Transactions in Distributed Event-Based Middleware

Luis Vargas Jean Bacon Ken Moody

Computer Laboratory, University of Cambridge

{firstname.lastname}@cl.cam.ac.uk

Abstract

Distributed event-based middleware (DEBM) provides a basis for the interoperation of autonomous components in large-scale systems. Transactions ensure the atomic and reliable execution of operations that involve interconnected clients. Integrating transactions and DEBM is a challenging software design problem with important applications. This paper presents the Hermes Transaction Service (HTS), a service that supports event-driven applications with transactional requirements. HTS uses transaction-context propagation via event notifications, where the contexts of publishers and subscribers are interrelated within a mixed transaction. Clients may opt to be compensatable or non-compensatable. Optional anonymity of transaction participants is supported through hash-based set memberships.

1. Introduction

Middleware allows distributed heterogeneous components to interoperate. Message-oriented-middleware (MOM), such as IBM's MQSeries, allows for asynchronous interoperation of components. MOM has been used widely in Enterprise Application Integration (EAI) to decouple the execution of cooperative information systems. However, its message-passing model assumes closely controlled environments, by requiring senders and receivers to know the locations of a number of message queues in advance. In MOM, scalability is increased by distributing a set of message topics between a large number of servers or by means of server clustering. With no support for distributed (multi-server) message routing decisions, the model is point-to-point between a number of heavyweight servers, making its deployment infeasible for large-scale environments.

Distributed event-based middleware (DEBM) such as Gryphon [Strom98] and Hermes [Pietzuch04], help alleviate the difficulties of component interoperation in large-scale environments. Built around the notion of an event, i.e. a happening of interest in the system, and the publish/subscribe paradigm, DEBM achieves scalable event delivery between publishers and subscribers by distributing the publication/subscription matching process between a large number of participants (brokers). Its many-to-many interaction model makes DEBM particularly suited to support applications that must monitor and disseminate component updates from many publishers to an even larger number of subscribers. Examples of these applications are general situation assessment systems and business process (workflow) control systems. In such applications, the transactional notion of *all-or-nothing* semantics and the definition of dependencies between the producer of an event and its consumers are often needed. To our knowledge, no published work exists on the integration of transactions with a distributed (multi-broker) event-based middleware. The integration of transactions and messaging has been studied in the context of MOM (e.g. with MQSeries in D-Spheres [Tai01] and TIB/Rendezvous in X2TS [Liebig01]). But since DEBM will constitute an integral part of most future, large-scale application-integration scenarios, we argue the need for a framework that supports the transactional, yet flexible integration of autonomous components connected via DEBM. We have used a locally developed DEBM, Hermes, as a basis for the work presented here. We next describe a motivating example for transactions on DEBM. Section 2 gives background on transactions and their integration with messaging. Section 3 describes the Hermes DEBM and discusses the design of HTS. Its event-action model is described in terms of *coupling modes* and its use of *2PC* and *compensation* in *mixed transactions* is discussed. Then, the transaction service's architecture and supported protocol are presented. Section 5 discusses related work and Section 6 concludes the paper.

1.1. Motivating Example

Consider the implementation of a collaborative workflow that integrates a number of autonomous components, as depicted in Figure 1. A process to schedule meetings is defined where sets of invitation messages (notifications) are (content-based) routed from a publisher to a number of subscribers via the DEBM, according to their previously stated interests (subscriptions). A subset of all the recipients will accept the invitation and acknowledge it to the initiator, updating the process' distributed information record (e.g. the various participants' diaries), booking the required resources at the initiator, and finally confirming the meeting.

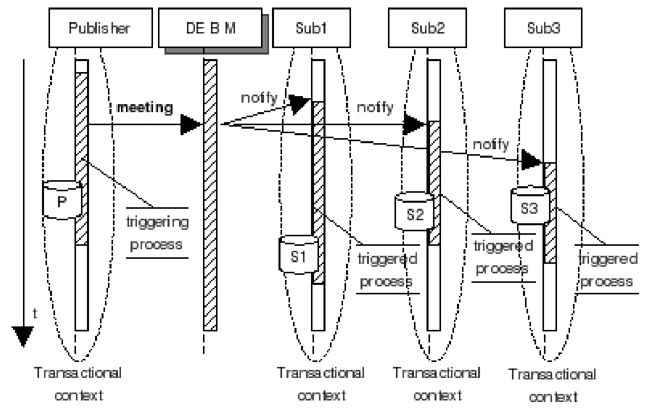

Figure 1 – A transaction on top of DEBM

Different requirements could be specified by the initiator e.g. the meeting might be conditional on a minimum number of participants. If the process is to be executed in an all-or-nothing manner, the notion of a transaction is required that allows the triggering, delivery, and processing of asynchronous event notifications to comprise an atomic unit-of-work.

2. Transactions

The concept of transaction is fundamental for database management systems. An ACID transaction provides a unit of reliable execution that brackets (atomically) a number of operations and ensures that all or none are carried out. Isolation of transactions preserves consistency of the database in that transactions move the database between consistent states.

2.1. MOM Transactions

In MOM, a notion of transaction exists where units-of-work (in MQSeries) or transacted sessions (in JMS) are used to group a set of messages for their atomic enqueuing/dequeuing. There, a message published in a MOM transaction is not sent until the MOM transaction commits, restricting the system to short transactions. Also, because MOM transactions always take place between a client (message producer or consumer) and a queue manager, no dependencies can be established between the publication of a message and its successful consumption by recipients and vice versa.

2.2. Distributed Transactions

In contrast to database and MOM transactions, transaction processing(TP)-monitors provide distributed transaction coordination. Distributed transactions, unlike ACID transactions, focus on the atomicity of operations across participants (resources), ensuring that they all receive a consistent view of the outcome of a transaction. This requires the TP-monitor to maintain a list of interacting resources and to make the commit/rollback decision by implementing an atomic commitment protocol such as two-phase commit (2PC). Any concurrency and reliability concerns, as well as durability guarantees with respect to the transaction outcome are left to the participating resources.

For integrating messaging with distributed transactions, current distributed transaction processing models (e.g. X/Open DTP [X/Open96]) only support the definition of *MQ-integrating transactions* via MOM, i.e. integrating message queues as resource managers into a distributed transaction. There, the enqueuing/dequeuing of messages is enclosed in a unit-of-work and made dependent on the overall distributed transaction outcome and vice versa. Only if the unit-of-work commits, are the messages available for consumption. [Tai00] identified various shortcomings associated with this model; in particular, its inability to associate the processing of a message by a consumer with the sender's transactional context. This often requires applications to maintain complex coordination logic to deal with errors in a transaction (e.g. due to unsuccessful processing of a message).

3. Hermes Transaction Service

First, we summarize the features of Hermes that are relevant to the Hermes Transaction Service. The design, architecture, and implementation of HTS is then discussed.

3.1. Hermes

Hermes [Pietzuch04] is a distributed, content-based publish/subscribe event-based middleware. It is built on Pastry, a peer-to-peer routing substrate to provide scalable event dissemination and fault-tolerance. A distributed event-based system implemented on Hermes consists of two kinds of components: *event brokers* and *event clients*. Event brokers form an application-level overlay network that propagates events. Event clients (publishers or subscribers) use the services provided by the broker network to communicate using events. Before publishing an event, a publisher advertises the associated *event type*. Subscribers specify their interests in (a subset of) these event types, via content-based subscriptions. Reverse path forwarding of subscriptions is used to create event dissemination trees from publishers to subscribers. Event notifications are delivered in FIFO order with respect to each publisher, and there is no total ordering in the case of multiple publishers on the same event type. End-to-end reliable delivery is an area of current work.

3.2 Hermes Transaction Service Design

HTS realizes an event-action model where the publication of events and the processing of event notifications by Hermes clients can be demarcated within a transaction. The outcome of a transaction and the effects of the processing of its notifications can therefore be made mutually dependent. This way, a transaction that is initiated by some triggering process at a publisher will be made to succeed only if the event notifications published within the transaction are successfully processed by the set of subscribers participating in the transaction. Correspondingly, the processing of a notification at a subscriber will be allowed to succeed only if the transaction associated with the notification is successful.

3.2.1. Coupling Modes. In the context of event-based interactions, [Buchmann94] introduced the notion of coupling modes and a set of constituent properties to determine the execution of triggered actions relative to the transaction in which a triggered event was published. Building on that notion, we now describe the different policies available in HTS with respect to *event notification visibility*, *transactional contexts*, and *publisher/subscriber dependencies*.

Unlike MOM, which supports *on commit* message visibility, HTS provides *immediate notification visibility*. With immediate visibility, published event notifications are delivered to subscribers in a non-blocking manner, allowing parallel activity within the system.

Event notifications published within a transaction include the transaction's context. Recipients of these notifications can decide whether to run in a *shared transaction context* with the publisher or whether to run in their own *separate context*. A shared context makes the processing of the notifications part of the transaction initiated by the publisher. A separate context consumes the notifications in the recipient's detached process.

A shared context implies *backward* and *forward dependencies* between the publisher of an event notification and the set of recipients. That is, the commit of the transaction triggering an event notification at a publisher is (backward) dependent on the success of the processing of the notification at every subscriber sharing the same transaction context, and vice versa (forward-dependency).

3.2.2. Mixed Transactions. Externalizing the effects of uncommitted transactions (e.g. via event notifications with immediate visibility) breaks the isolation of the triggering transaction at the publisher and may cause dirty reads (and reactions) by the notification consumers. Within a transactional context, coping with this requires not only that the failure in the processing of an event notification causes its triggering transaction to roll back; but also that, if the transaction fails after publishing event notifications, the processing effects of all these notifications be undone. There are two ways to ensure this. One is to require atomic commitment e.g. using two-phase commit (2PC), so that the triggering transaction at an event publisher is allowed to commit only after the triggered reaction at every relevant subscriber is prepared to commit and vice versa. However, relating clients of the DEBM by atomic commitment may sometimes not be desirable or even acceptable. One reason is that, with 2PC, a client exposes transaction control to other clients. If a client votes OK in response to a prepare request, the client has to be able to commit its local processing (e.g. hold locks) until instructed otherwise by the coordinator (which may be another client). Few services are willing to hand over transaction control in a loosely-coupled system. Another reason is that global transactions may be long-running. There are many factors contributing to this: long-lived business logic, delayed human input, network delays, etc. With 2PC, a client cannot commit until the distributed transaction can commit. Thus, a fast client may be forced to wait for a slow client.

Another (optimistic) way of dealing with dirty notification reads, while coping with the drawbacks of 2PC, is *compensation*. Using compensation as a recovery mechanism allows the triggered reaction at each subscriber to commit unilaterally, as an autonomous subtransaction, without waiting for the transaction coordinator's decision, with the promise that the effect of the subtransaction can be semantically cancelled afterwards, via a compensating transaction. Compensation is a generally accepted mechanism to deal with failures, fundamental to extended transaction models. However, not all transactions are compensatable. For example, transactions involving real actions are typically non-compensatable. In addition, for some clients, the cost of executing a compensating transaction may outweigh the costs of participating in a 2PC protocol.

For these reasons HTS adopts a flexible transactional model, with *mixed transactions* [Elmagarmid90], to accommodate both *compensatable* and *non-compensatable* clients within the same transaction. In a mixed transaction the constituent subtransactions which are compensatable may be allowed to commit before the global transaction commits, while the commitment of the non-compensatable subtransactions must wait for a global decision. When a decision is reached to abort a mixed transaction, the subtransactions in progress and the non-compensatable subtransactions waiting for a global decision are aborted, while the committed compensatable subtransactions are compensated. While the definition of compensating transactions is expected to be defined locally by each client, it is the responsibility of the transaction service to notify a compensatable client if the global transaction fails.

3.3 Hermes Transaction Service Architecture

HTS builds on two types of components to support transactions on top of Hermes: a *transaction manager* and a set of *client managers*. Each of these component types, depicted in Figure 3, exposes interfaces to: i) allow Hermes clients to access the transaction service functionality, and ii) establish contracts between the transaction manager and the clients (publisher and subscribers) involved in a transaction.

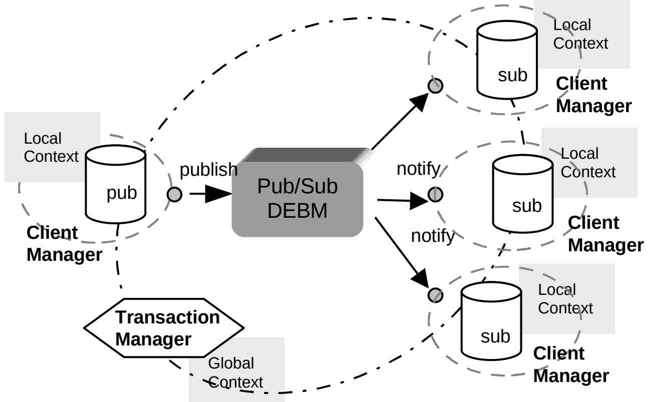

Figure 3 - Hermes Transaction Service

The *transaction manager* is the core component of the transaction service. It provides interfaces for transaction demarcation and the enlistment/delistment of transactional clients, and is responsible for propagating the transaction context with published event notifications. It also orchestrates the execution of the transaction service protocol; that is, the two-phase processing of mixed transactions and the acknowledgement of compensatable clients. By default in HTS, an event publisher takes the role of transaction manager for those transactions it initiates.

Client managers constitute the client-side component of the service. A client registers as a participant in a HTS transaction via its client manager. A client manager assumes different responsibilities depending on its client type. In the case of a non-compensatable client joining a transaction, the client manager is responsible for associating any work performed within the client's local transaction context (as a result of the processing of event notifications within the transaction) with the transaction's global context. At commit time, client managers of non-compensatable clients are informed by the transaction manager to prepare, commit, or rollback according to 2PC. For the case of compensatable clients joining a transaction, it is the responsibility of the client manager that, if the transaction aborts, any registered compensating function is executed to undo the effects of the client's local processing.

3.4. Hermes Transaction Service Protocol

The HTS protocol achieves the execution of mixed transactions using Hermes. As a service above Hermes, HTS builds on the asynchronous notification features of the underlying DEBM to publish, and deliver as event notifications, all the transaction-related messages (e.g. prepare, commit, and compensate) required by the transaction protocol. The semantic imposed by a shared transactional context requires that the participants of a transaction are known by the TM before the start of commit processing. For this reason, HTS adds an initial *census phase* to two-phase commit. Thus, the HTS commit protocol has three phases: *census*, *voting* and *decision*. In the census phase the TM determines the set of subscribers that will participate in a transaction initiated at some event publisher. In the voting phase, the TM is informed by the transaction participants, via their CM, whether the outcome of their local transactions was successful. Finally, in the decision phase, the TM determines a global outcome for the transaction, based on the participant replies. If the decision is to commit the transaction, the TM requests the set of non-compensatable clients to commit their local transactions, via their CM. Otherwise, if the global transaction is to be aborted, while non-compensatable participants are requested to abort their local transactions, compensatable participants are notified to execute a compensating action.

3.4.1. Census phase. The purpose of the HTS census phase is to determine the set of subscribers that will process a notification within a transaction, while still supporting optional client anonymity. The basic idea is to authenticate membership in a group of transaction participants, rather than identifying individual members for a transaction. We assume that CMs are able to generate a globally unique pseudonym *p* for every transaction they participate in, and that TMs and CMs know the same one-way hash function *H*.
On receipt of a transactional-aware notification with transaction id *xid*, a CM will evaluate whether the notification is to be consumed within the same transactional context at the subscriber, and if so, it will: 1) log the notification, 2) generate and remember a unique identifier *p*, 3) reply *join(xid, p)* to the TM, requesting to participate in the transaction *xid* with pseudonym *p*, and 4) Subscribe to transaction-related notifications for the transaction *xid*.
Based on the replies in the census phase, the TM will build a list of pseudonyms $l(xid) = [p_1, p_2, .., p_n]$ for subscribers participating in the transaction *xid*. Different application-defined conditions can be used to delimit the census phase (timeout *t*, min/max *n* subscribers, etc).
At the end of the census phase the TM will: 1) compute the hash *H(p)* of every pseudonym in $l(xid)$ into a list $Hl(xid) = [H(p_1), H(p_2), .., H(p_n)]$, and 2) publish *joined(xid, Hl(xid))* to notify the set of relevant subscribers about its participation in the transaction *xid.*
On receipt of *joined*, each CM will verify its membership in the list of transaction participants, computing the hash of its pseudonym, *H(p)*, and verifying its existence in *Hl(xid).* Notice that, as *H* is one-way, no pseudonym can be extracted from the published list. Having verified its membership in the transaction, the CM will establish the appropriate transactional context and pass the notification to the subscriber for its processing. Thereafter, the CM will receive either more notifications within the same transaction or a *prepare* notification. The CM will vote (commit or rollback) on the global transaction according to 2PC with the same pseudonym. Correspondingly, on receipt of any vote from a CM, the TM will verify the existence of the enclosed pseudonym in the list of transaction participants, in order to consider it. Different application-defined conditions could determine the transaction success (e.g. min *n* OK votes).

4. Related Work

X2TS [Liebig01] integrates CORBA's Transaction and Notification Services to provide transactional services above TIB/Rendezvous. X2TS is based on hierarchical subject-based addressing. Subdividing the message name space into subjects is inflexible, and may lead to subscribers having to filter events from general topics, or to the creation of large message hierarchies. Dependency(D)-Spheres [Tai01] focuses on grouping distributed object transactions and messages. The prototype is realized as an additional layer above the Java Transaction Service (JTS) and the MQSeries MOM. D-Spheres provides a flexible integration model for transactions above existing MOM. The model is statically based on a number of servers with no support for distributed message-routing decisions.

5. Conclusions

In this paper we presented HTS, a transaction service for Hermes, a scalable DEBM. HTS provides automatic transaction context propagation, via event notifications published with immediate visibility. It interrelates the transactional contexts of publishers and subscribers through a flexible model of mixed transactions. Optional anonymity of participants is supported via hash-based set memberships.

Acknowledgements. Thanks to Ehud Gudes and Lauri Pesonen for their valuable comments. Luis Vargas is supported by CONACYT Mexico.

References

[Buchmann94] A. Buchmann. Active Object Systems. In *Advances in Object-Oriented Database Systems.* Springer. Vol 130. pp 201-224.
[Elmagarmid90] A. Elmargamid, Y. Leu, W. Litwin, and M. Rusinkiewicz. A Multidatabase Transaction Model for Interbase. In *Proceedings of the 16th International Conference on Very Large Data Bases.* 1990. pp 507-518.
[Liebig01] C. Liebig, M. Malva, and A. Buchman. Integrating Notifications and Transactions: Concepts and X2TS Prototype. In *Proceedings of EDO'00.* Springer-Verlag LNCS 1999. 2000.
[Pietzuch04] P.R. Pietzuch. Hermes: A Scalable Event-Based Middleware. *University of Cambridge PhD Thesis and TR590.* 2004.
[Strom98] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. *IBM TJ Watson Research Center.* 1998.
[Tai00] S. Tai and I. Rouvellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proceedings of Middleware 2000.* Springer LNCS 1795. 2000. pp 308-330,
[Tai01] S. Tai, A. Totok, T. Mikalsen, I. Rouvellou, and S. Sutton Jr.. "Dependency-Spheres: A global transaction context for distributed objects and messages". In *Proceedings of EDOC'01.* IEEE Press. 2001.
[X/Open96] X/Open. "X/Open Guide Distributed Transaction Processing: Reference Model, version 3". 1996.