Middleware Mediated Transactions

Christoph Liebig* and Stefan Tai**

*Darmstadt University of Technology, Darmstadt, Germany
chris@informatik.tu-darmstadt.de

**IBM T.J. Watson Research Center, New York, U.S.A.
stai@us.ibm.com

Abstract

Middleware Mediated Transactions (MMT) integrate message-oriented transactions and distributed object transactions. MMT are suggested as an evolutionary and integrative approach to support reliable and flexible interactions between heterogeneous and autonomous components, which is a major challenge in enterprise application integration. MMT offer the ability to combine communication of messages and notifications with conventional transactional object requests. Thus MMT introduce the flexibility of mediated interactions with respect to topology, binding, time-dependencies and content transformation into distributed object transactions. MMT are characterized by coupling modes to control if notifications become visible immediately or are dependent on the transaction status, to include mediators as transaction participants, and to distinguish between message delivery and processing, as well as vital and non-vital participants. Furthermore, coupling modes interrelate different distributed transaction contexts of publishers and subscribers. This paper introduces the concept of MMT and presents two system prototypes implementing MMT, the Dependency-Spheres middleware service and the X^2TS middleware service.

1. Introduction

Middleware is application-independent connectivity software that is commonly used for the integration of enterprise applications (EAI) in distributed and heterogeneous environments. Cooperative information systems are one example class of systems that typically employ middleware for purposes of EAI. A cooperative information system connects formerly independent information systems into one comprehensive system, where each information system – characterized by its own application processes, common data and multiple users – acts in the role of a component [7].

Building cooperative information systems leveraging *explicit middleware mediation* is an attractive and commonly adopted approach. The explicit use of middleware mediation, as suggested with message-oriented middleware and object notification services, allows for flexible and complex interaction models while preserving component autonomy. *Object transaction processing*, on the other hand, also is an attractive and important approach in EAI to address issues of system reliability and correctness. However, object transactions are based on a synchronous 1:1 request/reply interaction model only, which induces a tight coupling among components. This is in contrast to the flexible messaging interaction models that often are required or desired.

The transactional, yet flexible integration of components of a cooperative information system, which typically are autonomous to varying degrees w.r.t. their design model, execution model, and/or communication model [6], consequently describes a difficult problem. Some components may not externalize an API for participating in the standard two-phase commit protocol (2PC) of a transaction, other components may not even support atomic executions at all. Two general solutions to this problem are to either exclude certain components from transactions, or to develop new components that render the autonomous components (if feasible) conforming to the transaction model used. We believe that compromising component autonomy for transaction processing in such ways is not always possible and advisable, and that instead a more tolerant approach to component autonomy in transaction processing is highly desirable and advantageous.

This paper presents *Middleware Mediated Transactions* as such a more tolerant approach. MMT offer the ability to combine communication of messages and notifications with conventional transactional object requests. Thus MMT introduce the flexibility of mediated interactions with respect to topology, binding, time-dependencies and content transformation into distributed object transactions.

1.1. Motivating Example

Consider the design and implementation of a collaborative workflow application that integrates various new and existing, autonomous components. A process to schedule group meetings is to be defined, which involves the sending of invitation messages (notifications) to multiple participants. A subset of all invited participants should accept the invitation (acknowledge the notification) in order for the meeting to take place and to confirm the update of distributed databases (e.g., for room reservation and participants calendars). That is, some of the participants are considered *vital* to the success of the scheduling process, while others are not.

If the process should be transactional (execute in an all-or-nothing manner), a transaction model is needed that allows to send out asynchronous invitation messages during the ongoing transaction with immediate visibility. In parallel, synchronous database updates may be performed. Before transaction commit, a reply from the vital participants is required in order for the transaction to commit. Therefore, a mixture of a (deferred) synchronous request/reply scheme, asynchronous (one-way) notifications, and synchronous database updates, all within the same transaction, is needed.

Furthermore, there may be technical challenges for implementing such transactions. For example, one database component may offer an object-oriented interface through an Enterprise JavaBean only, while another database may be accessible through a non-object asynchronous messaging interface only. The single transaction would need to integrate the two different, incompatible components.

The conventional transaction services provided by EAI middleware, such as the OMG Object Transaction Service (OTS) [27] or the message grouping mechanisms of message-oriented middleware [33], are not readily applicable for such problems. The OTS, for instance, groups synchronous requests to objects only, but does not include the immediate sending of messages. Similarly, messaging transactions group messages only, but do not integrate object invocations.

1.2. Paper Overview

In this paper, we present a solution to the described problem of transaction processing using standard EAI middleware. Our solution of *Middleware Mediated Transactions (MMT)* advocates the extension of object transactions to include message-oriented communication using messaging middleware.

MMT suggest to integrate mediation and mediators by means of queue-based and publish/subscribe messaging middleware into standard object transactions. We introduce the notion of *coupling modes* to define and describe kinds of transactional component and inter-transaction dependencies of MMT. We further present two different middleware services that support MMT. These services have been developed independently of each other, but both implement the idea of MMT by extending standard EAI middleware and providing functionality such as compensation support for revoked messages/notifications.

More particularly, among the unique features that MMT offer are the ability

- to combine communication of messages and notifications with transactional object requests to *extend the atomicity sphere* as established by conventional object transactions,
- to *include mediators and/or final message recipients* as transaction participants that can be either vital or non-vital to the success (respective failure) of the transaction,
- to *interrelate different distributed transaction contexts* of message senders/publishers and recipients/subscribers.

MMT are a novel approach to advancing the state-of-the-art in use of common middleware transactions. MMT is related to and incorporates concepts of extended transaction models and multi-database transaction management [6, 12, 17] and of distributed programming language systems [23, 14].

The paper is structured as follows. Section 2 reviews distributed communication and middleware mediation. Section 3 discusses background on transaction processing in the context of middleware. Section 4 then introduces *Middleware Mediated Transactions* and presents the concept of coupling modes of MMT. Section 5 describes the two service prototypes that realize MMT: the *Dependency-Spheres* service, which is based on message queuing middleware, and the *X^2TS* service, which is based on publish/subscribe middleware. Section 6 concludes the paper. Related work is mentioned and discussed throughout all sections.

2. Middleware Mediation

Middleware mediation refers to the indirection established by middleware for interaction among two or more (distributed) components. Middleware mediation can be implicit, or explicit. With *implicit* mediation, components are not aware of any intermediation by middleware; a component interacts with another component directly. With *explicit* mediation, components are well-aware of intermediation by middleware and explicitly make use of intermediation functionality provided by the middleware; a component communicates with another component through a mediator, a distinct entity (such as a message queue or an object channel) that essentially “decouples” the communicating components from each other.

Object middleware like CORBA promote implicit middleware mediation. The middleware itself may function as an explicit mediator for services such as naming and object binding, but the actual communication among the application components is direct by means of well-defined component interfaces. Message middleware, and messaging services of object middleware (such as CORBA’s Notification Service [26] or the Java Message Service JMS [16]), promote explicit mediation. Components do not provide application interfaces, but exchange messages (data) through mediators, using the services provided by the middleware for message exchange (e.g., put and get requests to queues, subscription to message subjects, and so on).

Explicit middleware mediation introduces significant flexibility for distributed communication. Explicit mediation further allows to incorporate message content transformations in the mediator. Data mappings for different message recipients may in this way easily be integrated. We believe that conventional transaction processing, which today is limited to implicit middleware mediation, needs to be extended to address explicit middleware mediation.

In the following, we identify different dimensions of distributed communication and distinguish implicit and explicit middleware mediation along those dimensions. We present two fundamental groups of distributed communication dimensions. The first group contains the dimensions related to the way that two or more communicating applications are connected to and interact with each other. The second group contains the dimensions that are related to the reliability associated to the interaction of the components.

Please note that we use the term *messaging* as a general term to refer to any kind of message communication that is based on explicit mediation. Mediators can be either *message queues*, or *publish/subscribe message brokers*. Similarly, we use the terms *sender* and *publisher* of a message, and *receiver* and *subscriber* (of queueing, and pub/sub communication, respectively), interchangeably.

2.1. Component Connection and Interaction

The *connection* between two or more components can be described in terms of topology and binding.

Topology. The topology describes the number (*arity*) of communication partners. The topology can be 1:1, 1:n, or n:m. The topology can be *fixed* or *variable*. For example, in a publish/subscribe system, there may be multiple publishers (n) and multiple subscribers (m), where the actual number of n and m may change over time.

Binding. The binding describes the means by which a relationship is established. With *reference-based binding*, at least one communication party has a reference to the others. With *mediator-based binding,* components are not directly connected, but use a mediator. The components do not have any references to each other themselves, but the references are stored by the mediator. The components are logically connected, for instance, by means of subjects (topics) of a defined subject-hierarchy, or by means of exchange of particular application-specific data (for example, with content-based publish/subscribe messaging).

Component *interaction* can be described in terms of life-cycle dependency and synchronicity.

Life-cycle dependency. The communicating partners can be *time-independent* or *time-dependent*. With time-independence, the components need not be available at the same time. With time-dependence, the components need to be available at the same time.

Synchronicity. The synchronicity of the communication partners describes the synchronization between them. With *synchronous* communication, one component A is synchronized with another component B with respect to the time that the component B replies to a request by A. The component A in this case is blocked waiting for a response from B. With *asynchronous* communication, the two components are not synchronized at all, and none of the components is blocked. With *deferred synchronous* communication, the communication is at first asynchronous, but synchronization takes place at a defined later point in time.

Common forms of component connection and interaction are, for example, the object-oriented communication model, which is 1:1 fixed, reference-based, time-dependent, and synchronous, and the publish/subscribe messaging model, which is n:m variable, subject- or content-based, time-independent, and asynchronous.

2.2. Reliability

Reliability of component interaction can apply to the guarantee of delivery of a request/message to a set of recipients, or, to the guarantee of processing of a request/message by a set of recipients. Reliability is addressed through mechanisms such as logging of transaction state and persistence of notifications that allow to recover from potential failures.

Delivery. The delivery of a request/message can either be not reliable (*best-effort* and *at-most-once* delivery), or be guaranteed to be reliable (*at-least-once* and *exactly-once*). With best-effort and at-most-once delivery, requests/messages may be delivered, and, with best-effort, may possibly even be delivered more than once. With at-least-once and exactly-once delivery, requests/messages are ensured to be delivered, and, with at-least-once, may possibly be delivered more than once.

There are various factors that determine or affect the process of delivery and the likelihood of delivery success, such as ordering and prioritizing requests/messages, the initiation of delivery (push versus pull), or the level of destinations addressed (delivery to mediators only or to final destinations) (see [29] for a detailed discussion). Further factors include questions of message integrity, security, and confidentiality.

Processing. The processing of a request/message can, correspondingly, be not reliable (*best effort*), or be guaranteed to be reliable (*atomic* and *transactionally coupled*) in case the delivery was reliable. For *atomic processing reliability,* the processing of the request/message is part of the transaction where the request/message was published. With *transactionally coupled processing reliability,* the processing is part of its own transaction (another transaction than the one where the request/message was published), and a dependency between the two transactions must be established.

The reliable communication models of delivery and processing raise the question of recovery and recoverability. Recovery can either be *forward (outgoing)* and/or *backward (incoming)* from the viewpoint of the publishing transaction. Factors that affect recoverability include the definition and determination of the set of recipients for which recovery must be executed. Likewise, the set of recipients that are to be included in the transactional scope for failure detection, the failure scope [29], needs to be defined.

Only few reliability guarantees are made for common communication models of existing middleware. For example, object request processing is only reliable w.r.t. processing and recovery if embedded in a predefined object transaction. Common messaging systems typically only guarantee reliable delivery of messages to mediators like queues, but not to sets of final recipients that make up a failure scope or recovery scope. The combination of object request processing and messaging today observes very limited or no reliability at all.

3. Transactions

Standard object interactions are limited in their flexibility of interaction models, due to limited variability in topology, binding, life-cycle dependency, and synchronicity (as defined above). Nevertheless, object interactions, when bracketed in transactions, offer atomicity and thereby support the building of reliable systems. Conversely, messaging systems and explicit middleware mediation allow for very flexible interaction models due to more variability in topology, binding, life-cycle dependency, and synchronicity. However, messaging systems do not offer a transaction processing and recovery model comparable to that of object transactions.

We believe that the high transactional reliability achieved with object transactions is not a contradiction to the weaker forms of reliability guaranteed for the more flexible interaction models of mediation-based messaging communication. Applications require and benefit from both communication models and reliability models, and a combination of the transactional object and messaging paradigm is very promising.

Middleware Mediated Transactions aim to achieve such a combination. The objective is to allow for more flexible interactions than fixed object interactions using mediator-based messaging, while retaining transactional reliability comparable to the one established for object transactions. Object requests should be carried out as conventional transactional object requests, while the integrated messages demand additional features and system support to address the concerns of atomic processing reliability and transactionally coupled processing reliability, as well as recovery.

Before introducing MMT in Section 4, we review the different styles of transactions of databases, object middleware and messaging middleware.

3.1. Database Transactions

The traditional transaction concept of databases encompasses the ACID properties (atomicity, consistency, isolation, durability) [4]. A transaction is the unit of reliable execution and brackets several operations on data items, such that the database state is transferred from one consistent state into another consistent state – possibly going through inconsistent intermediate states.

From a middleware technology perspective, a database provides *integrated transactions*, where client requests are shipped to the database and the transactional execution guarantees are monitored and enforced by the DBMS server. Integrated transactions are *closed* in the sense that all state manipulations are known to and are under control of the database and no communication to other components may be part of a transaction.

3.2. Distributed Object Transactions (DOT)

In contrast to stand-alone-DBMS systems, (object-oriented) TP-Monitors provide distributed transaction coordination [5,27,9]. While they also offer support for building scalable and robust systems (such as support for server activation, load balancing, management and others [13]) they all encompass a *distributed object transaction* service, which is responsible for managing the status and coordinated outcome of a transaction. This encompasses transaction context propagation and keeping a list of registered resources. And most important, the transaction service provides the coordination functionality to drive the two-phase commit processing, decide about commit or abort and realize its part of the recovery protocol.

Compared to the database ACID transactions, distributed object transactions resemble more a two-phase-commit service than an “ACID transaction” service, separating reliability and concurrency control concerns. Participating resources are obliged to follow the object transaction protocol and guarantee recoverability or durability with respect to the decision on transaction outcome.

Transaction services in distributed object systems are *open* in that they make no further assumption on the implementation of the participating resources. As long as an object provides the required interfaces and interacts in a 2PC conforming way it may be involved in a distributed transaction. Objects are characterized by their interface, whereas state and persistence of state are considered to be implementation concerns. The latter could be realized by traditional resource managers like databases or queues.

The same applies for concurrency control. Although concurrency control theory has been well researched for particular configurations like multi-database or multilevel systems [6,12,1], correctness criteria and guarantees for an overall enterprise application system seem to be hardly feasible, especially when execution autonomy and partial failures must be considered.

3.3. Message-oriented Transactions (MOT)

Another style of transaction processing addresses message-oriented middleware, namely *message-oriented transactions*. Enqueuing/dequeuing of messages and publishing/consumption of notifications is enclosed in a unit-of-work and dependent on the overall transaction outcome and vice versa. Only if the unit-of-work is committed, the messages will be sent out and consumed. The message mediator (for example, a queue manager) provides its own transaction manager and associated transaction demarcation API, or alternatively, may play the role of a transactional resource on behalf of an externally coordinated distributed transaction.

Both message-oriented transactions and distributed object transactions are accepted as common means to build cooperative information systems and to reduce the complexity of independent failure modes in distributed systems. Note however, that message-oriented transactions are fundamentally different from the other kinds of transactions discussed above. Message-oriented transactions group a set of messages that are to be published or consumed as a whole, and that may be included in the sphere of atomicity of a sender’s (consumer’s) execution. Message-oriented transactions do not group the consumer’s processing of data – as consequences of delivery or receipt of a message – within the sender's unit of work.

Therefore, a transactional messaging interaction typically spans multiple transaction contexts. Although the transaction contexts of sender and consumers are logically related, there is no middleware support to express and enforce such dependencies.

3.4. Comparing DOT and MOT

Distributed object transactions offer atomic processing reliability: the *execution* of a requested service is enclosed into a sphere of atomicity, i.e. the transaction is successful if the processing of the requested services, typically spanning multiple remote objects, is all successful. This has a deep impact on the life-cycle dependency between involved components, because all must be up during processing of the transaction. As a consequence, a message delivery failure (of e.g. an IIOP request message) typically will cause the transaction to fail. With respect to synchronicity, the termination model of current object transaction services assumes synchronous interactions, at least deferred synchronous, in the sense that the start of commit processing requires that beforehand all executions which contribute to the transaction outcome have been completed. This is a shortcoming compared to the termination model of peer-to-peer style interactions [5], and may be overcome when introducing asynchronous method invocation support as suggested in [25].

With message-oriented transactions, the goal is to *atomically publish* messages to remote participants and enforce processing thereof in an exactly-once-manner. The execution in response to the message receipt will not affect the outcome of the sender's transaction. While atomic processing reliability of producer and consumer is not guaranteed, the assumption is that eventually the message will be processed in a way expected by the application. In particular, the message will be processed not more than once, if the consumption is also transactional. Separating the execution of sender program and receiver program into multiple (depending on the number of receiver/subscribers) transactions is the key to the flexibility of mediated interaction. This approach provides for life-cycle independence and is considered one of the strengths of messaging middleware. Although this is a powerful concept, atomicity as provided by such transaction services has its drawbacks and deficiencies, especially when it comes to integrating both styles of transaction processing.

4. Middleware Mediated Transactions

Figure 1 shows a typical architecture where some components (`object1..object3`) interact through object interfaces on behalf of an object middleware transaction, and additionally, explicitly mediated interaction with further components is required, as exemplified by queue-based mediation to `recipient1` (and others) as well as publish/subscribe-based mediation to `consumer3` (and others).

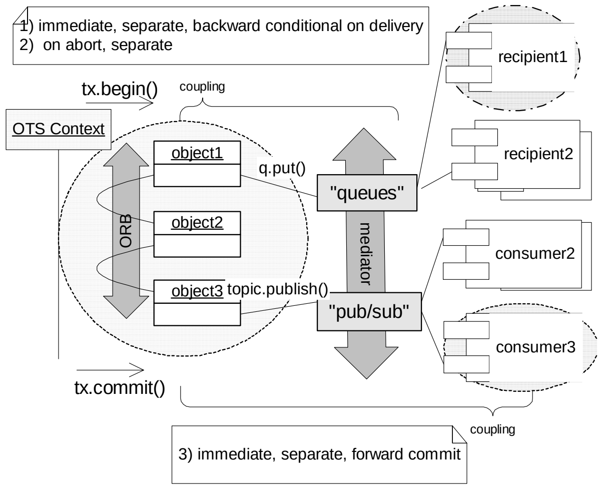

Figure 1. MMT Scenario

4.1. Sample Scenarios and Requirements

Assume that the recipients (`recipient1` etc.) are autonomous components, i.e., they do not cooperate in a 2PC (do not even expose a prepare state). Typically, there may be different recipients carrying out separate business functions. In a normal case, the message would *eventually* be delivered to and be processed by the recipients. However, from the point of view of `object1`, this may be too weak as a reliability. It is often required that some specified components will definitely receive the message, e.g. in a particular time frame, while the transaction is still ongoing (whereas for other recipients a fully time-independent delivery and processing may be adequate). With common messaging transactions on behalf of `tx`, the message will only be sent if `tx` commits and it will earliest be sent after commit of `tx`. In particular, it is not possible to make the `tx` outcome dependent on the acknowledgment of delivery to `recipient1` and furthermore, it is not possible to immediately sent out the message in parallel to the object transaction execution.

With MMT we suggest to allow *immediate message visibility* and to enforce a *backward dependency* of message delivery and/or processing on the outcome of `tx`, i.e., `tx` may only commit if the delivery/processing conditions are satisfied.

Assume, as another example, a cooperative workflow scenario, where `object1..object3` carry out computation steps on behalf of some process activity, and the `consumer3` component realizes a situation-aware worklist monitor [2,20]. Situation-awareness requires that an agent may monitor in a timely manner other processes and events in the environment, in parallel to the on-going process. The realization of the observer pattern (the implicit invocation) must be transaction-aware in the sense that notifications are displayed immediately, but the tentative nature of notifications must be taken care of. Using mediation through pub/sub preserves the decoupling in space.

The addition of MMT allows to immediately make such notifications visible and additionally enforce a *forward commit dependency* between `tx` and the transaction of the worklist manager. It will be enforced by MMT that the `consumer3` transaction may only commit if the triggering `tx` has committed. For example, a commit of `consumer3` might change the color of the worklist entry and save it in a persistent worklist area, whereas an abort may remove the entries. Notice that there is no 2PC synchronization between the object middleware transaction and the separate transaction of `consumer3`. In particular, notifications of `object3` are time-independent of `consumer3`, and a failure of processing in `component3` does not affect the success of `tx`, and thus does not impede the advance of the workflow process.

As can be seen from the examples, sending out messages immediately breaks the isolation sphere of the sender's transaction, if the transaction `tx` later aborts. This can be compared to a dirty read in CC theory [4]. Computations are externalized (by publishing notifications) that might be revoked later. With respect to serialization theory, publishing a notification can be compared to a write operation, while reacting to it introduces a read-from relation. Therefore, dirty reads – reactions to immediate visible notifications of not-yet-committed transactions – may lead to non recoverable reactions or necessitates cascading aborts. While in some cases it may be possible to revoke and compensate an already committed reaction, in other cases it might not be [18]. Weakening (communication) isolation in cooperative information systems often is considered to be a requirement and not as an error. Thus aborting a transaction is not only of local interest to the invoking component (i.e. transactional client) but to remote components that received "dirty notifications". As an implication, we require that MMT must provide a means to at least propagate the abort in order to allow compensating actions to take place where and when necessary. We suggest that the mediator provides a means to realize transactionally coupled processing reliability, which will be discussed next.

4.2. Coupling Modes

In order to provide the application developer with a flexible means to integrate mediated interactions in object middleware transactions we introduce the notion of a *coupling mode* (according to research carried out in the area of active object systems and active database management systems [10,8]).

Coupling modes determine the way that the message mediator relays notifications published in a producer's transaction to processing and/or delivery of the notification at the consumer. In particular, we suggest to distinguish the following properties that constitute a coupling mode:

- *Visibility*: the earliest point in time with respect to the producer's transaction status at which the mediator will relay a message to a consumer
- *Context*: the transaction context in which the message consumer component should execute its actions
- *Dependency*: the commit (abort)-dependency between the producer's transaction and the triggered action
- *Production*, *Consumption*: the impact of publishing and consuming messages on the transaction and the degree of delivery reliability

The following table enumerates different policies with respect to the coupling mode properties. We do not claim that this table is complete, nor that all policies are required in any case. The proposed coupling modes are introduced as a starting point and should undergo further discussion and evaluation in practice. The intention of the various policies are discussed below.

Visibility	immediate, on commit, on abort, deferred
Context	none, shared, separate
Forward Dependency	none, commit, abort
Backward Dependency	none, vital, mark-rollback
Production	transactional, independent
Consumption	on delivery, on return, atomic, explicit

Table 1: Coupling Modes

Visibility. With *immediate* visibility, messages are sent to consumers without waiting for the completion of the producer's transaction. Messages are sent out in parallel to the ongoing transaction without blocking and independent of the outcome of the triggering transaction.

In case of *on commit* (*on abort*) visibility, a consumer will receive the notification only if (and not before) the triggering transaction has committed (aborted). Note that abort visibility carries a weaker delivery guarantee than *on commit*, as a system crash may cause errors in sending out notifications.

With *deferred* visibility, the notification is propagated as soon as the producer starts commit processing. In a synchronous object transaction, this point in time is reached when all executions have completed.

Context. The transaction context may be propagated by the mediator and if the consumer component is willing to join the transaction, it may run in a *shared* context with the sender.

The mediator may also create a new transaction at the consumer's site and thereby establish a *separate* context. Or, the consumer may already run in a separate context in the first place.

In some cases, the mediator may not be able to influence the transaction management, or may not know about transactions at all.

Forward Dependency. A *commit* forward dependency specifies that the consumer's reaction may only commit if the sender's transaction commits.

An *abort* forward dependency specifies that the consumer's reaction may only commit if the sender's transaction aborts.

Backward Dependency. A backward dependency constrains the commit of the producer's transaction with respect to the success of delivery or processing at a consumer's site. If the recipient is *vital* coupled, the sender's transaction may only commit if the triggered transaction has been executed and completed successfully.

If the consumer is coupled in *mark-rollback* mode, the sender's transaction is independent of the consumer's transaction outcome, but the consumer application may explicitly mark the producer's transaction as *rollback-only.*

Both backward dependencies imply, that a failure of event delivery will cause the triggering transaction to abort. More fine-grained control of the impact of message delivery success (failure) on the sender – as mentioned in the example of the previous section – is provided by *conditional messaging* [31]. The sender may specify delivery conditions (e.g. regarding a time window for message receipt or processing by a defined set of recipients) which must be satisfied in order to allow the producer's transaction to commit.

Production. *Transactional* production specifies the well-known messaging transaction (unit-of-work) style, which makes the message delivery to the mediator dependent on the producer's transaction success and vice-versa.

With *independent* production policy, it is up to the application what should happen in case that the sending of a message fails. The application could abort the transaction but is not forced to do so.

Consumption. Once a notification has been consumed, the notification message is considered as delivered and will not be replayed in case the consumer crashes and subsequently restarts. The notification may be consumed simply by accepting the notification (*on delivery*) or when returning from the reaction (*on return*). Alternatively, consumption may be bound to the commit of the consumer's atomicity sphere (*atomic*) or be *explicit*ly indicated by the application at some point during reaction processing.

4.3. Discussion

Obviously, the defined policies are not independent of each other, not all combinations make sense, and some settings subsume others. Due to lack of space and to keep the paper at a reasonably level of abstraction, we will discuss some but not all issues that relate to the definition, semantics, realization and application of possible coupling modes.

If the reaction is coupled in a *shared* mode, it will execute on behalf of the sender's transaction. This implies a forward and backward dependency, which is just the semantics of a sphere of atomicity. This case is interesting for asynchronous mediated interactions, mostly in the sense of implicit invocation, where, for example, consistency constraints can be enforced (e.g. using immediate/deferred, shared, mark-rollback/backward commit). Thus, separation of concerns can be realized at component granularity.

Backward dependencies, even if implicit using a shared context, raise the difficulty of how to synchronize concurrently ongoing mediated computation and conditional delivery steps on behalf of an atomicity sphere. This is why checked transaction behavior must be enforced by MMT. A triggering transaction may not commit before the delivery conditions are satisfied and/or all reactions that have backward dependencies are ready to commit (and vice versa).

Mediated interaction in principle encompasses an unknown number of different consumers. Therefore, we require that the consumers that have backward dependencies and which need to synchronize with the triggering action must be selected by a predicate using conditional delivery or must be specified in a predefined group.

In cooperative information systems and EAI like settings, the reaction is typically executed in its own transaction context, i.e., a *separate context*, and the commit/abort dependencies to the triggering transaction can be established to structure the transactions dynamically.

A forward commit dependent reaction to the receipt of a message compares to a nested transaction structure, where a subtransaction may only commit through the top. It is different in the way that the "nested top" transaction of the consumer may be carried out time-independent (in particular after execution) of the sender's transaction.

An *abort* dependency allows to spin off exclusive alternative and contingency actions. Notice that chaining of abort dependencies may lead to inversion problems, which requires special treatment by the mediator (e.g. transitive establishing commit dependencies).

The on-commit (on-abort) visibility cannot be used in conjunction with a shared transaction context. Further, the on-commit (on-abort) visibility trivially implies a forward commit (abort) dependency.

Execution of the consumer's actions in a separate context may be achieved in different ways and with different degrees of flexibility and structuring power. The consumer may provide its own transaction context, which we then require to be controllable by the mediator conforming to the established dependencies (otherwise it is treated as context *none*). Or, the mediator creates a separate transaction context, which can be done i) on a per message basis or ii) in an aggregate manner, i.e., processing of several messages is grouped in one transaction.

Some couplings like forward commit/abort dependencies may not even be feasible, because one is faced with autonomous components that do not allow to control commit processing. If a forward commit dependency cannot be enforced, conservative on-commit visibilities may be used if the application can tolerate the delay and is not involved in a direct cooperation with the sender. Another approach is to let the consumer subscribe to on-abort visible messages (using immediate visibilities paired with dual subscriptions).

The latter can then be used to compensate dirty notifications in case that the sender's transaction fails.

On-abort visibilities are a very powerful and new concept in explicit middleware mediated interactions. Its potential use as a basis to build flexible, reliable, yet distributed exception handling seems very promising, especially in the context of "programming in the large" (also called "scripting objects" or "process enactment") [19]. Following the paradigm of separating control and data flow from implementation of (business) functionality requires a flexible interaction between components that implement activities and the process enactment engine which drives the process flow [19]. Additionally, programming in the large requires a *remote control* for scripting objects and a decentralized open architecture. Error handling cannot simply be achieved by atomicity of activity steps, which typically means backing out in case of error. Instead, appropriate means to externally interpret and react upon the situation of error are needed to facilitate forward recovery and progress of the overall process. Using notifications to monitor progress and abort-visible notifications as means of an activity execution-log seems to be a promising approach [28,22,15].

5. Service Support for MMT

Middleware Mediated Transactions extend the conventional distributed object transaction model by integrating explicitly mediated interactions provided by messaging middleware. MMT allow for additional flexibility in component interaction and enable to tolerate and support different forms of component autonomy.

Middleware-mediated transactions are programmed using combined object communication and mediator-based messaging communication. A *MMT middleware service* needs to provide a transaction service, a mediator-based messaging service, and features that realize MMT coupling modes as introduced above. MMT middleware services can be implemented as *integrated middleware service solutions*; services that integrate and extend existing transaction and messaging services already available with standard middleware. The integrated service can either introduce an indirection for use of the integrated transaction and messaging services, or expose the underlying service functionality and API directly to the application clients.

In this section, we briefly present two different MMT middleware services that have been developed independently of each other. The first service presented is the *Dependency-Sphere (D-Sphere)* service developed at IBM Research [30]. The second service presented is the *X^2TS* prototype developed at Darmstadt University of Technology [22]. These two services both support MMT with similar effects, but differ in their realization (in particular, the kind of mediation used), and, as a consequence, in the application use.

5.1. Dependency-Spheres

A *Dependency-Sphere (D-Sphere)* logically and operationally groups transactional distributed object requests and messages. The (synchronous) object requests occur as part of a conventional distributed object transaction, and the (asynchronous) messages are sent using standard messaging middleware [30].

D-Spheres are designed to accommodate a variety of object transaction models and messaging models. This includes the OMG OTS for standard distributed object transactions and the Long-Running Unit-of-Work transaction service (LRUOW) for long-lived transactions [3]. Messaging models supported include the Java Message Service (JMS) [16] and IBM's MQSeries [33].

The D-Sphere prototype focuses on systems that need to integrate Java2 Enterprise Edition (J2EE) applications with (JMS/MQSeries) message queuing applications. The D-Sphere service can be regarded as an additional layer above existing transaction and messaging middleware services, as illustrated in Figure 2.

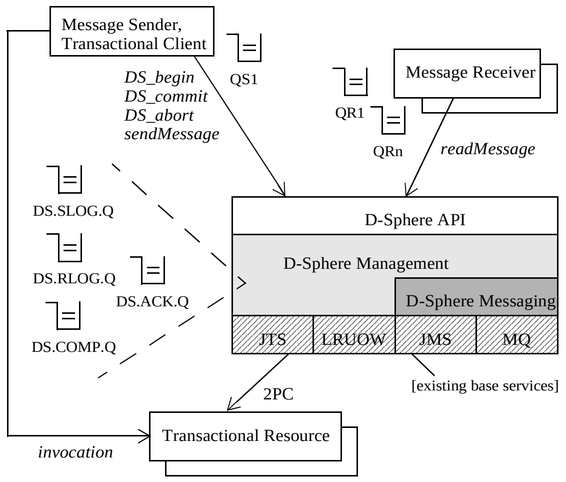

Figure 2. D-Spheres

A D-Sphere groups object requests on transactional resources and a set of *conditional messages* [31], i.e., messages to which diverse application-defined conditions for message delivery and message processing have been associated. These message conditions, such as different time constraints on delivery or processing for different recipients of the same message, are used to determine an outcome of success (or failure) for each D-Sphere message sent. This in turn affects the outcome of the D-Sphere as a whole including its constituent actions of object transactions, therefore allows to define and implement a variety of kinds of backward commit dependencies. D-Sphere conditional messages are sent out with immediate visibility as default, and final recipients who read messages from queues are associated to the D-Sphere. If a final recipient reads a message within his own D-Sphere transaction, the recipient-side transaction context is associated with the sender-side D-Sphere context.

The D-Sphere system provides its own API for a client to demarcate a D-Sphere, to create and send D-Sphere conditional messages, and to receive D-Sphere conditional messages. The D-Sphere system generates standard JMS/MQSeries messages for each conditional message, and provides additional functionality for monitoring message delivery and message processing, evaluating message conditions, and for taking actions such as the sending of success notifications or compensating messages dependent on the outcome of the evaluations.

Invocations on transactional objects are performed unchanged in the same manner as with the object transaction service selected (JTS or LRUOW). Object invocations implement a forward commit dependency.

The D-Sphere system uses persistent message queues and reliable message communication internally for purposes of logging (queue `DS.SLOG.Q` for send logs, `DS.RLOG.Q` for read logs), compensation support (`DS.COMP.Q`), and observation and evaluation of message condition satisfaction (respective violation) (`DS.ACK.Q`). These queues exist with the queue manager that a D-Sphere client uses for sending or receiving D-Sphere conditional messages.

The success (respective failure) of a D-Sphere is determined based on the outcomes of evaluation of its conditional messages and subsequently executing the conventional 2PC for all transactional object resources involved. In case of a D-Sphere success, all actions are committed. In case of a D-Sphere failure, object transaction rollback is enforced and compensation messages are (reliably) delivered to required message recipients.

The prototype currently implements the following:

- Messages are sent with immediate visibility (default). On commit and on abort visibilities are not yet supported.
- D-Spheres support the same propagation mechanism of the underlying object transaction service selected, but do not propagate object transaction contexts with messages. However, D-Spheres allow to associate separate (D-Sphere) transaction contexts of message recipients.
- Forward dependencies are supported for integrated object transactions and object requests.
- A large variety of kinds of (application-defined) backward dependencies is supported through the concept of conditional messaging.
- The production policy can be either transactional or independent, but the system employs in any case reliable messaging as guaranteed by the underlying messaging middleware.
- The consumption policies supported are on delivery and atomic. On return and explicit consumption policies will be supported in future versions.

5.2. X^{2}TS

X^{2}TS focuses on event-based publish/subscribe systems and their integration with distributed object transactions. It represents a combination of CORBA Transaction Service and CORBA Notification Service. X^{2}TS is implemented on top of a multicast enabled messaging middleware [32]. X^{2}TS supports indirect context management, implicit context propagation, interposition for distributed object transactions and the XA protocol for RDBMS resource management.

In our prototype, we only support the push-based interfaces with *`StructuredEvent`* one-at-a-time notifications. We assume that events (used as a synonym for notification) are instances of some defined type and that subscription may refer to specific types and to patterns of events.

The architecture of the combined transaction and notification service as provided by X^{2}TS is shown in Figure 3. A supplier creates, commits or aborts transactions through the use of the *`Current`* pseudo object. Notifications are then published on behalf of the current transaction context.

Subscription to patterns of events and coupling modes are imposed by the consumer by configuring the service proxy, i.e. setting properties through `QoSAdmin` and *`CompositorAdmin`*.

If any coupling modes are specified the couplings must refer to event types of the subscription (pattern declaration). We suggest that facades should be defined which simplify configuration by providing predefined coupling mode settings.

The prototype implementation currently supports the following MMT features:

- Deferred, on abort and on commit visibility.
- Currently we only support X^{2}TS managed transactions at the consumer. The consumer may provide its own transaction context, select a shared context or let X^{2}TS create a new transaction for each notification. Grouping of several notifications in one transaction can be facilitated by subscribing to an appropriate composite event.
- Checked transactions to support forward couplings and vital backwards
- Backward processing dependencies with the possibility to define groups of subscribers to be vital; X^{2}TS does currently not support conditional delivery
- Production policy is non-transactional, but can be explicitly programmed as transactional
- Consumption policies are on delivery, on return and explicit. Atomic consumption will be supported in a future version.

The design of X^{2}TS considers the fact that there may be a multitude of consumers with different couplings. While one consumer may be restricted to react to events *on commit* of the triggering transaction, others may need to react as soon as possible and even take part in the triggering transaction e.g. to check and enforce integrity constraints. Therefore, coupling modes, most important visibility, are configured on a per consumer basis. This has a significant impact on how visibilities and forward dependencies are realized. The basic idea is to always immediately and only once publish notifications. Additionally, transaction status changes are published (using a volatile EventReliability, only) on the message bus. The visibility guard component at the subscriber site enforces the visibilities by correlating transaction status change events and notifications appropriately. In order to deal with transaction and system failures, which may lead to lost status change notifications, we apply a graceful degradation mechanism, which in the worst case leads to polling the transaction status from the sender (details can be found in [22]).

The status change events are also used to enforce

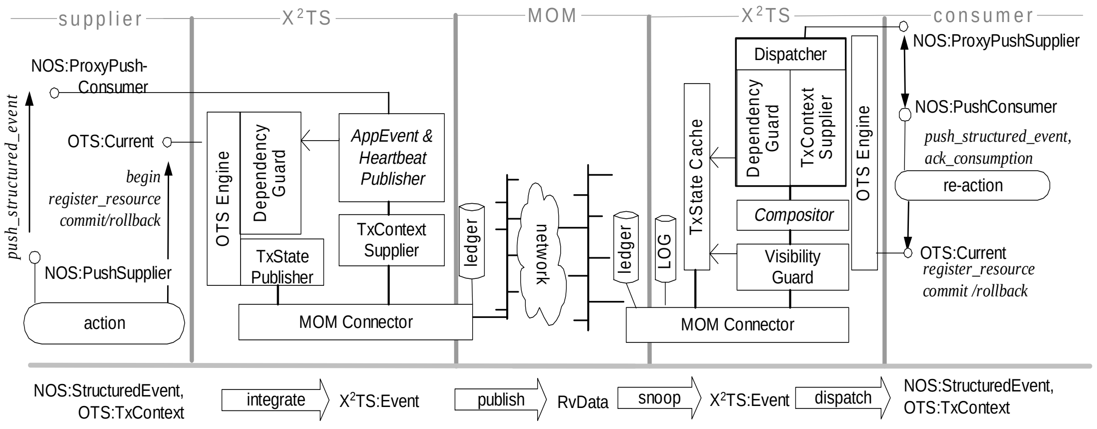

Figure 3. X^{2}TS

forward dependencies in a push-based manner and therefore preserve the loose coupling between sender and multiple subscribers.

Detecting patterns of events is realized by pluggable event compositors that are created by a specific *`CompositorFactory`*. We do not support the standard NOS filters but added our own proprietary *`Compositor`* interface. For event composition to cope with the lack of global time and network delays we introduce (in)accuracy intervals for timestamping and use supplier heartbeats [21]. Event composition may span several triggering transactions and we support different couplings with respect to multiple triggering transactions.

Reliable delivery and recovery of events leverages the persistent ledger of the underlying messaging middleware. In order to realize different consumption policies we had to implement our own acknowledgment and sequence number logging.

6. Conclusion

A major challenge of EAI is to achieve the reliable interoperation of diverse, autonomous components using standard middleware.

Reliability can be addressed through transactions, which provide the powerful abstraction of an atomic, indivisible execution of a set of actions. Transactions greatly reduce the number of possible errors an application programmer has to deal with. Two popular styles of middleware transactions are distributed object transactions and message-oriented transactions.

Component autonomy, i.e., variability in component-to-component topology, binding, life-cycle dependency and synchronicity, requires support for flexible interactions between components. Explicitly mediated interactions using mediators such as queues or publish/subscribe message services, as provided by messaging middleware, offer such interaction flexibility.

In this paper, we proposed *Middleware Mediated Transactions (MMT)* as an extension of the conventional distributed object transaction model. MMT integrate explicitly mediated interactions as provided by messaging middleware into standard object middleware transactions. MMT thus introduce additional flexibility of component interaction for object transactions.

We introduced the concept of *MMT coupling modes* which encompass visibility policies, context policies, dependency policies and production/consumption policies for transactionally interacting components.

MMT allow to publish messages from a distributed object transaction with immediate visibility as well as with transaction status dependent visibilities. The visibility policies suggested advance the state-of-the-art of conventional object and message transactions, where the visibility of messages is limited to the commit of the triggering transaction only and notifications cannot be sent out if the transaction aborts.

MMT uniquely allow to define spheres of reliable message delivery to, and of processing by, multiple messaging components that become transaction participants. With MMT, dependencies between multiple sender and receiver transaction contexts can be established. Furthermore, MMT support recovery through compensation for messages that were sent with immediate visibility. Also, MMT suggest flexible and modular exception handling – as supported by subscribing to notifications with on-abort visibility or forward abort dependency – which is again not possible with the transaction services as provided by middleware today.

We briefly presented two prototypes of new middleware services supporting MMT, the Dependency-Spheres service and the X^{2}TS service. Both services realize the idea of MMT employing queue-based messaging middleware in one case and publish/subscribe based middleware in the other.

Beside the area of EAI, we believe that process enactment using an event-driven approach will benefit from MMT services. Notifications for monitoring of process status, progress and errors, as well as provision of advanced situation awareness in collaborative workflow environments may be realized in a transaction-aware manner using MMT services.

7. References

[1] G. Alonso, A. Fessler, G. Pardon, and H.-J. Schek: Correctness in General Configurations of Transactional Components. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS'99)*, Philadelphia, Pennsylvania, USA, May 31-June 2, 1999.

[2] D. Baker and D. Georgakopoulos and H. Schuster and A.R. Cassandra and A. Cichocki. Providing Customized Process and Situation Awareness in the Collaboration Management Infrastructure. In *Proceedings of CoopIS'99*, Edinburgh, September 1999.

[3] B. Bennet, B. Hahn, 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. In *Proceedings of Middleware 2000*, Springer-Verlag LNCS 1795, 2000.

[4] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

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

[6] Y. Breitbart, H. Garcia-Molina and A. Silberschatz. Overview of Multidatabase Transaction Management. In *VLDB Journal*, 1 (2), 1992.

[7] M.L. Brodie and S. Ceri. On Intelligent and Cooperative Information Systems: A Workshop Summary. *Journal of Intelligent and Cooperative Information Systems*, 1(3), 1992.

[8] A.P. Buchmann. Active Object Systems. In A. Dogac, M.T. Szu, A. Biliris, and T. Sellis, (edit.), *Advances in Object-Oriented Database Systems*. Springer-Verlag, 1994.

[9] S. Cheung. Java Transaction Service (JTS). Sun Microsystems, Mar. 1999.

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

[11] L.G. DeMichiel, L.U. Yalcinalp, and S. Krishnan. Enterprise JavaBeans. Specification, Version 2.0, Sun Microsystems, JavaSoftware, May 2000.

[12] A.K. Elmagarmid (Edit.). *Database Transaction Models for Advanced Applications*. Morgan Kaufmann, 1992.

[13] I. Gorton. Enterprise Transaction Processing Systems. Longman, 1999.

[14] R. Guerraoui, R. Capobianchi, A. Lanusse and P. Roux. Nesting Actions Through Asynchronous Message Passing: the ACS Protocol. *Europ. Conf. on Object Oriented Programming (ECOOP'92)*, Springer-Verlag, June 1992.

[15] C. Hagen and G. Alonso. Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems. In *Intl. Conf. on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 1999.

[16] M. Hapner, R. Burridge, and R. Sharma. Java Message Service. Specification Version 1.0.2, Sun Microsystems, JavaSoftware, November 1999.

[17] S. Jajodia and L. Kerschberg (Edit.). *Advanced Transaction Models and Architectures*. Kluwer Academic Publishers, 1997.

[18] H.F. Korth, E. Levy, and A. Silberschatz. A Formal Approach to Recovery by Compensating Transactions. In Proc. *16th Intl. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990. Morgan Kaufmann.

[19] F. Leymann, and D. Roller. Workflow-based applications. *IBM Systems Journal*, 36(1), 1997.

[20] C. Liebig, B. Boesling, and A. Buchmann. A Notification Service for Next-Generation IT Systems in Air Traffic Control. In *GI-Workshop: Multicast - Protokolle und Anwendungen*, Braunschweig, Germany, May 1999.

[21] C. Liebig, M. Cilia, and A. Buchmann. Event Composition in Time-dependent Distributed Systems. In *Proceedings of CoopIS'99*, Edinburgh, September 1999.

[22] C. Liebig, M. Malva, and A. Buchmann. Integrating Notifications and Transactions: Concepts and X2TS Prototype. In *Proceedings EDO2000*, Springer-Verlag LNCS 1999, 2001.

[23] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *Communications of the ACM*, 36(9), 1983.

[24] Object Management Group (OMG). The Common Object Request Broker: Architecture and Specification, rev 2.2, Famingham, MA, May 1998.

[25] OMG. *CORBA Messaging*. OMG Document orbos/98-05-05, Famingham, MA, May 1998.

[26] OMG. Notification service specification. Technical Report OMG Document telecom/98-06-15, OMG, Famingham, MA, May 1998.

[27] OMG. Transaction service v1.1. Technical Report OMG Document formal/2000-06-28, OMG, Famingham, MA, May 2000.

[28] F. Ranno, S.K. Shrivastava, and S.M. Wheater. A system for specifying and coordinating the execution of reliable distributed applications. In *Intl. Working Conference on Distributed Applications and Interoperable Systems (DAIS'97)*, 1997.

[29] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings of Middleware 2000*, Springer-Verlag LNCS 1795, 2000.

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

[31] S. Tai, T. Mikalsen, I. Rouvellou and S.M. Sutton. Conditional Messaging in Enterprise Application Integration. May 2001. in submission.

[32] TIBCO Software Inc. TIB/ActiveEnterprise. www.tibco.com/products/enterprise.html, July 2000.

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