Message Queuing Patterns for Middleware-Mediated Transactions

Stefan Tai*, Alexander Totok**, Thomas Mikalsen*, Isabelle Rouvellou*

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

**Courant Institute of Mathematical Sciences, New York University, New York, USA
totok@cs.nyu.edu

Abstract. Many enterprise applications require the use of object-oriented middleware and message-oriented middleware in combination. Middleware-mediated transactions have been proposed as a transaction model to address reliability of such applications; they extend distributed object transactions to include message-oriented transactions. In this paper, we present three message queuing patterns that we have found useful for implementing middleware-mediated transactions. We discuss and show how the patterns can be applied to support guaranteed compensation in the engineering of transactional enterprise applications.

1 Introduction

Object-oriented middleware (OOM) and message-oriented middleware (MOM) are two widely used but different kinds of middleware. OOM, as exemplified by CORBA and Enterprise JavaBeans, promotes a synchronous, coupled, interface-driven communication style; MOM, as exemplified by MQSeries [7] and implementations of the Java Message Service (JMS) [17], promotes an asynchronous, decoupled, data-driven communication style. Many enterprise applications require the use of both OOM and MOM for integrating diverse legacy systems [14].

Two different notions of transactions correspondingly exist with OOM and MOM to address different kinds of reliability concerns. Distributed object transactions [15] are based on the X/Open distributed transaction processing model; they group a set of (remote) object invocations so that their execution is enclosed into an atomic sphere. Message-oriented transactions (messaging transactions) [2] group the publication and consumption of different messages (enqueing/dequeing to/from message queues) into an atomic unit-of-work. Distributed object transactions assume synchronous interactions and a life-cycle dependency between transaction participants (i.e., all object servers need to be available at the time of transaction processing), whereas message-oriented transactions separate the execution of sender and receiver programs, assuming eventual processing of requests by life-cycle independent message receivers.

Object transactions and messaging transactions each have their strengths and advantages, but often need to be combined in order to implement transactions that span across OOM application components and MOM application components. Different strategies for integrating the two have been identified [19]. One approach is to extend object transactions to include message-oriented transactions, as proposed by the Dependency-Spheres model [20] and the X^2TS model [12]. Dependency-Spheres and X^2TS transactions are *middleware-mediated transactions* [13] – distributed object transactions that integrate messaging as a communication style using messaging middleware for mediated interaction with loosely-coupled application components.

Middleware-mediated transactions require sophisticated system support to deal with recovery and fault-tolerance of object and messaging components. In this paper, we describe three message queuing patterns that we have found useful for implementing middleware-mediated transactions. We demonstrate the use of these patterns to support guaranteed compensation in the engineering of transactional enterprise applications.

The paper is structured as follows. In Section 2, we briefly summarize the concept of middleware-mediated transactions. In Section 3, we present the three message queuing patterns of *fire-and-remember*, *recoverable state machine*, and *detached compensation for orphaned transactions*. We discuss related work in Section 4, and conclude with a summary in Section 5.

2 Combining Object Transactions and Messaging Transactions

Figure 1 illustrates a common enterprise application scenario, where OOM application components (`Object1`, `Object2`, `Object3`) interact with each other and with MOM application components (`Recipient1`, `Recipient2`, `Recipient3`) using object interfaces (OOM brokering) and message queues (explicit MOM mediation), respectively.

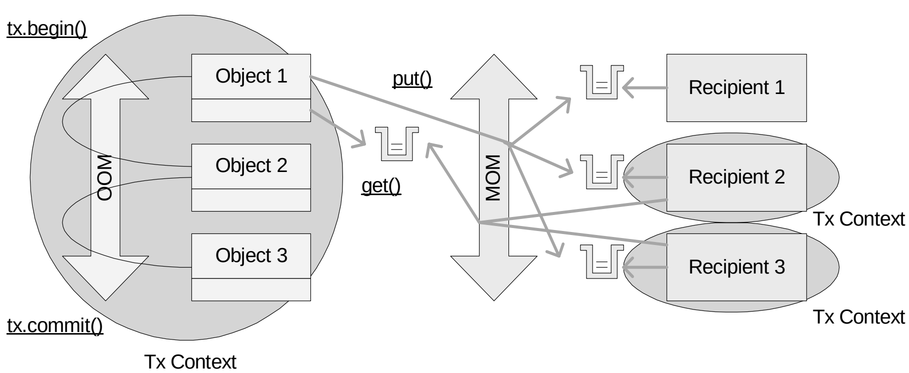

Figure 1: OOM Transactions and MOM Transactions

With standard middleware, no atomic unit-of-work grouping the diverse component actions and interactions can be defined. Standard middleware requires the single logical transaction of Figure 1 to be separated into multiple, independent transactions:

- The object transaction of the OOM application components, which may include *on-commit* message "put" requests to the queues of the MOM application components
- The transaction of `Recipient2`, which includes a message read from the queue of `Recipient2`, some processing, and a message "put" to the queue of `Object1`
- The transaction of `Recipient3`, which includes a message read from the queue of `Recipient3`, some processing, and a message "put" to the queue of `Object1`
- The transaction of `Object1`, which includes reading the result messages produced by `Recipient2` and `Recipient3` from the queue of `Object1`, and performing some subsequent processing

Standard middleware does not define and support any dependency management of these individual transactions; it is the responsibility of the application developer to program the global transaction and to ensure atomicity and system consistency in the presence of application and system failures.

Middleware-mediated transactions [13] overcome this limitation. Middleware-mediated transactions have been proposed as a framework for combining object transactions and messaging transactions, integrating concepts of standard transaction middleware, extended transaction models [4], and distributed programming language systems [8]. With middleware-mediated transactions like Dependency-Spheres [20], a global transaction context is established so that messaging interactions and individual messaging contexts of MOM application components are associated with a transactional object context. Therefore, in addition to object invocations, a transactional object client can also

- immediately publish or consume (sets of) MOM messages during an *ongoing* distributed object transaction as part of the object transaction,
- have message recipients with independent life-cycles and execution contexts be associated as participants of the object transaction, so that message recipient actions can affect the outcome of the object transaction, and vice versa.

Middleware-mediated transactions require compensation support for messages, as messages can be sent with immediate visibility during a transaction. Compensation is generally well-accepted as an important way to deal with failures [2], and fundamental not only to middleware-mediated transactions, but also to other extended transaction models such as Sagas [5] or ConTracts [16], to workflow and process support systems [11] [6], and conversational transactions of business-to-business interactions [3]. When a transaction fails, a cancellation/compensation message is imperative to undo any processing actions that the message recipient may have performed. The definition of the compensation semantics is an application responsibility, but the guarantee that a recipient receives a compensating message if a transaction fails (and also, that no recipient receives a compensating message if it has not prior received a primary message) should not be an application responsibility, but rather a function of the middleware.

In this paper, we report on our experiences in the design and implementation of middleware system support for guaranteed compensation in middleware-mediated transactions. We present three message queuing patterns, which are all motivated by transaction processing and compensation needs, but which may also be useful to address other, possibly non-transactional, application problems as well. The patterns have been implemented as part of the Dependency-Spheres middleware system, the principal architecture of which is described in [20] and [21].

3 Message Queuing Patterns

We introduce the three message queuing patterns of *fire-and-remember* (Section 3.1), *recoverable state machine* (Section 3.2), and *detached compensation for orphaned transactions* (Section 3.2). Each pattern is described in three subsections of problem/objective, solution/implementation, and use/consequences.

These patterns represent solutions that we have found useful and practicable as extensions to standard middleware. We believe that they can be applied not only as middleware extensions, but also to solve related design and implementation problems on the application level.

3.1 Fire-and-Remember

Standard message middleware [7] [17] supports two kinds of message sends: the immediate delivery of a message at the time it is created, and the (MOM-transactional) commit-based delivery of a message. A message is any application data plus MOM control data (and optionally also application control data), such as a unique message id or a message delivery timestamp. In a middleware-mediated transaction processing context, however, it may be useful to send a message predicated on the *failure* of the transaction (on-abort); this is not supported with standard MOM transactions.

Problem/Objective. Consider an application that implements a middleware-mediated transaction which includes messages that are sent out with immediate visibility. The application wants the simplicity to "fire and forget" when sending out a message. Yet, as firing takes place within a transaction, the application must "remember" something about the message in order to compensate in the event of a failure. The problem is amplified when the transaction failure model is complex. For instance, a particular set of acknowledgments of message receipt or message processing by final recipients may be required in order for the transaction to succeed ("conditional messaging") [21].

Ideally, the application would create a corresponding compensating message at the same time the primary message is sent, and have the delivery be predicated on the failure of the transaction. However, standard middleware does not support this type of message send. Therefore, the application typically will

- create some data structure for a compensating message,
- add application data for compensation and control data such as timestamps and the id of the primary message that it compensates, and
- store the data persistently.

Should the transaction abort, the application must then be (made) available to start a new messaging session to create and send out actual compensating messages using the stored data.

The objective of the "fire and remember" pattern is to support on-abort delivery semantics allowing an application to fire a primary message, create a corresponding compensating message at the same time, and have the middleware remember the compensating message to deliver it in case of a transaction failure. For every primary message that is sent, the existence of a compensating message can then be guaranteed. Furthermore, the application is no longer required to be (made) available should the transaction abort.

Solution/Implementation. The *fire-and-remember* messaging pattern is suggested to implement on-abort message transmission. For this purpose, a simple middleware system extending the functions of standard middleware (`Middleware Extension MWx`) is implemented, and a persistent message queue that is used to temporarily store compensating messages (`COMP.Q`) is set up (see Figure 2).

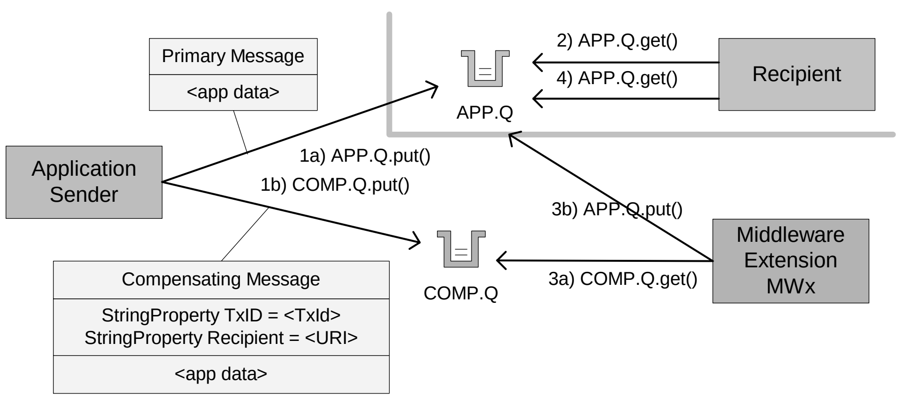

Figure 2: "Fire-and-Remember" Messaging Pattern

The `COMP.Q` is used as the destination for compensating messages. The compensating messages are put on this queue at the same time that the primary messages are sent out. The application encodes the actual message recipients for each compensating message using an agreed message property field, so that the MWx system can forward the compensating messages at a later time. The application itself can then forget about the messages.

The MWx system observes the sender's transaction, and once a transaction fails,

- the corresponding compensating messages are read from the COMP.Q queue (qualified reads based on the transaction id),
- the actual message recipient addresses are extracted from the messages, and
- the messages are forwarded to their designated destinations.

Using MOM transactional capabilities, compensating messages are forwarded atomically to ensure compensation across all required participants. This includes the atomic grouping of the "get" requests from the COMP.Q queue with the "put" requests to the target application queues.

The MWx system uses an agreed message property field to associate compensating messages to a transaction. The MWx system further encodes which action to take should the transaction succeed. For example, all compensating messages may be deleted.

Use/Consequences. The pattern requires a compensation queue (COMP.Q) to be set up. A further requirement is that the MWx system needs to be able to determine the transaction status of the sender application; the MWx system can register itself with the transaction monitor as a participant of the transaction (in order to be notified about the transaction status), or, the MWx system can be part of the transaction monitor itself.

We have implemented the MWx system as part of the Dependency-Spheres transaction monitor. We have found the "fire-and-remember" pattern to be a very simple and inexpensive solution that can easily be implemented. The pattern helps to significantly reduce application code and application programming complexity, as most of the required functions are now provided by the middleware.

3.2 Recoverable State Machine

Engineering a software system typically entails the definition of a model of the behaviour of the system. A state machine is such a model that specifies the sequences of states that a system goes through in response to events (such as signals, operations, or passing of time). It describes which state-dependent activities take place whenever an event occurs and when the system transitions from one state to another. In the following, we present a solution that represents and executes state machines reliably using persistent message queues and MOM transactions.

Problem/Objective. The implementation of a recoverable state machine involves the use of some persistent store to represent states, and the use of a reliable mechanism for transitioning between states. A persistent store is required in order to survive any hard- and software failures. A reliable transition mechanism is required to ensure that only consistent states are stored.

Databases and database transactions are a common, established approach to address this problem. However, if any pre-requisite for a database is to be removed (that is, a database is not otherwise required or desired), message queuing represents an alternative. Persistent queues and MOM transactions compare to databases, even though they have not been designed to replace databases (queues have been designed for remote messaging that is also persistent and transactionally reliable). Furthermore, by using queues for state machine representation and execution, MOM functionality can be employed to integrate any state-based notification that is of interest to the application. For example, in a transaction processing context, the event of entering the "aborting" state of a transaction may cause compensating messages to be automatically be sent out (see below).

Solution/Implementation. In our approach, we represent each state of a state machine as a persistent message queue. In the following, we illustrate the pattern with a state machine that models the states of a transaction. The transaction whose state is to be monitored is represented as a message, which is moved like a token from queue to queue whenever a state transition occurs. Each state transition is a MOM transaction that atomically groups the "get" operation from one state queue with the "put" operation to another state queue.

Figure 3 depicts the state machine of middleware-mediated transactions, and the corresponding persistent message queues, which are part of a larger middleware system (the transaction monitor) supporting the functions (API) and semantics of middleware-mediated transactions.

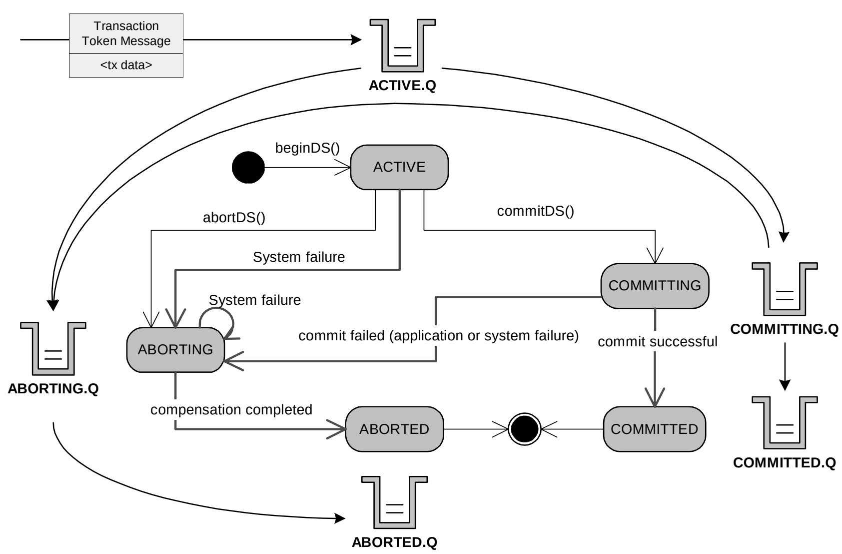

Figure 3: Recoverable State Machine Using Persistent Queues

Once a transaction is started, a message representing the transaction is created and put into the ACTIVE queue. The transaction remains active until the application tries to commit or to explicitly abort the transaction, or until a failure that ends the transaction occurs. A MOM transaction that moves the transaction message token to the COMMITTING or ABORTING queue is associated with these events as part of the commit() and abort() operation implementations, and as part of the component of the middleware system that detects application and system failures (see Section 3.3 below). The transaction message token remains in the respective queues as long as the commit or abort processes are ongoing (see [20] for a description of these processes). Whereas the commit process may also be timed out by the application, the aborting process cannot be timed out, but must be fully completed.

Use/Consequences. The use of persistent queues and MOM transactions allows to monitor multiple transactions independently of each other; the set of all active, aborting, aborted, committing, or committed transactions is easily determined by browsing the respective queues. As each queue is a persistent store that will survive any hard- and software failures, they reliably record the consistent state of transactions. Should a system failure occur (such as a crash of an application client or even a crash of the transaction monitor itself), the state of all transactions can be reconstructed based on the persistent queue data. Each transaction message token is guaranteed to exist in only one of the state queues at a given time, as MOM transactions are used to reliably move the message token between the state queues.

The queue-based state machine implementation can be combined with other messaging solutions and MOM functions, such as the "fire-and-remember" pattern introduced above. Compensating messages that are stored in the COMP.Q queue only need to be sent out when compensation is required, which corresponds to the transaction entering the ABORTING state. A message listener for the ABORTING queue could be used to automatically trigger a corresponding compensation process (the transaction monitor component that sends out the compensating messages that are stored in the COMP.Q queue).

A potential concern with this approach is its ability to scale: the number of queues that need to be set up and administered increases with the number of states in the state machine, and the number of MOM transactions executed increases with the state transitions taken. However, the distributed state representation lends itself naturally to workload management, failover, and parallelism.

3.3 Detached Compensation for Orphaned Transactions

Middleware-mediated transactions and other compensation-based processes and applications must perform compensation not only for explicitly aborted transactions or processes, but also for "orphaned" transactions. An orphaned transaction is an active transaction that cannot be committed or aborted because the application client process that started it has terminated abnormally. This can occur due to a system error such as a machine crash. In the following, we describe a pattern for detecting and driving compensation for orphaned transactions.

Problem/Objective. In middleware-mediated transactions, participants are loosely-coupled programs, with independent life-cycles, that communicate using asynchronous messaging. In this environment, there is no obvious difference between a participant not reacting due to a failure or due to deliberate muteness. Determining the failure of a participant whose processing is viable to the success of the transaction is therefore difficult.

Despite the difficulty, the failure of a transaction owner must be determined, as it results in an orphaned transaction that will not be terminated. Orphaned transactions consume resources unnecessarily and leave the involved participants forever in doubt about the outcome of the transaction.

A timeout can be used to detect orphaned transactions. However, this requires careful selection of the timeout value; otherwise, a non-orphaned transaction might be aborted, or, an orphaned transaction might be detected too late.

A heartbeat-based solution in combination with a detached process can be used to more accurately detect and abort orphaned transactions. This approach compares to work on group communication systems and distributed systems protocols for detecting process failures based on heartbeats. In a middleware-mediated transaction environment, using MOM to implement heartbeating enables the natural integration with compensation processes for orphaned transactions.

Solution/Implementation. An application client (transaction owner) regularly sends out heartbeat messages for its transactions to a dedicated queue, the `ALIVE.Q`. The time interval for sending heartbeats is specified by the application. A detached process (that is part of the transaction monitor) uses the `ALIVE.Q` queue to receive heartbeat messages. If it does not receive an expected heartbeat for a transaction, the application client process is considered dead and the transaction is orphaned, and compensation is initiated.

Figure 4 (left side) illustrates the heartbeat implementation for the Dependency-Spheres system. Each Dependency-Sphere transaction context (represented by the object `DSObject` of the D-Sphere system) is a two-threaded process. The main thread follows the application logic of the transaction originator, in which distributed object invocations and distributed messaging operations occur as part of the logical D-Sphere transaction. This thread is also associated with any conventional transaction context of the underlying distributed object transaction service (JTS [18]) that the D-Sphere system uses. The second thread, the heartbeat thread, sends out heartbeat messages to the `ALIVE.Q`. A `HEARTBEAT_INTERVAL` time is defined, which describes the cycle after which the heartbeat thread checks the liveness of the main thread and, if the main thread is alive, sends out a heartbeat message to the `ALIVE.Q`.

Figure 4 (right side) illustrates the detached process for detecting orphaned transactions and for driving compensation, as implemented in the Dependency-Spheres system. This im-

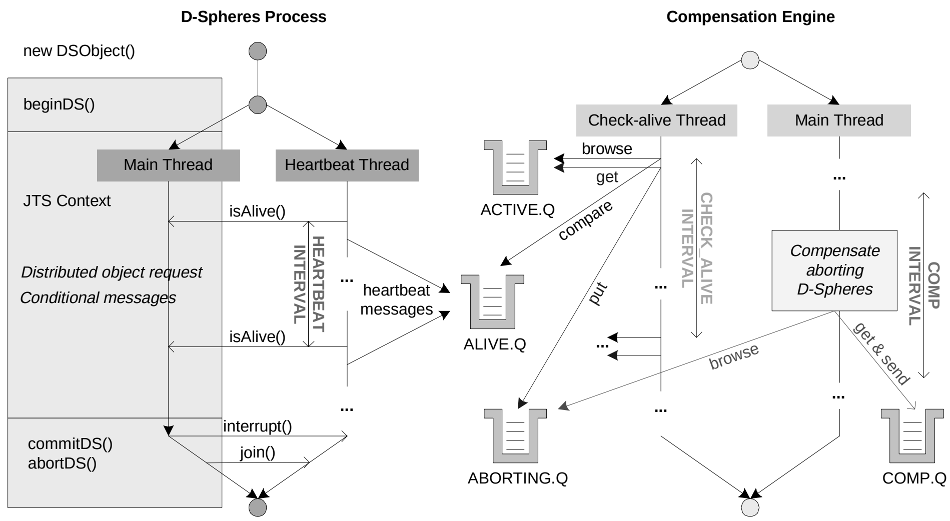

Figure 4: Detached Compensation for Orphaned Transactions

plementation applies the two previously described patterns of "fire-and-remember" and "recoverable state machine".

The detached process (the `Compensation Engine` component) finds out about all active transactions by browsing the `ACTIVE.Q` state queue. For each active transaction, corresponding heartbeat messages are expected in the `ALIVE.Q`. A `CHECK_ALIVE_INTERVAL` time (that is greater than the `HEARTBEAT_INTERVAL` time) is specified to set the time after which the compensation engine should repeat the task of browsing and comparing the `ACTIVE.Q` entries with heartbeats arriving in the `ALIVE.Q`. If a heartbeat message is found for an active transaction, the transaction is considered alive. If no heartbeat message is found, the transaction is considered orphaned and the compensation engine moves the transaction message token from the `ACTIVE.Q` to the `ABORTING.Q`. This will cause the compensation process for the transaction to be started, which comprises the forwarding of the compensating messages that are stored in the `COMP.Q` queue.

Use/Consequences. This pattern can be combined with the two previously described patterns to implement *guaranteed compensation* in middleware-mediated transactions. Application and system failures are tolerated, including failures of the compensation engine (the detached process) itself, and the delivery of compensating messages is guaranteed.

A potential concern with this approach is that the message queuing topology in combination with the MOM implementation could interfere with the timely delivery of heartbeat messages. The problem of timely delivery of messages, however, is not unique to this pattern. Further, we believe that appropriate topologies and MOM implementations exist.

4 Related Work

The patterns discussed in this paper address the development of middleware system support for transactional enterprise applications. They describe solutions employed in the context of the Dependency-Spheres research system prototype.

The development of system support for middleware-mediated transactions relates to the development of workflow management systems (WFMS) [11] and to the development of system support for other kinds of extended transaction processing models. In comparison to WFMS, our work describes a lower-level middleware-oriented approach to support the focused objective of middleware-mediated transactions; WFMS provide a much broader functionality since their goal is to coordinate users and programs in addition to activities and data. The arguments for using persistent queues and message-oriented middleware to build a workflow management system [10], however, are the same as in our discussion. As Leymann and Roller point out, the use of persistent queues introduces almost for free many features that help improve system and application robustness. In this paper, we introduced three specific message queuing patterns that reinforce these statements and showed how these patterns can be applied to implement guaranteed compensation.

Compensation, as mentioned earlier, is a well-accepted way to deal with failures, and is part of various advanced transaction models of theory and practice [4] [1], and recent proposals of general activity coordination frameworks [9]. Common to all compensation-based approaches is the need to monitor and drive compensation processes reliably until the processes are properly completed. The approach for guaranteed compensation in Dependency-Spheres as introduced in this paper accomplishes this need; compensation is implemented reliably in that different kinds of application and system failures, including failures of the compensation engine itself, are detected and tolerated. To our knowledge, related work on compensation has not reported on how to implement guaranteed compensation, but concentrated on other aspects such as formal specification or the construction of compensation graphs for partially executed business processes.

5 Summary

Object-oriented middleware and message-oriented middleware are often used in combination for purposes of enterprise application integration. Consequently, the combination of distributed object transactions and message-oriented transactions into middleware-mediated transactions is requested. In this paper, we described three message queuing patterns that we have found useful for implementing middleware-mediated transactions:

- *Fire-and-remember.* This pattern supports on-abort delivery semantics allowing an application to fire a primary message, create a correlated message (such as a corresponding compensating message) at the same time, and have the middleware remember the correlated message to deliver it in case of a transaction failure.

- *Recoverable state machine.* This pattern represents and executes a fault-tolerant state machine using persistent message queues and MOM transactions. A message is used to represent the entity that is to be monitored (e.g., a transaction, object, system), queues are used to represent entity states, and MOM transactions are used to reliably transition between states.
- *Detached compensation for orphaned transactions.* This pattern describes a heartbeat-based solution in combination with a detached process to detect and drive compensation for orphaned transactions.

These patterns are used in the Dependency-Spheres system prototype to implement guaranteed compensation as a middleware extension. The ideas presented may further help to solve design and implementation problems encountered by MOM applications.

Acknowledgments

We would like to thank Stan Sutton for his useful comments on a previous version of this paper. We would also like to thank Stuart Jones and Stewart Palmer for insightful discussions about message queuing in general.

6 References

[1] G. Alonso, D. Agrawal, A. El Abbadi, M. Kamath, R. Günthör, C. Mohan. Advanced Transaction Models in Workflow Contexts. In Proc. International Conference on Data Engineering (ICDE'96), IEEE, 1996.

[2] P. Bernstein, E. Newcomer. Principles of Transaction Processing. Morgan Kaufmann, 1997.

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

[4] A. K. Elmagarmid (Ed.) Database Transaction Models for Advanced Applications. Morgan Kaufmann, 1992.

[5] H. Garcia-Molina, K. Salem. Sagas. In Proc. ACM SIGMOD International Conference on Management of Data, 1987.

[6] P. Grefen, J. Vonk, P. Apers. Global Transaction Support for Workflow Management Systems: From Formal Specification to Practical Implementation. The VLDB Journal (2001), Springer-Verlag, 2001.

[7] IBM MQSeries. http://www-4.ibm.com/software/ts/mqseries/

[8] R. Guerraoui, R. Capobianchi, A. Lanusse, P. Roux. Nesting Actions Through Asynchronous Message Passing: the ACS Protocol. In Proc. European Conference on Object-Oriented Programming (ECOOP'92), Springer-Verlag, 1992.

[9] I. Houston, M. Little, I. Robinson, S. Shrivastava, S. Wheater. The CORBA Activity Service Framework for Supporting Extended Transactions. In Proc. International Conference on Distributed Systems Platforms (Middleware 2001), Springer-Verlag LNCS 2218, 2001.

[10] F. Leymann, D. Roller. Building A Robust Workflow Management System With Persistent Queues and Stored Procedures. In Proc. International Conference on Data Engineering (ICDE'98), IEEE, 1998.

[11] F. Leymann, D. Roller. Production Workflow: Concepts and Techniques. Prentice-Hall, 2000.

[12] C. Liebig, M. Malva, A. Buchmann. Integrating Notifications and Transactions: Concepts and X^2TS Prototype. In Proc. 2nd International Workshop on Engineering Distributed Objects (EDO 2000), Springer-Verlag LNCS 1999, 2001.

[13] C. Liebig, S. Tai. Middleware-Mediated Transactions. In Proc. 3rd International Symposium on Distributed Objects and Applications (DOA 2001), IEEE, 2001.

[14] D. Linthicum. Enterprise Application Integration. Addison-Wesley, 2000.

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

[16] A. Reuter, F. Schwenkreis. ConTracts: A Low-Level Mechanism for Building General Purpose Workflow Management Systems. IEEE Data Engineering Bulletin, vol.18, no.1, 1995.

[17] Sun Microsystems. Java Message Service API (JMS) Specification v1.02. Sun, 1999.

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

[19] S. Tai, I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In Proc. International Conference on Distributed Systems Platforms (Middleware 2000), Springer-Verlag LNCS 1795, 2000.

[20] S. Tai, T. Mikalsen, I. Rouvellou, S. Sutton. Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages. In Proc. 5th International Enterprise Distributed Object Computing Conference (EDOC 2001), IEEE, 2001.

[21] S. Tai, T. Mikalsen, I. Rouvellou, S. Sutton. Conditional Messaging: Extending Reliable Messaging with Application Conditions. In Proc. 22nd International Conference on Distributed Computing Systems (ICDCS 2002), IEEE, 2002.