Transactions in Content-Based Publish/Subscribe Middleware

Luis Vargas* Lauri I. W. Pesonen* Ehud Gudes† Jean Bacon*
*University of Cambridge, Computer Laboratory {firstname.lastname}@cl.cam.ac.uk
†Ben-Gurion University, Computer Science Department ehud@cs.bgu.ac.il

Abstract

Content-based publish/subscribe provides a flexible communication model for component interoperation in large-scale environments. In process support systems and other applications that follow an event-based architectural style, the definition of dependencies between interacting components and the notion of all-or-nothing semantics are often needed to ensure reliable inter-component task execution. In this paper we introduce publish/subscribe (P/S) transactions as an abstraction to support these needs in content-based publish/subscribe middleware. A P/S transaction demarcates within an atomic unit-of-work, the production, delivery, and processing of a number of related asynchronous event notifications. A transaction service, provided by the middleware, realises P/S transactions to support the transactional execution of processes on behalf of applications.

1. Introduction

Middleware provides application-independent connectivity for component integration in distributed and heterogeneous environments. As software systems continue to be distributed over ever-increasing scales, transcending traditional geographical and organisational boundaries, the demands placed upon the supporting middleware infrastructure increase. Publish/subscribe [5] has emerged as a popular communication model to face the challenges imposed by component interoperation in large-scale environments. Built around the notion of an event, i.e. a happening of interest in the system, the model encompasses the mediated dissemination of information, via events (messages), between sets of event publishers and event subscribers. Its interaction pattern particularly suits the construction of systems that must react to situations of interest and which exhibit many-to-many interactions.

In publish/subscribe as implemented by Message-Oriented-Middleware (MOM), e.g. MQSeries [10] or the Java Message Service [18], clients publish and receive messages on a particular topic, via a message provider. Scalability is realised by distributing the set of topics across a number of message providers or by means of providers clustering. A different communication approach is the one taken by content-based publish/subscribe [16] middleware such as Gryphon [17] or Hermes [15]. In these, subscribers, connected by a network of brokers to publishers, specify sets of filters on the content of events. The publish/subscribe middleware is then in charge of routing events from publishers to the relevant subscribers across the network. High scalability is then achieved by distributing the event-filter matching process between a large number of brokers.

Transactions [6] are a commonly used approach to model and build reliable, fault-tolerant systems. A transaction provides a unit of reliable execution that atomically brackets a number of operations ensuring that all or none are carried out. Often, communication middleware is extended to support transactional semantics to make it easier for the programmer to write and deploy reliable applications [14]. Rather than interacting directly with involved resources (e.g. database systems) and developing transaction management functions, applications are built on top of the middleware. The transaction-aware middleware provides the abstractions that allow the programmer to focus on business logic instead of low-level transaction control.

In the case of MOM, a notion of transaction exists that allows the definition of message groups for atomic publication/consumption. MOM transactions however can not demarcate, for a set of messages, the outcomes and effects of their publication and consumption within the same transaction. As no dependencies can be enforced between the publisher and the consumer(s) of a message, reliable component interaction relies on a series of direct transactions, each between a component (publisher or consumer) and a message provider. Complex coordination logic may thus be required by applications in order to deal with different transactional failure cases [19] (e.g. due to unsuccessful processing of a message). Projects such as X2TS [13] or D-Spheres [20], have thus studied the integration of messaging and transactions to provide more flexible MOM transactional support. However, to our knowledge, no published work exists on the integration of transaction processing over content-based publish/subscribe middleware. We believe that such a middleware, targeted to the scalable interoperation of components in large-scale environments, should provide transactional support to applications, if it aims to be robust and widely applicable.

1.1. Motivation

A variety of process enactment and workflow management systems [12] are event-driven and exhibit a publish/subscribe style of communication. In these systems, components register their interest in specific situations originating from other components, and react to them according to some business process logic. For example, in process support systems [8, 4], clients instantiating activities from a common process, subscribe and react to change status events on that process, produced by other clients or by a process manager. To achieve the reliable and recoverable execution of tasks, often, these systems must introduce the notion of atomicity of operations (activities) and the enforcement of dependencies between interacting components [8, 11]. These systems are closed, in the sense that, they extend

existing process-enactment architectures with special exception and recovery mechanisms to model transactional requirements (e.g. according to a workflow language specification), rather than providing transactional event-services per-se as a function of the underlying middleware.

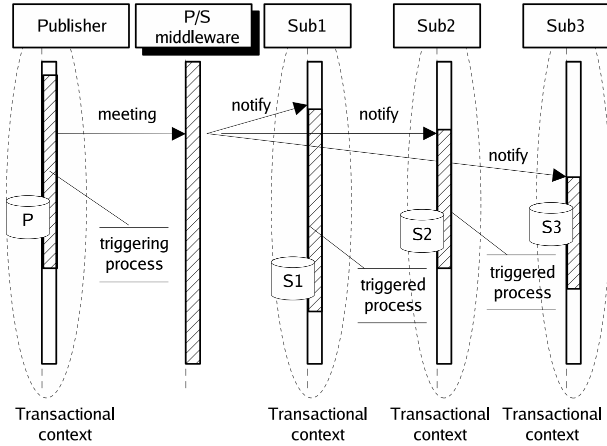


Figure 1. Event-based process enactment

Consider a collaborative workflow that integrates a number of heterogeneous components in a distributed system. Components in the system are only connected via a publish/subscribe middleware as depicted in Fig. 1. Each component is autonomous and maintains its own separate resources (e.g. databases) and business logic. Suppose a process to schedule meetings is to be defined where sets of invitation messages (i.e. meeting events) are (content-based) routed from publishers to subscribers, according to their stated interests (subscriptions). In response to an invitation, a subset of all the recipients will accept it and acknowledge it, updating the process' distributed information record (e.g. the various participants' databases S1, S2,.., Sn), booking the required resources at the initiator (database P), and finally confirming the meeting. Different requirements could constrain the meeting, e.g. to a minimum/maximum number of participants or to include those considered vital. If the process is to be executed in an all-or-nothing manner, a transaction model is required that allows the publication, delivery, and processing of events to comprise an atomic unit-of-work. Such a unit-of-work should thus interrelate the transactional contexts of the interacting components and enforce the required dependencies between them. We argue that such component-to-component reliability (and associated coordination) is not the responsibility of application developers and should instead be provided as an extended offering of the publish/subscribe communication middleware. In this paper we propose the use of publish/subscribe (P/S) transactions to support this offering in content-based publish/subscribe middleware, via a transaction service. Applications where the provision of publish/subscribe transactional services is useful range from general event-based process enactment systems to collaborative workflow management systems.

1.2. Contributions

In this paper we make the following contributions:

- We introduce P/S transactions as a new type of transaction context that interrelates the execution of clients in a publish/subscribe system by means of event publication, delivery, and processing operations. We discuss their failure model and combined use of 2-phase commit and compensation.
- We discuss a protocol that realises the execution of P/S transactions. The protocol accounts for the dynamic enlistment of participants in transactions that are advertised in the publish/subscribe system, and supports both compensatable and non-compensatable clients in the same transaction.
- We describe the architecture and programming interface of a transaction service that supports P/S transactions. We discuss the implementation of such a service in Hermes, a locally developed content-based publish/subscribe middleware. The service provides applications with the transactional execution of processes, allowing developers to focus on business logic rather than low-level transaction control.

2. Background

In this Section we establish the background on publish/subscribe as well as transactions in the middleware context.

2.1. Publish/Subscribe

Publish/subscribe [5] has emerged as a popular communication paradigm for large-scale distributed systems. In the publish/subscribe abstraction an event consumer subscribes to events of interest. Independently of any consumer, an event producer publishes events. If the event published by the producer matches the consumer's subscription, that event is asynchronously delivered to the consumer. This interaction is facilitated by a publish/subscribe middleware. The publish/subscribe middleware might be centralised as a single event broker node, or it might be decentralised as a network of broker nodes. Traditionally publish/subscribe middleware comes in two flavours: *topic-based* and *content-based*. In topic-based publish/subscribe, events are published under a topic and event consumers subscribe to that topic in order to receive those events. Topic-based subscriptions do not support filtering of events, i.e. the consumer will receive all events published on the given topic. In content-based publish/subscribe the subscription includes a filter expression which is applied to each published event. Events with content that matches the filter expression are delivered to the consumer.

In this paper we focus on the provision of transactional services in content-based publish/subscribe middleware. In particular, we discuss the Hermes Transaction Service (HTS), a transaction service for the Hermes [15] middleware. We believe however that the ideas presented here can easily be extended to other content-based publish/subscribe middleware, as well as to the more specific case of topic-based publish/subscribe. The main assumptions we make on the publish/subscribe middleware are reliable event delivery (i.e. a published message will eventually be delivered to the current subscribers) and a reply capability (i.e. a consumer is able to reply to the producer of an event). Reliable event delivery in a publish/subscribe system can be achieved via an event store-and-forward approach as in MQSeries [10] or by maintaining a knowledge model be-

tween publishers and subscribers with persistent storage at the publisher-side as in Gryphon [3]. Replies in a publish/subscribe system can be implemented in different ways [9]. The simplest option is to piggyback events with the location information of the event producer (e.g. a brokerID in the broker network) so that a consumer can reply to it directly. Another option is to log an event delivery at the intermediate brokers in the producer/consumer path and forward replies from consumers to the producer across the reverse delivery paths of the event.

2.2. Transactions

Transactions [6] are a commonly employed approach to model and build reliable, fault-tolerant systems. In a database system, an ACID (atomic, consistent, isolated, durable) transaction brackets a number of operations into an atomic unit-of-work, the execution of which transforms a database from one consistent state to another. To account for the reliable and atomic execution of operations in distributed scenarios, a number of architectures and communication paradigms have been extended to support transactional semantics [2, 14]. Of particular relevance to this paper are *transactional messaging* as provided by message-oriented-middleware (MOM) and *distributed transactions* as provided by transaction processing (TP-) monitors.

In transactional messaging, a *unit-of-work* in MQSeries [10] or a *transacted session* in JMS [18], is used to group a set of messages for their atomic sending and receiving from/to a message provider. From the sender's perspective, messages sent in a transaction are cached by the message provider and forwarded to consumers only after the sender's transaction commits. If a failure occurs or a rollback is issued by the sender, the messages are discarded. In the case of a receive, messages in a transaction are held by the message provider until the receiver issues a commit on its receive transaction. If a failure occurs or a rollback is issued by the receiver, the provider will attempt to redeliver the messages. This flexible mode of operation decouples the contexts of execution of senders and receivers. However, as messages are published only after the sender's transaction commit and because transactions take place only between a message provider and a client (sender or receiver), no dependencies can be structured between the sending of a message and the outcome of the message's processing. In particular, it is not possible to make the outcome of a client's transaction that sends messages as part of some application process, dependent on the outcome of the delivery and processing of those messages by the set of recipients that participate in the process.

Distributed transactions, unlike ACID transactions and transactional messaging, focus on the atomicity of operations across multiple resources (e.g. database systems), ensuring that they all commit or abort according to the transaction outcome. This requires a TP-monitor to maintain a list of participating resources and to direct the execution of an atomic commitment protocol such as *two-phase commit* (2PC). The goal is to ensure the atomicity of operations. Any concurrency and durability concerns with respect to the transaction outcome are left to the participating resources.

For integrating messaging with distributed transactions, current distributed transaction models only support the definition of *message provider (MP)-integrating transactions*, i.e. integrating a message provider as a resource in a distributed transaction. Here, message transmission or reception can be included in the transactional context of a client, together with other operations (e.g. database updates). Clients then interact with each other, via MOM, using a series of direct sender-to-MP and MP-to-receiver transactions for each recipient of a particular message. As in transactional messaging, no way exists to include within the same transaction, the receivers' consumption process for messages published inside the transaction. Because no dependencies between the producer and the consumers of messages can be defined and enforced by the middleware, applications must implement potentially complex logic for coordinating the effects and outcomes of message delivery and processing operations, and for returning to a consistent state in case of failures [19].

3. P/S Transactions

We define a publish/subscribe (P/S) transaction as a new type of transaction where the contexts of execution of a set of transactional clients are interrelated by means of event publication, delivery, and processing operations. Two types of clients are involved: one event publisher and one or more event subscribers. The P/S transaction demarcates within an atomic unit-of-work: 1) a process triggering one or more events at a publisher, 2) the set of triggered events and 3) a set of processes that are executed by the consumption of these events at the subscribers. The P/S transaction either succeeds or fails as a whole (see Section 3.1), making the outcome of the unit-of-work and the effects of the processing of enclosed events mutually dependent. As (un)successful event consumption determines a P/S transaction's outcome, events are published with immediate visibility (i.e. before the triggering processes commit). Externalising event publishers' computations in this way allows parallel activity in the system, as multiple subscribers can process events in the transaction at the same time. To deal with possible *dirty reads* (i.e. the consumption of events which triggering processes later abort) at the subscribers, the effects of processed events reflecting these computations can be revoked according to 2PC processing or compensated (see Section 3.2). For simplicity through the rest of the paper we will refer to a P/S transaction simply as a transaction.

3.1. Failure Model.

Within a transaction, every event carries the transaction context and thus it defines (with the rest of events in the transaction) its atomicity. The transaction's outcome (success or failure) is thus affected by the outcome of the publisher's triggering process and the outcome of every contained event. We distinguish between two types of events: *protocol* and *application* events, and define the outcome for individual events across two dimensions: *delivery* and *processing*. The first type of events corresponds to middleware-level events used to realise the transaction protocol described in Section 4. The second type are application-defined events used to convey any information between clients in the publish/subscribe system. If a transaction succeeds then all (protocol and application) events published as result of the transaction have been both successfully delivered

and processed by the set of relevant subscribers (i.e. those participating in the transaction). Correspondingly, if the delivery or processing of any event fails, the transaction is aborted.

How event delivery failures are detected is closely related to the type of reliability mechanism used by the publish/subscribe middleware. In general, the evaluation of an event delivery's outcome describes a "worrying-parent-model" [20]: if no acknowledgement by a recipient is received after some scope in time, the event is declared as failed. For example, in Gryphon, errors in the delivery of an event (e.g. due to a broker crash or a link outage) are detected by the lack of recipients' acknowledgements at intermediate brokers in the event delivery path and propagated to the publisher. In the context of a transaction, different conditions could be associated with the unsuccessful delivery of an event: the transaction could be immediately aborted or *n* delivery attempts could be made before aborting it. In Section 5.6 we discuss how we deal with the detection of failed protocol and application events in a transaction.

Event processing failures are caused by *system* and *application exceptions*. System exceptions range from an invalid pointer (e.g. NULL) operation to a database connection that cannot be obtained. Application exceptions are user-defined exceptions thrown after unexpected business logic situations (e.g a lookup method cannot find an object). While some of these exceptions will require a client to immediately abort a transaction, others can be handled transparently by the client. For an event, we thus consider its processing unsuccessful if any unhandled (system or application) exception is thrown while it is processed. We also provide clients with a declarative method (see Section 5.3) to explicitly abort a running transaction.

3.2. 2PC and Compensation

We distinguish two ways of dealing with dirty reads (and reactions) by event consumers. One is to require atomic commitment using two-phase commit (2PC), so that the triggering process at an event publisher is allowed to commit only after the triggered process at every relevant subscriber is prepared to commit and vice versa. However, relating clients by atomic commitment may sometimes not be desirable. 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 transaction manager. Another reason is that transactions may be long-running because of long-lived business logic, delayed human input, etc. With 2PC, a client cannot commit until the global 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 is compensation [7]. Using compensation as a recovery mechanism allows the triggered process at each subscriber to commit unilaterally without waiting for the transaction manager's decision, with the promise that its effect can be semantically cancelled afterwards, via a local compensating process. Compensation is a generally accepted mechanism to deal with failures, fundamental to extended transaction models and workflow systems [12]. However, not all operations are compensatable. For example, processes involving real (*pivot*) actions are sometimes non-compensatable [6]. In addition, for some clients, the cost of executing a compensating process may outweigh the costs of participating in a 2PC protocol. For these reasons we adopt a model following *flexible transactions* [1], to accommodate both compensatable and non-compensatable clients within the same transaction. In a flexible transaction, the processes of participating compensatable clients are allowed to commit before the global transaction commits, while the commitment of processes of non-compensatable clients must wait for a global decision. When a decision is reached to abort the transaction, the processes in progress and the processes of non-compensatable clients waiting for a global decision are rolled-back, while the committed processes of compensatable clients are locally compensated.

4. Transaction Protocol

A transaction consists of three phases: 1) a *census phase* for registering the set of subscribers participating in the transaction, 2) a *transaction phase* consisting of events published as a single transaction by a transaction manager on behalf of an event publisher, and 3) a *commitment phase* which is used to either commit or abort the transaction among the transaction manager and the participant clients.

4.1. Census Phase

The semantic imposed by a P/S transaction requires that participants of a transaction are known by the transaction manager before the start of the transaction phase. Furthermore, we need to guarantee that only those subscribers that are accepted as participants of a transaction can act (e.g. vote) upon it. This is the purpose of the census phase.

In this phase, a transaction manager advertises a new transaction, on behalf of a publisher, by publishing a `census` event with the name of the transaction (e.g. "meeting"), a unique transaction identifier (tx id), and other optional attributes describing the transaction (e.g. "subject" and "time"). Subscribers have registered their interest in `census` events where the name and attributes match a transaction that they are interested in. As part of a registration each subscriber also locally specifies a (non-) compensatable client type. The `census` event is then delivered to all subscribers with the appropriate registrations, informing them of the tx id of a new transaction.

We rely on the use of pseudonyms and *hash-based set memberships* to build authenticated groups of transaction participants. The basic idea is to verify membership in a group of transaction participants, rather than identifying individual members for a transaction. We assume that 1) subscribers can generate a globally unique pseudonym p for every transaction they participate in, and 2) the transaction manager and all subscribers know the same one-way hash function H.

Subscribers who are interested in taking part in a transaction reply to its related `census` event with a `census_reply` event that contains: 1) the tx id, 2) a randomly generated pseudonym p, and optionally, 3) the subscriber's identity (e.g. public key). Notice that the subscriber's (non-) compensatable client type is not included in this event, as the abort mechanism used by a subscriber is only of self-relevance. Based on the subscribers' replies the transaction manager will form a list

of pseudonyms $l(tx\,id) = [p_1, p_2, .., p_n]$ for the subscribers interested in tx id. At some point in time the transaction manager must decide to stop waiting for more participation replies from the subscribers. This decision might be based on a timeout, on the number of replies received, or on conditions defined by the publisher who initiated the transaction. e.g. a transaction manager might wait for replies for 30 seconds or 10 subscribers, or it might wait for specific participants. The nondeterminism of the census phase makes it necessary to inform the subscribers of the list of of accepted participants. For this, at the end of the census phase, the transaction manager will: 1) compute the hash H of every pseudonym p in $l(tx\,id)$ into the list $Hl(tx\,id) = [H(p_1), H(p_2), .., H(p_n)]$ and 2) publish a `2PC` event with `state=begin` containing $Hl(tx\,id)$.

On receipt of this event, each subscriber will verify its membership in the transaction by computing the hash H of its pseudonym p, and verifying its existence in $Hl(tx\,id)$. Notice that, as H is one-way, no pseudonym can be extracted from the published list, allowing only subscribers accepted as participants of a transaction to act upon it. Also, the use of pseudonyms allows subscribers to remain anonymous in processes that don't require disclosing the participants' identities. Having verified its membership in the transaction, the subscriber will then subscribe to events with the given tx id in order to receive events related to this specific transaction.

We leave publishers to define the *scope* of transactions as private or public. In a private transaction, events published as part of the transaction are delivered only to the participants determined in the census phase. Contrarily, events published in a public transaction are dispatched not only to the transaction participants but also to every subscriber who has specified a subscription matching the event content. This allows non-participant subscribers to still receive events of interest while consuming them in their own separate execution contexts.

4.2. Transaction Phase

In the transaction phase the transaction manager publishes events, on behalf of a publisher, inside the transactional context created in the census phase. Published events are tagged with 1) the tx id, 2) the transaction manager's contact information, 3) a public/private flag, and 4) a sequence number. This allows us at the subscriber-side to: 1) relate event consumption to a particular transaction, 2) reply to the transaction manager according to the transaction protocol, 3) not forward events from private transactions to a subscriber who is not a participant, and 4) detect delivery failures for application events in the transaction. Subscribers are able to reply to each published event with an abort event `2PC_reply(vote=abort)`. I.e. if an event causes an error at one of the subscribers that would cause the subscriber to vote abort in the commitment phase, the subscriber is able to vote abort immediately. This allows the transaction manager to abort the transaction immediately without wasting time and resources in publishing the remaining transactional events and going through the commitment phase.

4.3. Commitment Phase

The transaction manager ends the transaction by initiating the commitment phase on behalf of the publisher. As in *flexible transactions*, this phase consists of a vote among the participating subscribers deciding whether to commit or abort the transaction. For this, the transaction manager publishes a `2PC` event with `state= prepare`, the tx id, and the last sequence number published in the transaction. The subscribers vote by replying to this event with a `2PC_reply(vote=commit)` or a `2PC_reply(vote=abort)` event. While compensatable clients are allowed to commit at this time, non-compensatable clients must wait for a global decision on the transaction. The transaction manager publishes the result of the vote as a `2PC` event with `state=commit/abort` depending on the results of the vote. Votes from subscribers that are not in the list of participants are silently ignored. The participating clients are responsible for committing or aborting the transaction according to the voting result. Aborting, depending on the subscriber's client type, will involve either rolling back or compensating any work done as part of the transaction.

5. Transaction Service

The idea of publish/subscribe transactions is supported by a transaction service. In this section we first describe Hermes and then we discuss the architecture and service support offered by HTS, its transaction service. HTS realises the transaction protocol described in the previous Section while supporting:

- explicit transaction demarcation at the publisher side.
- automatic transaction context propagation, via event notifications published with immediate visibility.
- census handling functions to support dynamic groups of transaction participants.
- implicit transaction demarcation at the subscriber side.
- automatic management of involved resources.
- support for compensation mechanisms.

5.1. Hermes

Hermes [15] is a distributed, content-based publish/subscribe event-based middleware built on a peer-to-peer routing substrate. A distributed event-based system implemented on Hermes, depicted in Figure 2, consists of two kinds of components: *event brokers* and *event clients*.

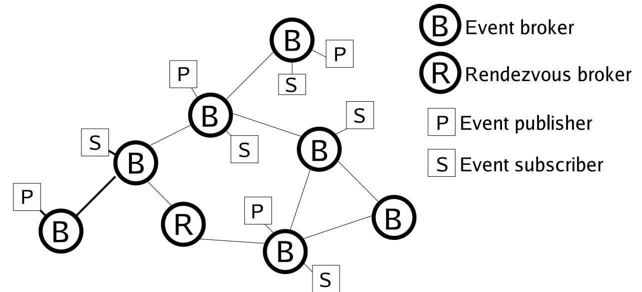


Figure 2. Hermes event-based system

Event brokers form an application-level overlay network that propagates events. Event clients (*publishers* or *subscribers*) use the services provided by the publish/subscribe broker network to communicate. For this they connect to a local broker, which then becomes *publisher-hosting (PHB)*, *subscriber-hosting (SHB)*, or both. Brokers without connected

clients are called *intermediate brokers*. Before publishing an event, a publisher must advertise the associated event type. As a result of this action, routing state from the publisher towards an intermediate broker, known as the *rendezvous broker*, is created. The address of this broker is computed from the event type name using a distributed hash table (DHT) algorithm. Subscribers specify their interests in (a subset of) these event types, via content-based subscriptions. Reverse path forwarding of subscriptions, towards the corresponding rendezvous broker and any related advertisements are used to create event dissemination trees from publishers to subscribers. Event notifications are delivered in FIFO order with respect to each publisher.

5.2. Architecture

HTS introduces an additional layer of abstraction above the standard Hermes middleware to provide an integrated view of transaction and event notification services for transactional application development. The different HTS components at the publisher and subscriber-side are depicted in Figure 3. The deployment of these components is discussed in Section 6.3.

A publisher request a `TxManager` to begin, commit or abort a transaction, via a `PSTransaction` object. As part of the context associated with the transaction, the publisher can publish one or more events and access local resources (see Section 5.4). The `TxEngine` component of the `TxManager` orchestrates the execution of the transaction protocol. For this, it uses the services of two other components, the `CensusCoordinator` and the `TxEventPublisher`. The first establishes censuses for new transactions according to the publisher's conditions. The latter attaches the transaction context to events received from the `publisher` or generated by the `txEngine` in response to transaction state changes (e.g. prepare/abort). It piggybacks to each event: the tx id, the `TxManager`'s contact information, a public/private flag, and a sequence number. Events are then passed to the `Hermes connector` to publish them in the publish/subscribe broker network. A persistent `log` is used by the `TxEngine` to store: the tx id, information about publisher's accessed resources, the list of transaction participants, and the state of the transaction.

A subscriber registers his interest in participating as a (non-) compensatable client in a transaction with the `TxEvent-Subscriber` component, via a `registration` object. According to this registration, the `TxEventSubscriber` is in charge of 1) setting an appropriate subscription on the corresponding `census` event in the publish/subscribe broker network, via the `PSConnector`, 2) replying to a `census` event using a randomly generated pseudonym, and 3) subscribing to transaction-related events for those transactions the subscriber joins. The `TxEventSubscriber` thus allows a subscriber to participate in specific transactions without having to deal with low-level details such as pseudonyms and the tx id.

At the subscriber, all events are received through the `Hermes connector`. From these, those carrying a transaction context are pushed to the `TxEventGuard`. The functions of this component are: 1) to log the participation of a subscriber in a transaction and 2) to verify that events from private transactions are pushed forward only if the subscriber participates in the transaction. Events from the `TxEventGuard` are subsequently passed to the `TxEventDispatcher`. For each event this component will then establish the appropriate transaction context for a consumption process to run (according to the registration), associating its execution's thread with the tx id of the running transaction. For compensation purposes (See Section 5.5), consumption of events on behalf of the transaction is logged by the `TxEventDispatcher`. Implicit access to the transaction's context is made available through the `TxCallback` interface and used to locally register resources or abort a running transaction. The current state of a transaction is accessed and updated via the `TxStateHandler` component, according to 1) the outcome of the delivery and consumption processes for events in the transaction and 2) the reception of transaction status events published by the `TxManager`. Finally, the `TxCompensator` is in charge of executing compensating processes to undo the effects of a failed transaction. A persistent `log` is used to store information about those transactions the subscriber joins. For each transaction we store: the tx id, information about accessed resources, the state of the transaction, and the set of consumed events.

5.3. Application Programming Interface (API)

The purpose of the transaction service's API is to allow the publication and consumption processes of events to be demarcated within a transaction. In general, developers implementing publish/subscribe applications that handle specific event types need the definition of these event types at development time. The use of transactions also requires that event types published as part of a transaction are known. The definition of a transaction then includes: 1) the definition of a `census` event type `TxType` used to advertise the transaction and 2) the definition of each event type `evType` published in the transaction.

With the proposed API, publishers explicitly demarcate the scope of transactions that are advertised via census events. On receipt of a `census` event, a subscriber can decide whether to join a transaction; in which case, the consumption process for any received events within the transaction is implicitly considered as part of the transaction. The API of HTS for both types of publish/subscribe clients is presented in Tables 1 and 2.

advertise(TxType)
t = new PSTransaction(TxType)
t.begin(txType, scope, conditions)
t.publish(evType)
t.register_resource(resource)
t.commit(), t.abort()

Table 1. HTS publisher API

A publisher uses `advertise` to state its intention to publish transactions of a certain type `TxType`, where `TxType` is defined by a name (e.g. "meeting") and a set of application-defined attributes that describe the transaction (e.g. subject, date, time). The publisher starts a new transaction by: 1) creating a `PSTransaction` object associated with an already advertised transaction type and 2) requesting the transaction manager to start a transaction instance of this type (e.g. meeting["DSOnline", "15/03/06", "15:00"]) using `begin`. The

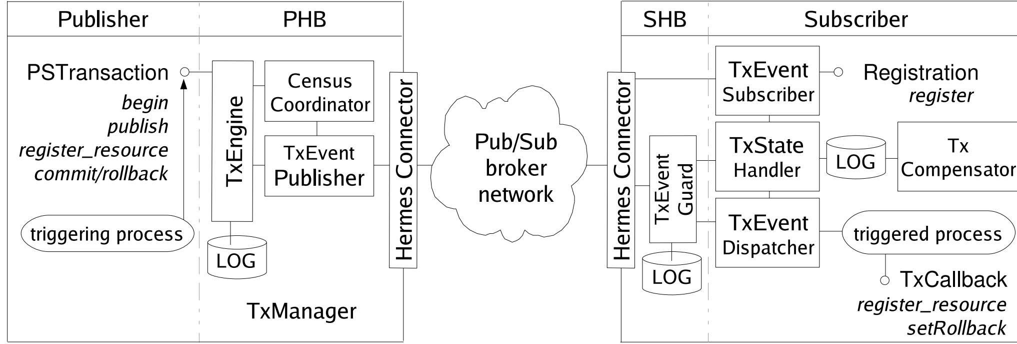


Figure 3. HTS architecture

publisher specifies with the request the scope of the transaction (i.e. public/private) and a set of census conditions (e.g. min 10 and max 20 participants). As part of the transaction, the publisher can publish one or more events of the same or different type `evType` using `publish` and register local resources (e.g. databases), via `register_resource`. Depending on the application scenario, events published in the transaction could just extend the information conveyed in the `census` event or provide a basis for structuring interrelated data (e.g. a `census` event change[processID] followed by multiple changeDetail[componentID, changeDescription] events). Finally, a publisher requests the transaction manager to commit or abort a transaction using `commit` or `abort`.

r = Registration(clientType, TxType, filter, census_callback) r.addHandler(evType, evT_callback) r.addCompensationHandler(evType, evT_c_callback) r.execute() CensusCallback.join() TxCallback.register_resource(resource) TxCallback.setAbort()

Table 2. HTS subscriber API

Subscribers specify their interest in a particular transaction type `TxType` (e.g. “meeting”), via a `registration`. A registration is created by specifying a (non-) compensatable client type, an optional filter on the content of `TxType` (e.g. subject=“DSOnline”), and a callback class `census_callback` implementing the `CensusCallback` interface. It is in this callback that census events for transactions of interest will be delivered, and where the subscriber programatically requests to join a transaction, via the `join` method. As part of the registration, handlers for the different event types published within the transaction are specified using `addHandler`. Each handler specifies an event type `evType` and a callback class `evT_callback` implementing `TxCallback`. It is within this callback, that events of the specified type will be processed as part of the transaction associated with the registration. Compensatable clients should also specify, using `addCompensationHandler`, a compensating function `evT_c_callback` to undo the effects of events of type `evType` that are processed as part of the transaction. Finally, the registration is activated using `execute`.

Contrary to the explicit transaction demarcation used by the publisher, the scope of a transaction at the subscriber is implicitly defined by the set of registered `evT_callbacks`. That is, the start and end of each callback triggered in response to an event mark the boundaries of the execution context that is associated with the running transaction. Implicit access to this context is made available through the `TxCallback` interface and used to register local resources, via `register_resource`. For each individual event, we consider its consumption successful if its triggered callback returns without errors. An unhandled exception within the callback will immediately rollback the transaction at the subscriber and send an abort vote to the transaction manager. We also provide a `setAbort` method that a subscriber can use to declaratively abort the transaction.

5.4. Resources Support

As part of a transaction, middleware clients (publishers and subscribers) are expected to interact with local resources (e.g. database systems). We provide clients with the `register_resource` method to enlist a resource in a running transaction. While at the publisher `register_resource` is supplied by the `PSTransaction` object, at the subscriber it is supported via the `TxCallback` interface. Calling `register_resource` results in the transaction service logging, at the client, the fact that a resource was used as part of the transaction. This association, as well as status information about the resource (e.g. `is_prepared`) are maintained during the lifetime of the transaction. The transaction service at the client-side can be seen as an interposed coordinator between a local resource used by a client and the transaction manager. Instead of having every resource directly involved with the transaction manager, at each client the transaction service merely collects votes from locally involved resources and passes a general decision to the transaction manager, which then decides on the overall outcome of the transaction. Afterwards, the transaction manager sends the commit/abort decision to the participant clients, so that at each client, the transaction service commits, rollbacks, or compensates work on the involved resources.

We expect that clients interact with different resources, via standard (e.g. *XA* [21]) adapters. In our current prototype, we incorporate an XA-enabled data adapter to integrate PostgreSQL database systems as recoverable resources in a transaction. The XA specification defines an interface that allows work executed on a resource to be associated with a transaction that can be prepared and committed or rolled-back according to 2PC. Through the adapter, operations on a PostgreSQL database are associated with the tx id of the current transaction. Isolation of operations in the transaction with regards to other

database operations is a responsibility of the database. At the commitment phase, the database is asked to prepare the work associated with tx id for commit. If the prepare fails for any reason, the transaction service is informed via the adapter, rolling back any work done at the client as part of tx id and sending an abort vote to the transaction manager. Finally, when a decision from the transaction manager is received, work associated with tx id in the database is committed, rolled-back, or compensated by the transaction service.

5.5. Compensation

In the case that a transaction fails, recovery for events processed as part of the transaction must be performed at each compensatable subscriber. Compensation relies on the idea that for some task executed as part of a transaction, a corresponding compensation task can be designed. While compensation tasks are expected to be locally defined by a developer, according to the application semantics, the middleware must support the mechanisms for doing so. For compensation purposes, HTS provides clients with 1) automatic logging of events consumed as part of a transaction and 2) the possibility of defining a compensating (callback) function for each event type in the transaction. The set of compensation functions defined by a client for a transaction constitute the transaction's compensation task.

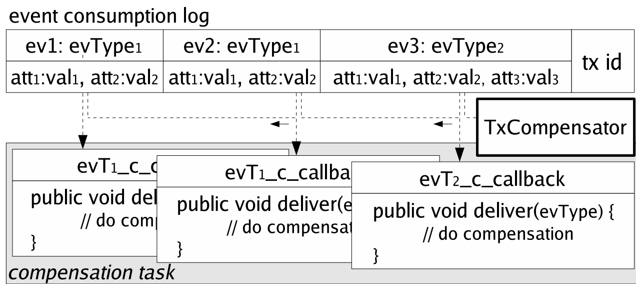


Figure 4. HTS compensation process

The compensation process, depicted in figure 4, works as follows. After a subscriber receives a `2PC(state=abort)` event for the transaction tx id, the `TxCompensator` component will inspect the local `log` and execute, for each event ev_i consumed in tx id, a defined compensating function `evT_c_callback`. Compensating functions are selected according to each event type and executed in the inverse consumption order of events. As part of the compensation process, updates to local resources may be required, e.g. to undo committed operations on a database. Ensuring the idempotency of this process requires that these updates are performed atomically. For a single resource this is enforced by the `TxCompensator` using a local transaction. If multiple resources must be updated, the `TxCompensator` creates a new transaction and associate all updates on these resources with it. Atomic commitment between the resources is driven by the `TxCompensator` using standard 2PC. In both cases, the execution of the compensation process remains local and does not involve the `TxManager`.

5.6. Failure Detection and Recovery

The failure model introduced by publish/subscribe transactions requires that the failed delivery of events published as a result of a transaction are identified. Even if the publish/subscribe middleware provides reliable delivery, an event could take an arbitrarily long time to arrive at its destination, e.g. due to communication failures. For this reason we introduce additional mechanisms to deal with the detection of failures in the delivery of 1) protocol and application events published by the `TxManager` and 2) reply events sent by the transaction service at the subscriber-side.

At the transaction level, we identify failures in the delivery of application events by making the `TxEventPublisher` piggyback sequence numbers to published events and effecting a subscriber-side verification at commit time. For this, the `TxEventPublisher` includes in the transaction's `2PC(state=prepare)` event, the last sequence number in the set of published application events. Upon detecting a missing event, the `TxStateHandler` at the subscriber will rollback its local work and vote abort on the transaction.

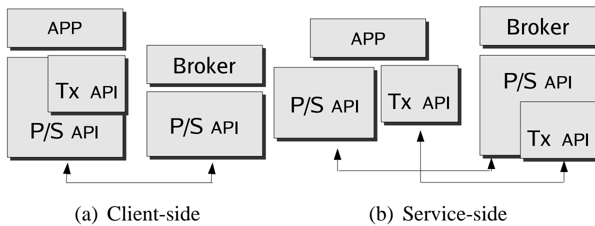
In traditional distributed transactions, a common approach to achieve eventual progress in the case of lost protocol messages is to let participants query each other about the current state of the transaction [2]. In publish/subscribe transactions however participants' contact information is not made available to each other nor to the `TxManager`. i.e. all communication between parties in the system is is not direct but via the publish/subscribe middleware using events. To deal with the detection of failed protocol events we leverage the functionality of the publish/subscribe middleware as follows: At the subscriber we feed timeouts to the `TxStateHandler` component. If for a running transaction, the timeout for an expected protocol event is exceeded, the `TxStateHandler` will publish a `2PC` event with the corresponding tx id and `state=in_doubt_`*`current_state`*; where *current_state* is the subscriber' state at the time of the timeout. Possible in-doubt states are: *census*: the subscriber has requested participation in the transaction and is waiting for a `2PC(state=begin)` event. *transaction*: the subscriber is in the transaction phase and is waiting for a `2PC(state=prepare)` event. *commitment*: the subscriber is in the commitment phase and is waiting for a `2PC(state=commit/abort)` event. After other subscribers taking part in the transaction receive the event, their local `TxStateHandler` will check whether it has information concerning the in-doubt state; in which case, it will reply to the in-doubt subscriber's `2PC` event with a `2PC_reply` event enclosing the relevant state information (e.g. the participants list for the census state or the transaction's outcome for commitment phase). After any reply is received, the `TxStateHandler` of the in-doubt subscriber will act accordingly. Notice that this mechanism also allows participants to overcome blocking in the case that the `TxManager` fails after publishing a `2PC(state=commit/ abort)` event.

At the `TxManager` we must deal with the failed delivery of reply events at the commitment phase. Note that a failed reply event at the census phase will only result in the related subscriber not taking part in the transaction. Although a failed abort reply in the transaction phase will result in unnecessary work done, an abort outcome will still be determined by having the subscriber reply his vote at the commitment phase. We must also consider that the execution of some

subscribers' processes may not have completed by the time the publisher's `commit` call is issued. For this, we feed timeouts to the `TxEngine` for reception of replies. If within a timeout one or more reply events are not received in response to a `2PC(state=prepare]` event, the publisher's `commit` call will raise an `UncheckedTransactionBehaviour` exception. The publisher can then decide to either abort the transaction at this time, or wait and retry at the next timeout.

6. Implementation

The transaction service can be implemented either a) at the client-side, as a library on top of the publish/subscribe service offered by a publish/subscribe middleware, or b) at the service side, as an additional middleware service alongside the publish/subscribe service. Both approaches, depicted in Fig. 5, have their advantages and disadvantages.



(a) Client-side (b) Service-side

Figure 5. Implementation approaches

6.1. Client-Side Implementation

A client-side implementation of the transaction service consists of a layer between the publish/subscribe API and the application. The application is free to access both the publish/subscribe and the transaction service APIs (tx API). The transaction service then relies purely on the services exported by the publish/subscribe API: `publish` and `subscribe`. This allows us to extend the transaction service to any existing publish/subscribe middleware.

From an application's point of view it is important that both transactional and non-transactional events are identical. I.e. an application should be able to use any event type both inside and outside transactions. Therefore the tx API cannot use a specific attribute for storing the tx id in an event. In order to attach a tx id to an event the tx API will have to wrap the event inside an instance of another event type which defines an attribute for the tx id. We can define a global event type, `TxWrapper`, to wrap events published by an application inside a transaction. The wrapper type consists of two attributes: the tx id and the wrapped event. The transaction service at the publisher-side receives an event from the application, wraps it inside a new instance of `TxWrapper`, and sets the tx id attribute to the id of the current transaction. At the subscriber end the tx API unwraps the event and delivers it to an application that has subscribed to the given transaction based on the tx id. The downside of the wrapping approach is twofold: 1) non-transactional subscribers will not see the wrapped event, and 2) in a DHT-based publish/subscribe system, like Hermes, all transactional events will be routed through the same *rendezvous* broker.

6.2. Service-Side Implementation

By implementing the transaction service at the service-side, i.e. as an additional offering of the publish/subscribe middleware, we are not limited to using the basic publish/subscribe API. This allows the transaction service to add *meta-attributes* to published events. That is, the publish/subscribe service can add an attribute to an event which is not part of the event type. At the subscriber end the transaction service will then remove that attribute before delivering the event to a subscriber's process. The tx id can thus be added to an event as a meta-attribute. At the publish/subscribe service level this means that we are dealing with the published event instead of a wrapper. Therefore this implementation approach does not suffer from any of the disadvantages of the client-side implementation: 1) events can be delivered to both transactional (based on the tx id) and non-transactional (based on the event content) subscribers, and 2) events will be routed though the rendezvous broker of the event's type. The obvious downside is that changes to the existing publish/subscribe system are required.

6.3. HTS Implementation

We have implemented the HTS architecture (see Section 5.2) following a service-side approach. In our current prototype, we realise the propagation of transaction context with published events at the local broker of a Hermes publisher (PHB), via the `TxEventPublisher` component. In addition, at this broker we have located the other two components of the `TxManager` (i.e. the `CensusCoordinator` and the `TxEngine`). These three components interact, as part of the Hermes service's process at the broker, to provide transactional services to the publisher. Separating transaction management functions from the publisher is appealing as: 1) we are able to detect crashes in the publisher's process at the `TxManager` (using periodic *heartbeats*), so that, if a transaction's outcome has not been defined, we can immediately abort it; and 2) we can instantiate transaction coordination functions at brokers that are part of a trusted domain, rather than at possibly untrusted clients.

At the subscriber-side, we have located the `TxEventGuard` at the local broker of a subscriber (SHB). Through this component we: 1) realise the functionality of private transactions by discarding events from private transactions that a subscriber does not participate in; and 2) detect crashes in a subscriber's process, so that, if no vote on a transaction has been generated by the subscriber, an abort vote can immediately be sent to the `TxManager`. The rest of components (`TxEventHandler`, `TxStateHandler`, `TxEventDispatcher`, and `TxCompensator`) provide functionality that requires close interaction with the subscriber application's process. We have thus colocated them with the subscriber, as part of the services exported by Hermes.

7. Related Work

The provision of middleware-mediated messaging systems that are transaction aware has been a recent subject of study. In [19], the shortcomings of MP-integrating transactions were identified. *Message delivery* and *message processing* transac-

tions were suggested as alternative integration strategies. Their focus was on integrating distributed object requests with MOM within the same client transaction. Publish/subscribe transactions share some of the general considerations taken by these strategies, most notably sending messages at any point in a transaction. Our focus is however on the reliable execution of processes in a publish/subscribe system. Two projects are of particular relevance to this paper: the DAOS project and its X2TS prototype [13], and the Dependency-Spheres service [20].

The X2TS prototype integrates the CORBA Object Transaction Service (OTS) and the CORBA Notification Service to provide transactional services above TIB/Rendezvous, a multicast-enabled messaging middleware. X2TS, as HTS, provides implicit context propagation with messages. Atomic message grouping is however not supported. Different visibilities for a message with regards to a transaction (e.g. on commit) can be configured at the consumer-level. This is enforced by publishing all messages with immediate visibility, like HTS, but caching them at the consumer until the specified time. Atomic commitment is handled by OTS via 2PC with no support for extended transactional mechanisms such as compensation. Implemented on top of a message bus architecture, X2TS is based on topic-based addressing. Subdividing the message space into topics may lead to subscribers having to filter messages from general topics, or to the creation of large message hierarchies.

The Dependency-Spheres service focuses on operationally grouping distributed object transactions with transactional messaging. The prototype is realised as an additional layer above the Java Transaction Service (JTS) and MQSeries. While JTS' 2PC is used to drive the commit of transactional objects, compensation messages are sent to cancel enqueued messages of failed transactions. Descriptive consumer-to-producer dependencies can be defined using conditions on the delivery and processing of particular messages. Dependency-Spheres provides a flexible integration model for transactions above existing MOM. However, with no support for distributed message-routing decisions, the producer-consumer interaction model is point-to-point, statically defined across a number of message queues that act as intermediators.

8. Conclusions

A major challenge of large-scale cooperative information systems is the reliable interoperation of heterogeneous and autonomous components. Content-based publish/subscribe middleware helps in facing part of this challenge by providing a flexible communication model between publisher and subscriber components. In this paper we have introduced publish/subscribe (P/S) transactions as an abstraction to support the reliability needs of applications that involve components connected via a content-based publish/subscribe middleware. A P/S transaction demarcates, within an atomic unit-of-work, the production, delivery, and processing of a number of related asynchronous event notifications. P/S transactions are realised by a transaction service provided by the middleware. The service is in charge of enforcing the required dependencies between the transactional contexts of interacting components and ensuring the atomicity of operations. This allows application developers to focus on business logic instead of component-to-component reliability (and associated coordination) and low-level transaction control. Applications where the provision of publish/subscribe transactional services is useful range from general event-based process enactment systems to collaborative workflow management systems.

Acknowledgements. Luis Vargas is supported by CONACYT. Lauri Pesonen is supported by EPSRC (GR/T28164).

References

[1] W. Litwin A. Elmagarmid, Y. Leu and M. Rusinkiewicz. A Multidatabase Transaction Model for InterBase. In *Proc. of the 16th Int. Conf. on Very Large Data Bases*, 1990.

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

[3] S. Bhola, R. Strom, S. Bagchi, Y. Zhao, and J. Auerbach. Exactly-once Delivery in a Content-based Publish-Subscribe System. In *Proc. of the 2002 Int. Conf. on Dependable Systems and Networks*, pages 7–16, 2002.

[4] C. Collet, G. Vargas-Solar, and H. Grazziotin-Ribeiro. Open Active Services for Data-Intensive Distributed Applications. In *Int. Database Engineering and Application Symposium*, pages 349–359, 2000.

[5] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[6] J. Gray and A. Reuter, editors. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[7] E. Levy H. Korth and A. Silberschatz. A formal Approach to Recovery by Compensating Transactions. In *Proc. of the 1991 ACM SIGMOD Int. Conf. on Management of Data*, 1991.

[8] C. Hagen and G. Alonso. Beyond the Black Box: Event-based Inter-Process Communication in Process Support Systems. In *Proc. of the 19th Int. Conf. on Distributed Computing Systems*, pages 450–457, 1999.

[9] J. Hill, J. Knight, A. Crickenberger, and R. Honhard. Publish and Subscribe with Reply. Technical report, University of Virginia, 2002.

[10] IBM. *WebSphere MQ Application Programming Guide 6.0*, 2005.

[11] N. Krishnakumar and A. Sheth. Managing Heterogeneous Multi-system Tasks to Support Enterprise-Wide Operations. *Distributed and Parallel Databases*, 3(2):155–186, 1995.

[12] F. Leyman and D. Roller. *Production workflow: concepts and techniques*. Prentice-Hall, 2000.

[13] C. Liebig, M. Malva, and A. Buchmann. Integrating Notifications and Transactions: Concepts and X2TS Prototype. In *Proc. of the 2nd Int. Workshop on Engineering Distributed Objects*, volume 1999, pages 194–214. Springer-Verlag LNCS, 2001.

[14] Q. H. Mahmound, editor. *Middleware for Communications*. Wiley, 2004.

[15] P. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, 2004.

[16] D. Rosenblum and A. Wolf. A Design Framework for Internet-Scale Event Observation and Notification. In *Proc. of the Sixth European Software Engineering Conf.*, pages 344–360. Springer–Verlag, 1997.

[17] 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. In *Proc. of Int. Symposium on Software Reliability Engineering '98*, 1998.

[18] Sun Microsystems. *Java Message Service Specification 1.1*, 2002.

[19] S. Tai and I. Rovellou. Strategies for Integrating Messaging and Distributed Object Transactions. In *Proc. of Middleware 2000*, volume 1795, pages 308–330. Springer-Verlag LNCS, 2000.

[20] S. Tai, A. Totok, T. Mikalsen, and I. Rouvellou. Dependency-Spheres: A Global Transaction Context for Distributed Objects and Messages. In *Proc. of the 5th Int. Enterprise Distributed Object Computing Conf.*, pages 105–115. IEEE Press, 2001.

[21] X/Open. *Distributed Transaction Processing Ref. 3*, 1996.