Strategies for Integrating Messaging and Distributed Object Transactions

Stefan Tai and Isabelle Rouvellou

IBM T.J. Watson Research Center, New York, USA
{stai,rouvello}@us.ibm.com

Abstract. Messaging, and distributed transactions, describe two important models for building enterprise software systems. Distributed object middleware aims to support both models by providing messaging and transaction services. But while the concept of distributed object transactions is well-understood, support for messaging in distributed object environments is still in its early stages, and not nearly as readily perceived. Integrating messaging into distributed object environments, and in particular with distributed object transactions, describes a novel and complex software design problem. This paper details this problem, presenting first results from our project of developing a messaging and transaction integration facility. The first contribution of this paper is a comprehensive messaging classification framework, which defines messaging concepts and terminology, and enables us to compare different messaging architectures. Second, we analyze sample messaging middleware using this framework, and identify the architectural messaging styles that they induce. Third, we derive four different strategies for integrating messaging and distributed object transactions. We discuss each of these integration strategies, and outline the open research issues that need to be solved. Overall, this paper advances our understanding of the motivation for, the problems of, the current state-of-the-art in, and future models for integrating messaging and distributed object transactions.

1 Introduction

Messaging, and distributed transactions, are among the most demanded and important models that distributed object middleware is required to support.

Transactions, as known from database systems and transaction processing monitors [1] [5], guarantee that a set of operations transforms the shared state of a system from one consistent state to another consistent state. Standards like the CORBA Object Transaction Service OTS [13] address transaction processing in distributed object environments, and with CORBA Object Transaction Monitors [15], or component technologies like Enterprise Java Beans [11], the middleware for building transactional distributed object systems is available today. Distributed object transactions are considered essential for the development of industrial-scale n-tier distributed object systems, where some server layers manage a persistent store [16].

J. Sventek and G. Coulson (Eds.): Middleware 2000, LNCS 1795, pp. 308–330, 2000.

Distributed object messaging, on the other hand, is not nearly as well-understood and readily perceived as distributed object transactions. There exists no common notion of messaging, and support for messaging in distributed object environments is more in its experimental, than adopted stage. In general, messaging refers to the communication model of asynchronous, possibly multicast message exchange for event notification, or request processing. In addition, messaging refers to the system development paradigm associated with messaging middleware like message queueing (MQ) systems [2] [8]. This paradigm is based on the strong decoupling of clients and servers using message queues as intermediators. Messaging is considered beneficial to enhancing system reliability, stability, and flexibility for system evolution [2].

Distributed object middleware that supports both distributed transaction processing, and messaging, promises to allow for addressing a large problem domain of software systems. Consequently, messaging service specifications, including CORBA Messaging [14], or the Java Messaging Service (JMS) [17], have recently been proposed. These messaging services are intended to be used in addition to the existing and already employed distributed object transaction services, and are different from existing event notification services like CORBA Events [13], or CORBA Notification [13].

The emergence, respective availability, of distributed object messaging and transaction services now inaugurates the question, and potential, of *distributed objects integrating messaging and transactions*. While middleware services in general attempt to mimic the Bauhaus principle of clear separation of functionality and concern to enable service integration, the integration of messaging and distributed transactions describes a more fundamental, complex problem. We need to better understand messaging and different models of messaging in order to approach integration with distributed transactions, and we need to identify the different objectives of integration, and how these objectives can be achieved in an efficient way.

We have begun to design a new middleware facility for integrating messaging and distributed transactions in a distributed object environment. We aim to integrate two advanced distributed object messaging and transaction services, which are currently being developed in two related projects. In this paper, we present the results from our first step in this design process, exploring the problem of integrating messaging and distributed transactions in detail.

The paper is structured as follows. In Section 2, we present a novel messaging classification framework, which defines messaging concepts and terminology. The framework establishes a common language to communicate about messaging, and enables us to compare different messaging architectures. In Section 3, we analyze sample messaging middleware using this framework, identifying the architectural messaging styles that they induce. Though we will eventually focus on a particular middleware and service environment for our integration facility, the objective in our first step has been to study messaging across different messaging middleware. In Section 4, we present four strategies for integrating messaging and distributed object transactions. We discuss each of these inte-

gration strategies, and outline the open research issues that need to be solved. Section 5 concludes with a summary of the work presented, and discusses our plans for future work.

2 Messaging Classification Framework

This section presents the *messaging classification framework* that is central to understanding and discussing different strategies for the integration of messaging and distributed transactions.

The framework is organized around three models: *message delivery model*, *message processing model*, and *message failure model*. The message delivery model defines fundamental properties of message delivery. The message processing model extends the delivery model with properties additional to message processing. The message failure model defines properties of message failure, with respect to message delivery or message processing.

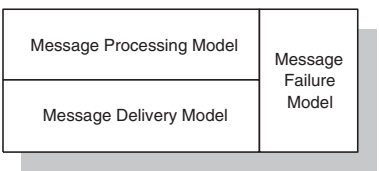

Fig. 1. Messaging Classification Framework

The framework is quite comprehensive, and deliberately addresses a variety of messaging aspects. Each model defines a number of messaging properties, and for each model property, we describe different possible values. Combinations of such values characterize a messaging architecture. Not all combinations of property values can be implemented using currently existing messaging middleware, and some combinations may not be feasible at all.

The properties that we define in each model represent those aspects of messaging that we find important to capture differences between messaging models and understandings. We have experienced practitioners and researchers to refer to messaging as "programs communicating by putting messages on queues", as "multicast event notification", or as "processing requests asynchronously". The classification framework aims to establish a common language to better communicate about messaging. We selected the properties based on communication with colleagues, including [9].

2.1 Message Delivery Model

The *message delivery model* defines the properties related to exchanging (sending and receiving) messages between message producers and message consumers.

Message delivery is not concerned with the processing of messages, i.e., the effects resulting from message exchange. Message delivery essentially is about *event notification*: ways to inform interested consumers about the occurence of some state transition.

The message delivery model comprises the following properties:

Representation defines how a message is represented in the system. A message may be represented as an *object*, or as a *data* element (possibly following a standard message format of message header and message body). Messages that are represented as either objects or data can be passed as parameters to operations for message exchange. In lieue, a message may have no representation as an entity, but corresponds to an (asynchronous) *operation invocation* on an object only.

Messaging API defines whether application-independent, or application-specific operations (or, the combination of both) are used for messaging. An *application-independent* messaging API describes generic operations for sending, receiving, or administering messages. An *application-specific* messaging API, on the other hand, is an interface of application functionality defined by the application developer.

Message representation, and messaging API, are the two most fundamental aspects that distinguish different messaging middleware.

Table 1. Message Delivery Model (1)

1. Representation	2. Messaging API
1. as object	1. application-independent
2. as data	2. application-specific
3. operation invocation	3. combination of 1. and 2.

The property of **initiation** defines who causes a message delivery to happen. The delivery can either be initiated by a producer who sends a message (*push*), or by a consumer who queries for a message (*pull*). *Mixed initiation* refers to the case where the same message is both pushed and pulled by different consumers. Pull and mixed initiation require that a message is represented as an entity.

Intermediation defines whether or not *intermediators* such as message queues or channel objects are part of the message exchange. There may be none, exactly one, or a set of intermediators involved for a single message delivery. Intermediators are used when messages are represented as own entities in the system, and they are essential to realize a messaging model on top of a synchronous object invocation model.

The **multiplicity of ultimate recipients** defines the number of the final (not intermediate) consumers of a message sent. If there is only one such ultimate recipient, the message delivery is *unicast*, or point-to-point. The message delivery

is *multicast*, if there are multiple ultimate recipients. *Broadcast* is a special case of multicast delivery, where the message is sent to all consumers on the network.

The **anonimity of ultimate recipients** defines whether the final consumers are all known, partly known, or all unknown to the message sender. The consumers typically are known to the messaging middleware, or to an intermediator that is part of the message exchange.

The property of **subscription** defines whether consumers have declared their interest in receiving messages by subscribing according to some subscription mechanism, or whether no subscription was necessary for message exchange. Consumers may subscribe to intermediators, or to message producers directly. Subscription commonly is required for push event notification, and may address all messages that a producer or intermediator publishes, or only selected messages, for example based on message types.

Synchronicity defines the model of synchronization that is used for message delivery between a sender and its *direct* consumers (intermediators, or ultimate recipients, if no intermediator exists).

Messaging in general implements an *asynchronous* communication model between a message producer and its *ultimate* recipients. However, as illustrated in Figure 2, this asynchronicity may be implemented using synchronous communication via an intermediator. Alternatively, direct asynchronous calls (e.g., CORBA IDL oneways [12], or CORBA AMI and TII [14]) may be used.

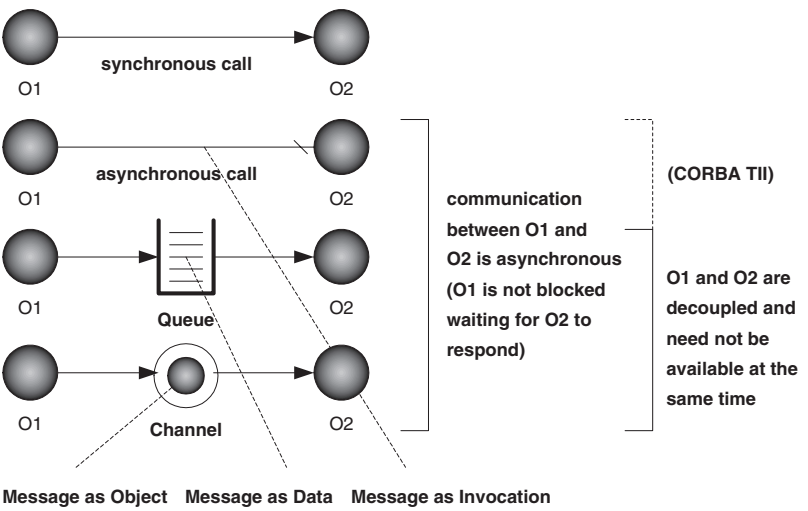

Fig. 2. Event Notification

The property of **delivery guarantee** defines the level of guarantee that is assured for message delivery. The *best-effort* delivery semantics is the lowest level of guarantee, and essentially describes no guarantee. With *at-most-once*,

we refer to the semantics of basic delivery guarantee, and with *exactly-once*, to the highest level of delivery guarantee. The exactly-once delivery guarantee may be implicit (is transparently assured by the middleware), or may be explicit (is visible in the form of acknowledgments to the sender).

Table 2. Message Delivery Model (2)

3. Initiation	4. Intermediation	5. Multiplicity	6. Anonimity
1. by producer	1. none	1. unicast	1. known set
2. by consumer	2. exactly one	2. multicast	2. unkown set
3. push and pull	3. multiple	3. broadcast	3. mixed

7. Subscription	8. Synchronicity	9. Delivery Guarantee
1. subscription	1. synchronous	1. best effort
2. no subscription	2. asynchronous	2. at-most-once
3. mixed		3. implicit/explicit exactly-once
		4. either 2. or 3.

In addition, there are also the following properties of message delivery that are typically associated with intermediation ("queue-management"):

Persistence defines whether a message is persistent, or transient. *Persistent* messages are messages that are copied to a persistent store (of the intermediator), in order for the messages to survive failures such as system crashes. *Transient* messages, on the other hand, are kept temporarily, based on message birth and expiry times.

The **ordering** property defines the sequence in which a message is delivered w.r.t. other messages. Intermediators may implement a temporal-based ordering (*first-in-first-out (FIFO)*, or *last-in-first-out (LIFO)*), a *random* ordering, or a *priority-based* message delivery. Message priorities may include, for example, the specification of an allowed time window of start time and end time inside of which the message must be delivered.

Filtering defines whether a message is subject to a selection mechanism by intermediators in order to be further distributed to consumers. Filters may be defined for all kinds of selection, for example based on message timestamps, message size, or message content. We subsume these under filtering based on message *headers* (message properties), and filtering based on message *bodies* (message content).

Filters allow a single intermediator to handle various messages for a set of consumers with different interests. Filtering typically involves a prior subscription of consumers to the intermediator.

Table 3. Message Delivery Model (3)

10. Persistence	11. Ordering	12. Filtering
1. persistent	1. FIFO	1. none
2. transient	2. LIFO	2. header/type
3. either 1. or 2.	3. random	3. body/content
	4. priority-based	4. 2. and 3.

2.2 Message Processing Model

The *message processing model* defines the properties related to communicating back the results that are consequences of a message delivery. Message processing goes one step further than event notification, as it is essentially about *asynchronous request processing*: ways to asynchronously request results from a remote server.

There are two major roles that software components play for asynchronous communication: the role of a *client*, i.e., a sender of a request message and a consumer of a reply message, and the role of a *processing server*, i.e., an ultimate consumer of the request message and a sender of the reply message.

The message processing model defines the properties that characterize a *processing result*, and how the result is *communicated* from a processing server to the requesting client. These properties are in addition to those captured in the message delivery model.

Processing result defines whether the result of a message sent is a *single return value*, a *single integrated return value*, or a *set of individual return values*. The latter two address multicast or broadcast messages, where multiple recipients process the same message, and multiple acknowledgments of processing and/or processing results have to be communicated back. An intermediator is needed in order to integrate multiple returns.

Communication defines how the client receives the processing result. The client may receive a *separate reply message* to the request message, for which a message correlation mechanism (for example, using message ids) to associate the two messages as one request/reply message pair is needed. Two other, common patterns for returning processing results are the callback approach and the polling approach. With the *callback* approach, the client passes a callback object reference with the request, which in turn is invoked by the server when the results are ready. With the *polling* approach, a poller object is returned by the request, which the client can query for results.

In all three of these cases, the client can continue its processing independent of the request sent, but still expects a result of processing to be communicated back to him. Using the polling approach, the client may become blocked until a response is available to the poller. This model is also referred to as the *deferred synchronous* model.

Table 4. Message Processing Model

13. Processing Results	14. Communication
1. single return value	1. separate reply message
2. single integrated return value	2. callback
3. set of individual return values	3. polling

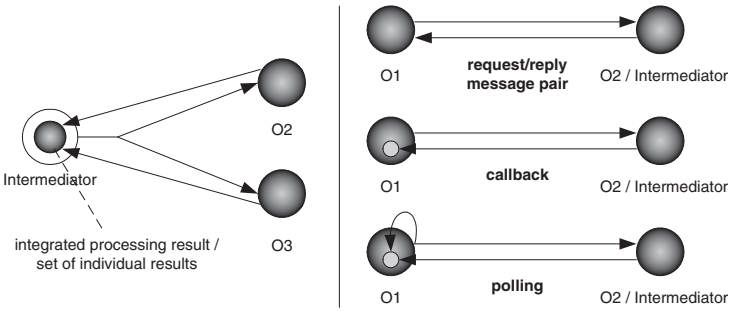

Fig. 3. Asynchronous Request Processing

2.3 Message Failure Model

The *message failure model* defines the properties related to message failures. The message failure model is of predominant importance for integrating messaging and distributed transactions, as any failure of a message affects the *acidity* ("all-or-nothing") property of transactions. But what constitutes a message failure (respective success), and how is the failure detected?

The **failure level** defines whether a message success is based on its successful *delivery only*, or whether a message success is based on the successful delivery and the successful *processing* of the message as well. For example, a `debit(account, value)` message can be considered successful if the message is successfully delivered, or, if the message is successfully delivered *and* the actual debit for the specified account is successful, too.

With **failure scope**, we distinguish whether the success of a message is based on *a defined set of particular, ultimate recipients* (for example, one specific recipient, or a defined list of recipients), *a defined number or range of any recipients* (for example, exactly one, at least one, or 2-5 recipients), or *all ultimate recipients*. The failure scope thus describes how the messaging property of multiplicity relates to message failure definition.

The property of **failure detection** finally defines how a messaging failure is discovered. *Acknowledgments* of message reception are necessary for the message delivery failure level, and actual *results of processing* are necessary for the message processing failure level. *System* and *user-defined exceptions*, a *timeout* for

the acknowledgments/replies that is specified and monitored, or the combination of both may be used for failure detection.

Table 5. Message Failure Model

F1. Failure Level	F2. Failure Scope	F3. Failure Detection
1. delivery	1. particular recipients	1. exceptions
2. processing	2. number of any recipients	2. timeout of ack/reply
	3. all recipients	3. combination of 1. and 2.

3 Sample Messaging Middleware and Architectures

In this section, we analyze selected examples of messaging middleware: *CORBA Messaging*, *CORBA Events* and *CORBA Notification*, *Java Messaging*, and *Message Queueing (MQ)* systems.

The purpose of this section is not to evaluate each middleware regarding its strengths and weaknesses, but to identify the *messaging architectural styles* that they induce, and the commonalities and differences that exist between the different styles. The notion of an *architectural style* has been introduced in software architecture research to describe the communication and cooperation, and composition and design rules that a set of software systems shares. A middleware has an impact on the architecture of a system that is implemented on top of it, and implicitly defines an architectural style, or sub-style. [3] discuss event-notification styles as defined by event-based middleware. Our work focuses on messaging middleware for use in distributed object environments, and uses our more comprehensive classification framework for this purpose.

We aim to understand the different notions of messaging that are suggested by these middleware. Therefore, we focus on the messaging programming models that the technologies offer to developers, but we are not concerned with internal details of messaging middleware realization. The messaging classification framework introduced in the previous section allows us to describe each technology very briefly, by repeatedly looking at the same common messaging properties as defined in the framework.

3.1 CORBA Messaging

CORBA Messaging refers to the messaging style based on *asynchronous method invocations (AMI)* as proposed with the OMG CORBA Messaging service [14].

A message corresponds to an application-specific AMI on an object, which can either be a callback- or a poller-based request. Callback-based AMIs work on behalf of `ReplyHandler` CORBA object references, and poller-based AMIs

on `Poller` objects that are instances of a CORBA value type. AMIs introduce a truly asynchronous invocation model in addition to the standard synchronous CORBA invocation model[1].

With CORBA Messaging, an application-specific, standard CORBA IDL server interface is mapped to an AMI "implied-IDL" interface, a client-side view of the interface containing either callback or poller-based operation signatures. In this way, servers need not be modified to serve asynchronous requests. Each message (AMI) is client-initiated (push-model), and no intermediator is used. Thus, the current CORBA messaging model does not support multicast, but is unicast, and targeted (the server is nonimous).

CORBA AMIs are asynchronous calls with at-most-once delivery semantics. For target servers that are not active (or, activatable) at the time the request is issued, CORBA Messaging introduces the notion of a *time-independent invocation (TII)* as a special kind of AMI. TIIs can outlive client and server process lifetimes (can be persistent, and have exactly-once delivery semantics). For both AMIs and TIIs, priorities can be specified, otherwise the message delivery is temporal, based on the time that the request is issued (FIFO). There is no filtering supported.

CORBA Messaging addresses message processing (not only event notification), and results are single return values due to CORBA AMIs being unicast. Results are communicated back to a client using the callback, or polling approach, alternatively. The failure level of messages is the processing level, and the failure scope are particular servers. CORBA Messaging does not support user-defined exceptions for AMIs, but message failure detection is based on system exceptions and processing results only (messages returning no results are successful if they return without an exception).

3.2 CORBA Events and Notification

CORBA Events and Notification refers to the messaging style based on the CORBA Events service [13], and its successor, the CORBA Notification service [13]. A variety of CORBA products currently support both services.

CORBA Events distinguishes generic (untyped), and typed event architectures. In the untyped case, a message corresponds to data that is passed in the form of the CORBA IDL type `any`. In the typed case, a message corresponds to an operation invocation of an application-specific IDL interface. In both cases, an application-independent messaging API is used, which defines standard interfaces for either push- or pull-based consumers and suppliers. Any number of event channels (being standard CORBA objects) may be used as intermediators

[1] CORBA also defines two other models of asynchronous invocation: IDL oneway operations, and deferred synchronous invocations. IDL oneways are, however, of unreliable best-effort delivery semantics, and serve only for event notification, but not for request processing returning results. CORBA deferred synchronous invocations, on the other hand, are only available with the CORBA dynamic invocation interface DII, which is a very complex, and thus less practical model.

for event notification, and a standard subscription mechanism for consumers and suppliers is defined. Using channels, multicast message distribution based on subscription is supported. All communication partners are known to the intermediator, but suppliers and consumers are anonymous to each other.

Asynchronicity is obtained using synchronous calls via an intermediator. If no intermediator is used, the communication is synchronous. With typed events, the application-specific interfaces contain standard synchronous CORBA operations, which in addition must not have any return values and out/inout-parameters. The delivery guarantee is at-most-once (standard CORBA). Messages may be persistent in the untyped case, if the intermediator (the service implementation) used supports persistence as a feature. The message delivery order is undefined for CORBA Events, but CORBA Notification supports per-consumer priority-specified message ordering. Message filtering, both header (type)-based, or content-based, is supported as well with CORBA Notification, but not with CORBA Events.

CORBA Events and Notification address event notification, but not message processing. Messages are parameters to standard CORBA requests, and results of message processing would need to be communicated back as parameters of separate reply requests. However, this is outside the scope of the services.

Figure 4 illustrates examples of channel-based messaging using the CORBA Events or Notification service.

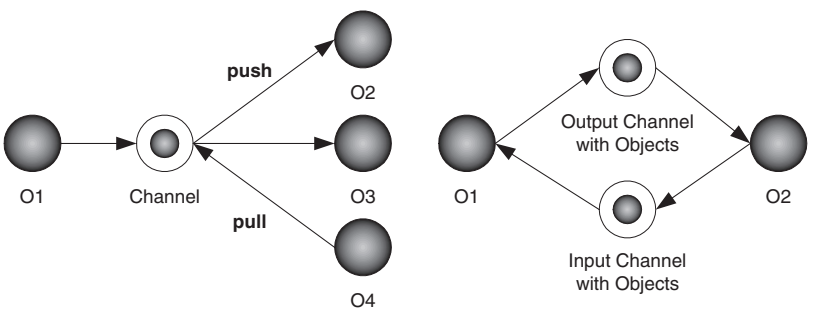

Fig. 4. Channel-based Messaging

3.3 Java Messaging

Java Messaging refers to the messaging style based on the *Java Messaging Service (JMS)* [17]. JMS addresses the integration of *message-oriented middleware (MOM)* to support messaging in Java object systems.

A message is an object of one of five JMS message types (`BytesMessage`, `TextMessage`, `MapMessage`, `StreamMessage`, or `ObjectMessage`), which are all specializations of the general JMS `Message` type, defining common message

header fields, properties, and operations. The JMS messaging API is application-independent, and comprises a set of interfaces for point-to-point (PTP) messaging, and for publish/subscribe messaging. PTP is the model of sending a message to a `Queue` object as an intermediator. A `Queue` encapsulates a specific, single MOM message queue in order to integrate MOM-based applications. Publish/subscribe is the model of sending a message to a `Topic` object. A `Topic` is an object in a content-based hierarchy, and serves as an intermediator to which interested Java consumers can subscribe to. `Queue` objects and `Topic` objects are specializations of the JMS `Destination` type.

Both models support multicast communication (to either MOM-based consumers, or Java consumers), with the set of consumers being anonymous to the message producer. A subscription mechanisms exists for the publish/subscribe model. Asynchronicity is achieved using synchronous calls to `Destination` objects. Messages can be declared to be persistent, or transient. Persistent messages are guaranteed to be delivered exactly-once, whereas delivery for non-persistent messages is at-most-once. Messages are delivered in the order they were sent (FIFO), however, message priorities can be specified. In addition, consumers may select messages from intermediators.

JMS describes a message processing model, where results of message processing are communicated back as separate reply messages (messages carry ids and can be correlated). The failure level of messages is the processing level, and the failure scope are specific servers. The combination of exceptions, processing results, and/or timeouts can be used for message failure detection.

3.4 MQ Messaging

MQ Messaging refers to the messaging style as suggested by message queueing (MQ) systems [2] [7].

A message corresponds to data that is structured according to a standard format of message header and message body. Messages are exchanged using (multiple, input and output) message queues as intermediators. On each processor in the network, a queue manager exists, and clients and servers use an application-independent message queue interface for exchanging messages, i.e., to put messages on queues (push), or to read messages from queues (pull). Target queues must be specified, but ultimate message consumers (unicast, or multicast) are anonymous to the sender. There is no subscription mechanism to queues, as clients and servers always decide themselves when and if to take a message from a queue[2].

Asynchronicity is realized using synchronous calls to queues. Message delivery is guaranteed to be exactly-once, when messages are declared to be persistent. Delivery is FIFO, or priority-specified, and message selection is the responsibility of the program reading from the queue.

[2] The exception are *MQ trigger queues*, which notify an application about the arrival of a message in a queue. The message itself is, however, not automatically pushed by the queue to the application, but must be read by the application itself.

MQ Messaging addresses event notification, but also message processing using separate reply messages with correlated ids. If the request/reply message model is selected, the message failure level is the processing level. Message failures are detected by queue managers and applications using exceptions and timeouts. Figure 5 illustrates a common queue-based messaging architecture.

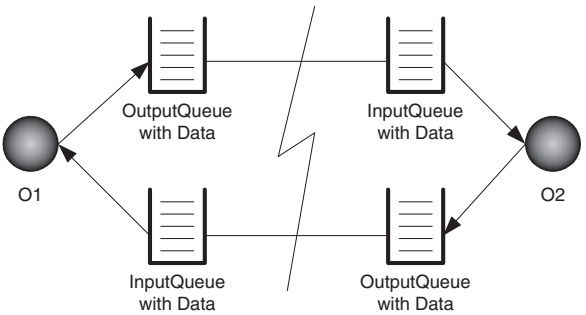

Fig. 5. Queue-based Messaging

3.5 Comparison

CORBA Messaging is the only messaging middleware that introduces an asynchronous invocation model for distributed objects. All other middleware represent messages as objects or data that is send as parameters of requests, and asynchronicity is achieved using synchronous communication via intermediators and an application-independent messaging API. Message initiation is commonly push and pull, but CORBA Events is the only model where messages can also be pushed by intermediators. With all middleware except CORBA Messaging, multicast (as well as unicast), and anonimity of ultimate recipients is supported. Subscription models vary. Exactly-once message delivery guarantees are provided with persistent messages only, the at-most-once semantics is the common case otherwise. Standard message ordering is FIFO (time the request is issued), or priority-specified. Filtering by intermediators is only supported with CORBA Notification. JMS and MQ provide, however, message selection features for applications connecting to an intermediator. None of the middleware provides a feature to return a single, integrated processing result of a multicast message, but results of a multicast message are returned as a set of individual reply messages. All models except CORBA Events/Notification address message processing, and not only event notification. The message failure level thus is the processing level in these cases. The failure scope is in any case a defined set of particular servers (or, intermediators) only (typically a single server). Failure detection commonly is the combination of exceptions, processing results, and/or timeouts.

Table 6. Messaging Comparison Table

Middleware	1. Representation	2. Messaging API
CORBA Messaging	3. invocation	2. specific
CORBA Events untyped	2. data	1. independent
CORBA Events typed	3. invocation	3. combination
CORBA Notification	2. data	1. independent
JMS Messaging	1. object	1. independent
MQ Messaging	2. data	1. independent

3. Initiation	4. Intermediation	5. Multiplicity	6. Anonimity
1. by producer	1. none	1. unicast	1. known set
3. push and pull	3. multiple	3. both	2. unknown set
3. push and pull	3. multiple	3. both	2. unknown set
3. push and pull	3. multiple	3. both	2. unknown set
3. push and pull	3. multiple	3. both	3. mixed
3. push and pull	3. multiple	3. both	2. unknown set

7. Subscription	8. Synchronicity	9. Delivery Guarantee	10. Persistence
2. no subscription	2. asynchronous	4. either 2. or 3.	3. either 1. or 2.
1. subscription	1. synchronous	2. at-most-once	2. transient (*)
1. subscription	1. synchronous	2. at-most-once	2. transient (*)
1. subscription	1. synchronous	2. either 2. or 3.	3. either 1. or 2.
3. mixed	1. synchronous	4. either 2. or 3.	3. either 1. or 2.
2. no subscription	1. synchronous	4. either 2. or 3.	3. either 1. or 2.

11. Ordering	12. Filtering	13. Proc. Results	14. Communication
1. FIFO/4. priority	1. none	1. single value	2. callback/3. polling
undefined	1. none	n/a	n/a
undefined	1. none	n/a	n/a
any/4. priority	4. 2. and 3.	n/a	n/a
1. FIFO/4. priority	2. header/type	3. set of values	1. separate message
1. FIFO/4. priority	2. header/type	3. set of values	1. separate message

F1. Failure Level	F2. Failure Scope	F3. Failure Detection
2. processing	1. particular	3. combination
1. delivery	n/a	n/a
1. delivery	n/a	n/a
1. delivery	n/a	n/a
2. processing	1. particular	3. combination
2. processing	1. particular	3. combination

(*) Persistence may be supported by service implementations

4 Integration Strategies

There is no single approach to integrating messaging and distributed transactions, due to different integration objectives, and the variety of the messaging models existing. In the following, we present four integration strategies, which all follow a distinct flavor of messaging set in the context of distributed object transactions. The strategies include a discussion of the initial integration models that are currently proposed by the JMS and CORBA Messaging.

The distributed transaction model that we use as a basis here is the two-phase commit transaction model as proposed, for instance, by the CORBA OTS, the transaction standard for distributed objects [13], and as specified by the X/Open XA distributed transaction processing model [18]. The OTS is a common and well-accepted transaction model, used by many modern middleware like CORBA OTMs, and the Java transaction service JTS, the Java mapping of the OTS.

4.1 MQ-Integrating Transactions

The first strategy is called *MQ-Integrating Transactions*. This integration strategy is the only one of the four strategies that is readily supported with current messaging and distributed object transaction middleware.

Intent. The intent is to integrate message queues of common MQ-systems as *resource managers* into the distributed object transaction. The distributed object system can in this way make use and incorporate the quality-of-services that are associated with message queues and MQ-systems.

Concept. MQ-integrating transactions integrate messaging in the sense that messages are transactional data that are managed by persistent message queues acting as resource managers. Data `get()` and `put()` calls on queues are made within a transaction scope. The following simple example illustrates this.

```
try {
   tx.begin();
   {
      data = inputQueue.getData();
      result = distributedObject.process(data);
      outputQueue.putData(result);
   }
   tx.commit();
} catch (Exception e)
{
   tx.rollback();
}
```

The transactional semantics here is that the dequeuing of the data from the input queue, the processing of the data by the distributed object, and the enqueuing of the data in the output queue, are executed as *one atomic* action.

Integrating message queues as resource managers basically compares to integrating a server that uses an XA-based database resource manager. Note that all put- and get-calls on the queues are local calls, and not remote. With MQ-integrating transactions, distributed message exchange is *not* part of the transaction. The transaction begins after a message has arrived in a local input queue, and the transaction commits when data is written to a local output queue.

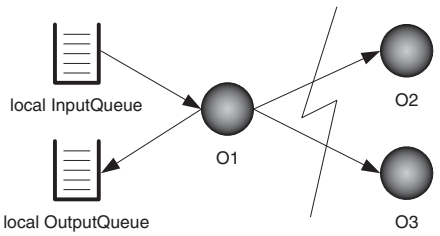

Fig. 6. MQ-Integrating Transactions

Implementation All major MQ-systems support the X/Open XA interface, which allows a message queue to be easily integrated as a resource manager into an OTS-like transaction.

4.2 Message Delivery Transactions

Message Delivery Transactions refers to the approach of integrating a *message delivery model* into distributed object transactions.

Intent. The intent is to enable *event notification* between remote communication partners: a message is send from a client to one or multiple distributed servers within the scope of a transaction, and among other, synchronous transactional requests (to the same or different servers). The ACID properties of the transaction must still be guaranteed, i.e., failure of message delivery must cause the transaction to abort, and if the transaction fails for some reason after a message has been sent out, a compensation strategy is needed to back out all messages sent.

A typical scenario and use of message delivery transactions is illustrated in Figure 7. The transactional client begins the transaction, sends out a message `m1` to a defined set of direct consumers, does some distributed transaction processing, and sends out another message `m2` to the same (or, a different) set of

consumers. If this transaction were to be implemented as a non-messaging, standard OTS-transaction, each message must be mapped to a number of individual, synchronous, and blocking calls, leading to a more complex and time-consuming transaction.

The main purpose and benefit of message delivery transactions is the ability to send messages during the course of the transaction. Asynchronous event notification is particularly useful for long-running transactions. Message delivery transactions allow a client to begin or continue distributed transaction processing without being blocked when sending notification messages.

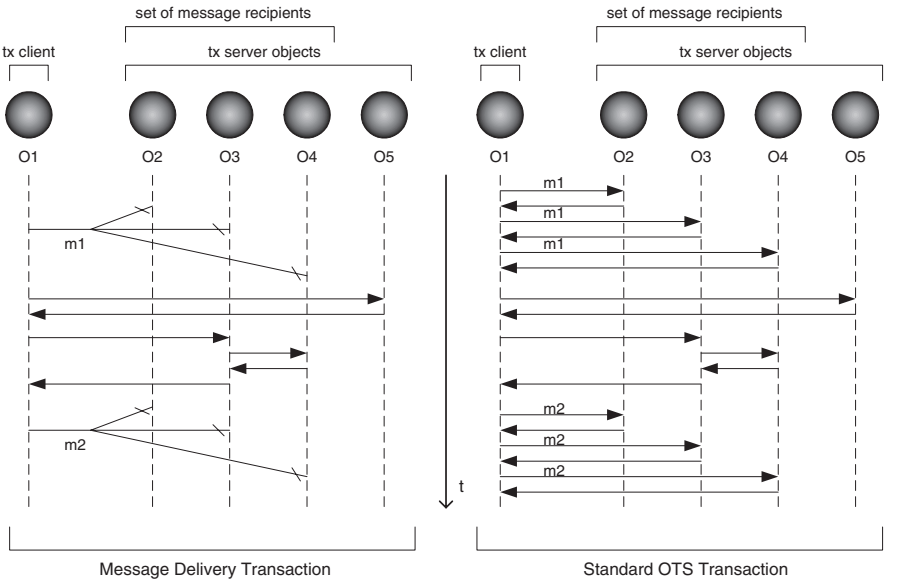

Fig. 7. Message Delivery Transactions Example

Concept. Message delivery transactions extend the standard transaction model in that

1. messages can be send in addition to synchronous object invocations, at any point in the transaction,
2. message delivery success is observed, and delivery failure can cause a transaction to abort, and
3. messages that are sent prior to a transaction failure are compensated.

Messages become part of the *atomic sphere*, the set of operations that make up the transaction. Compensation can be achieved by dequeing an original message from the intermediators that the message was sent to, if the message has

not been read by consumers from the intermediators at the point of transaction rollback. Otherwise, a separate message to undo the effects of the original message is sent out to the same consumers, which requires the definition of a *compensation* sphere, the set of compensating operations that are part of the transaction [10].

Table 7 describes the messaging model for message delivery transactions.

Table 7. Messaging Model for Message Delivery Transactions

1. Representation	any
2. Messaging API	any
3. Initiation	push-model: the message delivery is caused by the transactional client as the message producer (the pull-model is not desired, as this would require the transactional client to wait for a message pull and impact the transaction control flow)
4. Intermediators	any
5. Multiplicity	any
6. Anonimity	any
7. Subscription	any, if subscription is with intermediators (subscription to the transactional client itself does not affect the transaction)
8. Synchronicity	any
9. Delivery Guarantee	exactly-once semantics
10. Persistence	persistent (exactly-once delivery guarantee)
11. Ordering	any
12. Filtering	any
13. Processing Results	n/a
14. Communication	n/a
F1. Failure Level	delivery
F2. Failure Scope	any
F3. Failure Detection	combined model of exceptions and timeouts for delivery acknowledgments

Implementation. Message delivery transactions are not readily supported with current messaging and transaction services, as the integration model proposed by messaging services does not conform to the model of message delivery transactions. Most notably, current messaging middleware does not allow to send messages at any point of a transaction, but if they allow to include messages in a transaction, the messages are only send after and in case of a successful commit the transaction. Consequently, message failure as well as message compensation are not addressed at all.

For example, the JMS talks about *transacted sessions*, which allows to group a set of produced messages and consumed messages as one atomic unit of work [17]. However, transacted sessions are not distributed transactions, but local, and the produced messages are sent out to distributed partners only in case of a successful commit. JMS does not address distributed transactions, but suggests to use the JMS-supported XA resource manager interface in the case of distributed transactions. Thus, JMS distributed transactions essentially are MQ-integrating transactions.

To implement message delivery transactions with current messaging and transaction middleware, two principal options exist:

1. Messages are declared to be outside the scope of the transaction, and sent concurrently to the running transaction. Message delivery observation, as well as message compensation, must be implemented at own costs.
2. A series of transactions is defined. Each subtransaction represents a synchronization point at which messages are sent. The series of transactions is a saga transaction with a corresponding set of compensating transactions [4]. Sagas must be hand-coded as well.

Both solutions are unsatisfactory, as they are very costly to be implemented, and even more, to be maintained. By matching Table 6 and Table 7, we can further identify that none of the current messaging middleware allows for a flexible message failure scope definition. Ideally, any cardinality specification for message recipients is useful, especially as ultimate consumers are anonimous.

4.3 Message Processing Transactions

Message Processing Transactions refers to the approach of integrating a *message processing model* into distributed object transactions.

Intent. The intent is to enable *asynchronous request processing* between transactional distributed objects. A transactional client requests some distributed processing within the scope of a transaction, but is not blocked until the processing results for the request return. In addition, message processing transactions do not require transactional servers to be available at the time that the request is issued by the client. The ACID properties must still be guaranteed to the transaction.

A typical scenario and use of message processing transactions is illustrated in Figure 8. A transactional client begins a transaction, invokes on a distributed transactional server O2 asynchronously, continues its processing, and eventually receives a processing result from O2 prior to committing the transaction. This transaction model cannot be mapped to a standard OTS transaction, due to the fact that a standard OTS transaction requires all servers to be available at the times that the client issues the request.

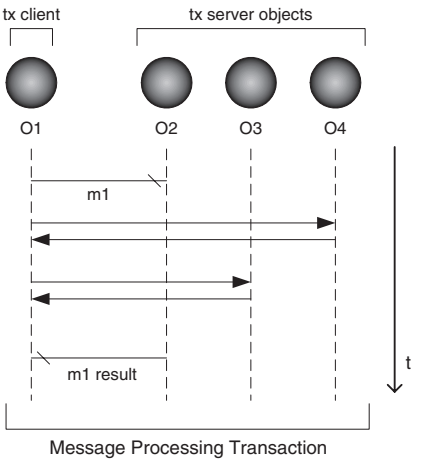

Fig. 8. Message Processing Transactions Example

Concept. Message processing transactions extend the conventional transaction model in that the atomic sphere of the transaction includes asynchronous requests. Different to message delivery transactions, this requires that the *transaction context* is shared between all transaction participants, i.e., the transaction context must be propagated from the client to the processing server[3]. The processing server is nonymous to the client, but need not be available at the time the client request is issued. The main purpose and benefit is the ability to process requests asynchronously, and to allow for independent client and server process lifetimes.

The messaging model for message processing transactions must meet the criteria as described in Table 8.

Implementation. The CORBA Messaging specification briefly addresses a special, CORBA-specific case of message processing transactions. CORBA Messaging refers to this transaction model as *unshared transactions*.

CORBA unshared transactions involve CORBA TIIs (time-independent requests), and thus do not have end-to-end transaction semantics (transaction contexts are not propagated from client to server at the time the request is made). Unshared transactions are different from CORBA messaging *shared transactions*, which involve CORBA AMIs only, and therefore can be mapped to standard OTS transactions that have an end-to-end transaction semantics.

CORBA unshared transactions describe a new transaction semantics. In principle, they may be implemented using three separate transactions: (1) the send-

[3] Transaction context propagation is not necessarily required for message delivery transactions, due to the client declaring a lack of interest in the consequences and processing of the message sent.

Table 8. Messaging Model for Message Processing Transactions

1. Representation	any
2. Messaging API	any
3. Initiation	push-model
4. Intermediators	any
5. Multiplicity	unicast or multicast
6. Anonimity	known set
7. Subscription	any, if subscription is with intermediators
8. Synchronicity	any
9. Delivery Guarantee	exactly-once semantics
10. Persistence	persistent
11. Ordering	any
12. Filtering	any
13. Processing Results	any
14. Communication	any
F1. Failure Level	processing
F2. Failure Scope	defined set of particular recipients
F3. Failure Detection	combined model of exceptions and timeouts

ing of the request by the client, (2) the delivery of the request, and the reception of the processing result, by the middleware, and (3) the propagation of the processing result to the client. (This model corresponds to the non-object-oriented notion of transactions as proposed with MQ systems.) However, such an implementation requires an ordering guarantee for requests, and a concept to identify a set of ordered requests as a single entity. Both is not defined and supported with CORBA and CORBA Messaging.

The problems described for CORBA unshared transactions apply to message processing transactions in general. Table 8 also reveals other open issues. For example, for communicating back processing results, a mechanism that supports the integration of a set of results of a multicast request is highly desired.

4.4 Full Messaging Transactions

With *Full Messaging Transactions*, we refer to the approach of a complete distributed object transaction model that allows for message delivery transactions and message processing transactions at the same time.

The intent is to enable *event notification* and *asynchronous request processing*, alternatively, and in addition to synchronous object invocations, between remote components and within the scope of a single transaction.

Full messaging transactions combine message delivery transactions and message processing transactions. Thus, they require an even more flexible model for defining messages and message failures, as they must distinguish the messaging models and conflicting values such as failure scope definition (property F2 in Table 7 and Table 8).

4.5 Related Work

The ACS object communication protocol of the KAROS system [6] is a notable previous work in this field. With ACS, a request message is associated to an atomic action, which may comprise other messages. ACS thus provides transactional semantics to nested actions through asynchronous communication. Three different kinds of messages with different failure recovery semantics for asynchronous request processing (apply, call), and for event notification (send), are supported. ACS addresses reliable distributed object messaging using an implicit atomic operation execution semantics, as opposed to explicit transaction demarcation as proposed with distributed transaction services like the OTS. The ACS protocol can thus not directly be adopted for distributed object messaging/transaction architectures as discussed in this paper. Also, we aim at supporting asynchronous communication in addition to synchronous communication, but not as an exclusive alternative.

5 Conclusion

In this paper, we addressed the problem of integrating messaging and distributed object transactions. We stated the need for a common language to communicate about messaging and different models of messaging, and introduced a comprehensive messaging classification framework that serves for this purpose. The framework defines messaging properties and property values organized around three models: message delivery model, message processing model, and message failure model.

We demonstrated the use of this framework for two purposes: to study and compare different messaging architectural styles as induced by messaging middleware, and to characterize the messaging models behind different strategies to integrate messaging and distributed object transactions. The messaging middleware comparison revealed a number of important differences between the notions of messaging currently supported, including fundamental differences, for example, regarding message representation, synchronicity, or message delivery guarantees, and more subtle differences, for example, regarding the support for multicast communication, or for message filtering by intermediators.

We derived four strategies for integrating messaging and distributed object transactions, each serving for a specific integration objective and following a distinct flavor of messaging: MQ-integrating transactions, message delivery transactions, message processing transactions, and full messaging transactions. We described the intent and concept for each of these strategies, and identified open issues for future integration support. These include, most notably, the ability to send messages at any point within the scope of the transaction, to support message compensation, to allow for time-independent transaction context propagation, and to support flexible message failure definition w.r.t. message delivery and/or message processing.

Our plans for the future are to step-wise address the issues discussed for each integration strategy, and to eventually provide a middleware support for

full messaging transactions. The basis for our integration facility are a novel distributed object messaging, and an advanced distributed object transaction model that are currently being developed in two related projects at IBM Watson. We expect future work to expose additional, and more specialized aspects in the problem domain of integrating messaging and distributed transactions in distributed object environments, which we will need to address. The classification framework and the integration strategies presented in this paper are the first step towards our project goal.

References

1. Bernstein, P., Newcomer, E.: Principles of Transaction Processing. Morgan Kaufman. (1997) 308
2. Blakeley, B., Harris, H., Lewis, R.: Messaging and Queuing Using the MQI. McGraw-Hill. (1995) 309, 319
3. Carzaniga, A., DiNitto, E., Rosenblum, D., Wolf, A.: Issues in Supporting Event-based Architectural Styles. Proc. ISAW3, ACM. (1998) 316
4. Garcia-Molina, H., Salem, K.: Sagas. Proc. ACM SIGMOD. (1987) 326
5. Gray, R., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufman. (1993) 308
6. Guerraoui, R., Capobianchi, R., Lanusse, A., Roux, P.: Nesting Actions through Asynchronous Message Passing: the ACS Protocol. Proc. ECOOP'92, LNCS 615, Springer-Verlag. (1992) 329
7. IBM Corp.: MQSeries Application Programming Guide, 10th ed. IBM Corp. (1999) 319
8. IBM Corp.: An Introduction to Messaging and Queueing, 2nd ed. IBM Corp. (1995) 309
9. Jacobson,H. A., Olken, F., MacParland, C.: A Taxanomy of Event Services for Internet-Scale Monitoring and Control Applications (Draft). Technical Communication. (1998) 310
10. Leymann, F., Roller, D.: Production Workflow: Concepts and Techniques. Prentice-Hall. (1999) 325
11. Monson-Haefel, R.: Enterprise JavaBeans. O'Reilly. (1999) 308
12. The Common Object Request Broker: Architecture and Specification. OMG. (1995) http://www.omg.org 312
13. CORBAServices: The Common Object Service Specifications. OMG. (1997) http://www.omg.org 308, 309, 317, 322
14. CORBA Messaging (Joint Revised Submission). OMG TC Document orbos/98-05-05. (1998) http://www.omg.org 309, 312, 316
15. Orfali, R., Harkey, D.: Client/Server Programming with Java and CORBA, 2nd ed. Wiley. (1998) 308
16. Slama, D., Garbis, J., Russell, P.: Enterprise CORBA. Prentice-Hall (1999) 308
17. Sun Microsystems: Java Message Service, version 1.0.1. (1998) http://java.sun.com/products/jms/docs.html 309, 318, 326
18. X/Open Guide Distributed Transaction Processing: Reference Model, version 3. X/Open Ltd. (1996) 322