Spheres are concerned with message delivery to final recipients who read messages from message queues. The D-Sphere model implements special acknowledgments for readers reading from queues in order to determine numbers or identities of final recipients and different results of message processing. D-Spheres in this way extend the conventional model of message delivery beyond intermediate destinations to include final recipients and their processing.

D-Spheres distinguish two types of internal acknowledgments:

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

The first kind of acknowledgment is generated if a recipient has read a message from a queue unconditionally. I.e., 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 read a message from a queue conditionally, and the conditions for reading were satisfied. I.e., the message read occurred within a recipient's transaction (a MOM transaction or an object transaction that comprises dequeue operations), and the transaction committed. In case that the recipient's transaction has failed, no acknowledgment is generated and the message is put back to the queue by the MOM.

For a failed recipient's transaction, two outcomes regarding the dequeued message are possible. The message can either be put back on the queue, and subsequent attempts to read and process the message can be made. Or, the message can be deleted, preventing subsequent attempts to be made. The first behavior is desired in case that the transaction failure resulted from a system failure; for example, a server crash. The second behavior is desired in case that the transaction failure resulted from an application error; for example, the failed creation of a database account due to an invalid account number.

With the current version of the D-Sphere service, a message is always put back to the queue from which it was dequeued in case of a recipient's transaction failure. In accordance with the D-Sphere acknowledgment model, no acknowledgment of successful processing will be created. Therefore, the sender side D-Sphere will eventually fail and recovery and compensation be initiated.

To prevent re-attempts of processing of the successfully delivered, but not successfully processed message, a compensation message should be sent to the recipient's queue (see Compensation below). The compensation message and the original primary message will then cancel each other out automatically. We have implemented this behavior for the D-Sphere `readMessage()` operation.

4.1.3. Commit Protocol

Conditional messaging requires that the delivery of messages to destinations be monitored and that message conditions be evaluated so that the success or failure of message delivery and processing can be determined.

The evaluation of message conditions can be started immediately after the message has been sent out, and must ultimately be terminated at commit-time or when the D-Sphere times out. The D-Sphere approach of

conditional messaging introduces an extension to the conventional two-phase-commit (2PC) protocol for transactions [6] in that the "voting" phase for individual conditional messages is performed concurrently to the ongoing transaction and thus prior to the voting phase of the 2PC for the transactional resources involved.

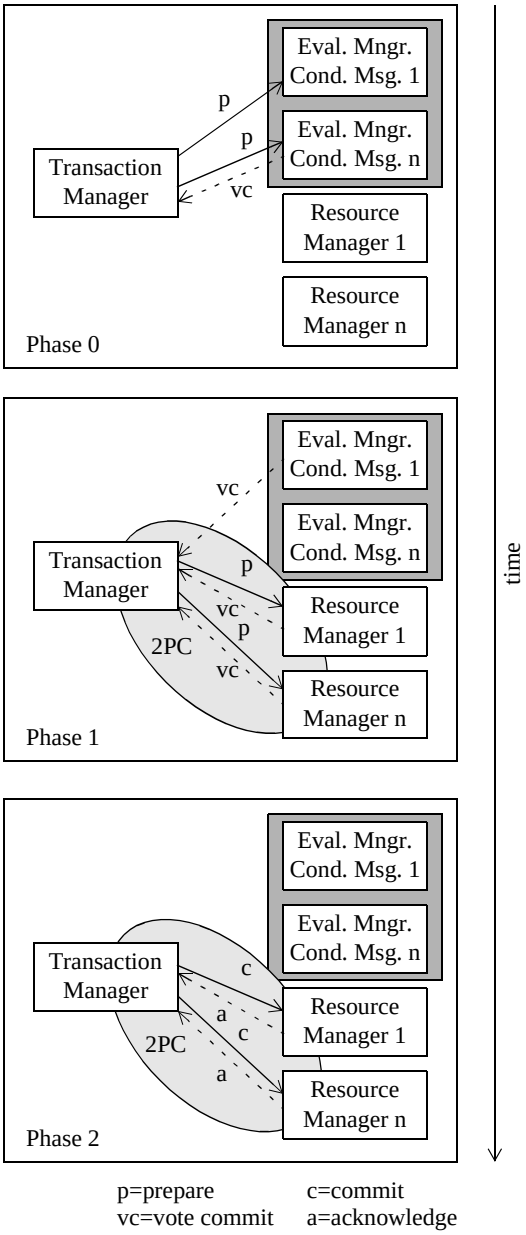

Figure 4. D-Sphere Commit Protocol

Figure 4 illustrates this protocol. The evaluation for each conditional message is started during the ongoing transaction (Phase 0) after the message is sent out. During Phase 0, a "vote commit" may be determined for those messages for which acknowledgments (and results of processing) were received. In case that an unsuccessful conditional message evaluation occurred in Phase 0, the conventional 2PC will not be performed at all, but an abort be enforced. A message failure corresponds to marking the transaction as rollback only.

Other pending evaluation results may be determined shortly before Phase 1, the voting phase of the conventional 2PC, begins. The prepare/vote-commit requests to the transactional object resources will only be driven if all message evaluations were successful. Correspondingly, Phase 2 (commit) of the 2PC will only performed if in addition all resource managers voted commit.

4.1.4. Compensation

In the event that a D-Sphere fails, recovery for its messages must be performed. The sending of its messages can be stopped (if it was not completed), sent messages can be retracted from intermediate destinations like queues (if feasible), and compensating messages can be sent to destinations that have received the message.

Compensation introduces some special issues. So long as the only recipients considered are known intermediate destinations, then compensating messages can be sent directly to those destinations. A compensating message may, however, alternatively be delivered to the final recipients of the primary message or to some other application or system-defined destination. If final recipients are to be included in compensation, then these must also be known (for example, through registration of final recipients or read acknowledgments that encode the final recipients' identities). Otherwise, compensating messages sent to intermediate destinations may suffice for anonymous final recipients. This approach offers some flexibility to applications, though, in that it allows for the handling of compensation by processes other than those that received the original message. Furthermore, it allows a compensating message to cancel out (to delete) the original message, in case that both messages exist in parallel in the same queue (see above, Section 4.1.3).

Because compensation is important to the integrity of Dependency-Spheres, the delivery of compensating messages is an important concern. Therefore, compensating messages must be sent by a reliable means, and the delivery of compensating messages must be monitored. MOM generally provide support for guaranteed message delivery. However, the successful delivery of compensating messages to final recipients cannot be absolutely guaranteed. Therefore, the D-Sphere system provides as an additional means compensation logging to record the state of compensation delivery for possible handling outside of the system (see below, Section 4.2).

4.2. Queues and Internal MOM Transactions

Three message queues are fundamental to the D-Sphere service: the `DS.LOG.Q`., the `DS.ACK.Q`, and the `DS.COMP.Q` (see Figure 3). These queues are used internally by the D-Sphere service for purposes of

implementing persistent logging, message monitoring, message condition evaluation, and compensation. These queues need to reside with the MOM queue manager that a sender chooses for running his D-Sphere; these queues need not be set up with the MOM queue manager that a message reader uses.

The `DS.LOG.Q` is the queue used to store log entries, including the actual destinations to which MOM messages implementing D-Sphere messages have been sent.

The `DS.ACK.Q` is the queue to which internal acknowledgment messages from final recipients are sent. These acknowledgment messages are created automatically by the `DSObject` on the reader side when a D-Sphere client in role of a message reader successfully reads a message from a queue using the `readMessage`() operation of the D-Sphere API. The `DSObject` on the sender side uses the `DS.ACK.Q` for message condition evaluation.

The `DS.COMP.Q` contains compensation information and actual compensation messages needed in case that the D-Sphere aborts or fails and compensation needs to be run. Compensation messages are created by the sender application using the `DSObject` at the same time that the primary original messages are created. Compensation messages are specially marked messages that correlate to the primary message sent. The `DSObject` then stores the compensation messages in the `DS.COMP.Q`. The independent D-Sphere compensation process, if and once started, forwards these messages to all recipients that need to receive the compensation message.

A D-Sphere client may in addition have any number of additional application queues for incoming (reply) messages.

The D-Sphere service employs internal MOM transactions to work with the three D-Sphere queues. Figure 5 depicts an example of a message send. An application sends a D-Sphere message that has two remote destinations `Q1` and `Q2` using the D-Sphere service running on Machine 0. The MOM queue manager on Machine 0 consequently maintains the three aforementioned queues. The sending of the D-Sphere message maps to two separate MOM messages for each destination queue, multiple log entries to the `DS.LOG.Q` queue, and the creation of a compensation message put to the `DS.COMP.Q` queue. All messages together form one atomic MOM unit-of-work (MOM Tx 0).

Figure 6 depicts the corresponding MOM transactions created for reading a message. Two separate clients in role of message receivers use the D-Sphere service running on two different machines. The messages are received from either application queue `Q1` or `Q2`, and, if successful, the recipient's D-Sphere object will create the respective acknowledgment messages encoding information about the final recipient (for example, his identity), the message read (for example, the time the message was read), and/or the message processing (the time the transaction during which the message was read and processed committed). This acknowledgment message is sent to the sender

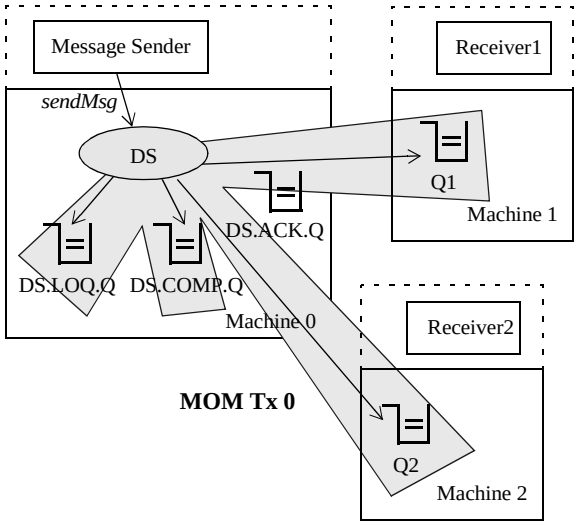

Figure 5. D-Sphere Message Send

side `DS.ACK.Q` queue; the information about the host and the MOM queue manager that the sender used was encoded with the received MOM message in `Q1` or `Q2`, as created by the `DSObject` of the sender. The acknowledgment message in turn contains the information about each reader's MOM queue manager. The message consumption from the queue Q1 or Q2, and the sending of the acknowledgement message for receiver 1 or receiver 2, respectively, each form an atomic MOM unit-of-work.

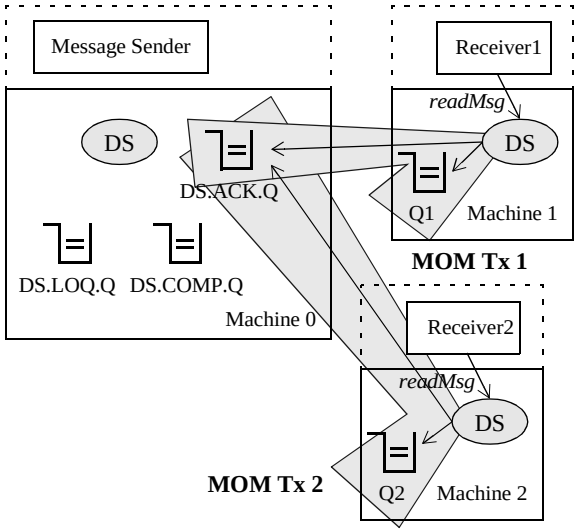

Figure 6. D-Sphere Message Receive

5. Motivating Example Revisited

Consider again the motivating example given in Section 2.2. Two accounts in two different distributed databases should be created in one single atomic unit-of-work, with one database offering an EJB interface, and the other database offering an asynchronous JMS/ MQ interface only.

5.1. Non D-Spheres Solutions

Without D-Spheres, there are basically two general approaches. One is to define a series of transactions with corresponding compensating transactions, where each transaction consists of the individual database update plus a enqueue or dequeue operation. Figure 7 illustrates this saga-like [5] solution using transactional enqueue/dequeue steps [11,2].

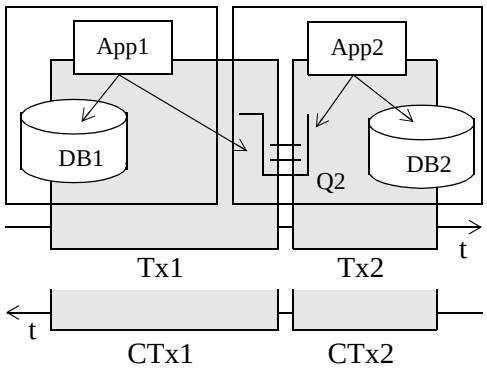

Figure 7. Saga-Like Solution

Application 1 (App1) is the first transactional client who creates an account in the EJB database (DB1) and, if successful, sends out a distributed message request to queue Q2. This is the first transaction (Tx1). Application 2 (App2) is the second transactional client who dequeues the message arrived and creates an account in database 2 (DB2). This is the second transaction (Tx2). If Tx2 fails for any reason, recovery for Tx2 must be initiated, and, in addition, a compensating transaction CTx1 for Tx1 must be run. This can be, in effect, a complex exercise due to technical constraints of existing middleware functionality (e.g., remote reading from queues cannot be transactionally coupled with a local database update).

With the saga-like solution sketched above, you worry not only about the business logic of the initial and compensating transactions, but also about complex transaction management. There is no middleware service support available that readily aids in this process.

The second non D-Spheres solution would be to hand-code the intended transaction as one transaction within the EJB. That is, the EJB that creates an account in DB1 would send and receive different messages to Q2 to request the account creation in DB2 and to react upon its outcome. Asynchronous messaging can be performed by EJBs (or other Java clients) using the Java Message Service (JMS). However, the JMS messages are independent of the EJB's transaction context. Therefore, all transaction dependencies and transaction management needs to be hand-coded as well. There is no middleware service support to aid in the definition, management, and recovery for failure of such a transaction.

5.2. D-Spheres Solution

The D-Spheres solution offers a structured approach to the definition and management of object transactions that comprise messages. It is a straightforward and more reliable solution to the problem described, as issues of transaction management are already provided by the middleware. The application can focus on the business logic of transactions and messages (including both original and compensating messages).

With D-Spheres (Figure 8), the client application (App1) demarcates a D-Sphere transaction using `dsphere_begin()` and `dsphere_commit()`. The D-Sphere comprises the sending of a message to Q2 for account creation in DB2 (marked with the condition that successful processing is required), and the invocation on the EJB to create an account in DB1. If any failure of either the EJB account creation transaction, or of message delivery or message processing, occurs, the D-Sphere service automatically performs recovery and compensation as defined in Section 4.

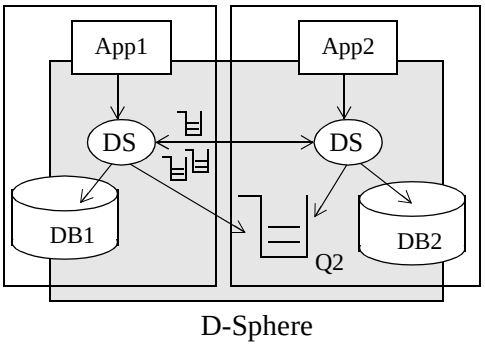

Figure 8. D-Sphere Solution

6. Related Work

In our previous work [22], we defined a messaging taxonomy and proposed the general idea of message delivery and message processing transactions as advanced integration models for distributed object transactions and messaging. The work included a comparison and discussion of these models with the basic integration model of transactional enqueue/ dequeue using the taxonomy. The specific model of

Dependency-Spheres as first presented in this paper instantiates and specializes the general idea of message delivery and message processing transactions with a concrete design and implementation.

Other work related to Dependency-Spheres can be roughly classified into three categories: advanced transaction models, workflow systems, and integrated middleware services.

In general, an advanced transaction model is any transaction model other than a flat, ACID transaction. Nearly all such advanced models relax the ACID properties in some way to achieve an advantage: to support long-running computations, to improve throughput, etc. Workflow systems have been proposed as a vehicle to support such advanced transaction models [10], and vice-versa.

Like Dependency-Spheres, many of these advanced models rely on the mechanism of compensation to provide transaction atomicity. For example, the saga model [5] relaxes the isolation property by allowing sub-transactions to commit prior to the completion of the saga. If a sub-transaction fails, the saga is "undone" by automatically executing the compensating transactions associated with successfully executed sub-transactions. Thus, either the entire saga executes, or its effects are semantically reversed. The Multi-Level Transaction model [6] and the ConTract model [16] use compensating transactions in a similar way. Applications that employ these models must program transactions and compensating transactions in a manner that ensures application consistency.

Dependency-Spheres suggests a model similar to these compensation-based transaction models, and we believe that our work can benefit from the research in this area (especially in the area of transaction consistency). Unlike these models, however, Dependency-Spheres promotes an evolutionary approach motivated by the need to integrate applications using existing middleware technologies. We believe that our approach, which is compatible with existing middleware standards and which supports a familiar programming model, is better aligned with the demands of enterprise-scale application development.

To our knowledge, the DAOS project and its X^2TS prototype [12] is the only effort relating directly to the approach of Dependency-Spheres. The X^2TS compares to Dependency-Spheres in that it is also an integrated middleware service. The X^2TS addresses the integration of CORBA OTS transactions with CORBA event notifications, and suggests different coupling modes based on concepts of distributed active object systems. A coupling mode defines, for instance, whether a transaction context is shared, whether forward or backward commit dependencies between multiple contexts contained in the overall context are supported, and so on. X^2TS observes and supports a subset of the various different coupling modes feasible, as do D-Spheres. A detailed description of coupling modes and a detailed comparison between the two approaches can be found in [13].

7. Summary and Discussion

Many of today's enterprise systems require the use of different middleware and consequently employ both OOM and MOM. While each middleware has its strengths, they have been designed to work in isolation and therefore support for the interoperation between OOM software components and MOM software components is very limited or non-existing. This is especially true for transaction processing across distributed objects and messaging components. Standard object transactions consist of synchronous requests on distributed objects only, but do not integrate asynchronous message deliveries. This introduces a limitation to enterprise systems that use both OOM and MOM. A common demand in enterprise systems is thus to integrate object transactions and messaging.

In this paper, we introduced the concept and middleware service of Dependency-Spheres as a novel, structured and reliable approach to integrating distributed object transaction processing and messaging in standard OOM and MOM environments.

D-Spheres support both standard short-lived transactions through services like the OTS, as well as long-lived transactions through the LRUOW transaction service. The D-Sphere messaging model allows associating conditions of various kinds to messages in order to define messaging failure or messaging success on a per-message basis (if desired).

We presented the D-Sphere integrated middleware service, which implements the context using a combination of an object-oriented architecture with persistent queues and MOM messaging transactions.

D-Spheres advance the state-of-the-art in transaction processing with messaging as well as middleware integration. With D-Spheres, it is possible to

- define message outcome conditions (i.e., criteria for message success or message failure), including criteria relating to multiple, final recipients,
- associate these conditions with messages on a per-message basis,
- send messages to remote queues at any point in time during the course of an object transaction,
- have the message delivery be performed and observed according to the specified conditions,
- have messaging outcome be evaluated, and respective consequences of action for transaction success or transaction failure and message compensation automatically be taken.

D-Spheres uniquely make an object transaction and conditional messages dependent on each other.

As argued in [3], specific software architectural choices may be constrained by the middleware used for system implementation, or, conversely, middleware often influences the software architecture of a system. Dependency-Spheres overcome one such limitation by not constraining a (workflow, business process, or system) transaction that comprises synchronous and asynchronous requests to be mapped to a pure synchronous distributed object transaction. Rather, D-Spheres support both synchronous requests and asynchronous messages. And conversely, D-Spheres open up a whole range of new possibilities for

transactional processes to be designed in a software architecture.

Future Work. We are investigating a variety of extensions of the D-Sphere idea and the D-Sphere prototype. These include additional support for different messaging models, including content-based publish/subscribe, or, extending the "flat" D-Sphere model to advanced D-Sphere models of D-Sphere sagas, nested D-Spheres, and other. We further believe that more research is needed to better understand and differentiate the possible dependencies of a sender side transaction with message deliveries only, different kinds of (transaction or non-transactional) message processing on different receiver sides, and of combinations thereof.

8. Acknowledgments

We would like to thank Ignacio Silva-Lepe for his valuable comments on the D-Sphere project, Peri Tarr for her comments on a previous version of this paper, and Christoph Liebig for sharing his thoughts and expertise on advanced transactions.

9. References

[1] B. Bennett, B. Hahm, A. Leff, T. Mikalsen, K. Rasmus, J. Rayfield, and I. Rouvellou, "A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes". *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, New York, USA, 2000, Springer-Verlag LNCS 1795, pp. 331-348.

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

[3] E. Di Nitto and D. Rosenblum, "Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructure", *21st International Conference on Software Engineering (ICSE 99)*, Los Angeles, CA, 1999, ACM Press, pp. 13-22.

[4] W. Emmerich, *Engineering Distributed Objects*, Wiley, 2000.

[5] H. Garcia-Molina and K. Salem, "Sagas", *SIGMOD International Conference on Management of Data*, ACM, May 1987, pp. 249-259.

[6] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Francisco, 1993

[7] IBM, *MQSeries Application Programming Guide, 10th Ed*. IBM, 1999.

[8] IBM, *MQSeries Using Java, 5th Ed*. IBM, 2000.

[9] IBM, WebSphere software platform. http://ibm.com/websphere

[10] F. Leymann and D. Roller, *Production Workflow: Concepts and Techniques*, Prentice Hall, Upper Saddle River, NJ, 2000.

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

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

[13] C. Liebig and S. Tai, *"Middleware-Mediated Transactions"*, 3rd International Symposium on Distributed Objects & Applications (DOA 2001), Rome, Italy, IEEE Press, September 2001.

[14] OMG, *The Common Object Request Broker: Architecture and Specification, rev. 2.2*., OMG, 1998.

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

[16] A. Reuter, K. Schneider, and F. Schwenkreis, "ConTracts Revisited", *Advanced Transaction Models and Architectures*, S. Jajodia and L. Kerchberg (eds), Kluwer Academic Publishers, Boston, 1997, pp. 127-151.

[17] W. Ruh, F. Maginnis, W. Brown, *Enterprise Application Integration: A Wiley Tech Brief*, Wiley, 2001.

[18] D. Slama, J. Garbis, P. Russel. *Enterprise CORBA*. Prentice-Hall, 1999.

[19] Sun Microsystems, *Enterprise JavaBeans Specification v1.1*, Sun, 1999.

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

[21] Sun Microsystems, Java Transaction API (JTA) and Java Transaction Service (JTS). http://java.sun.com/j2ee/transactions.html

[22] S. Tai and I. Rouvellou, "Strategies for Integrating Messaging and Distributed Object Transactions", *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)*, New York, USA, 2000. Springer-Verlag LNCS 1795, pp. 308-330.

[23] X/Open, *Distributed Transaction Processing: Reference Model, version 3*, X/Open, 1996.