(c) Copyright 2001 Hewlett-Packard Company

A framework for implementing business transactions on the Web

Hewlett Packard Arjuna Labs,
Newcastle upon Tyne, England

Submitter: Mark Little (mark@arjuna.com)
(c) Copyright 2001 Hewlett-Packard Company
The Submitter(s) offer this document for consideration for incorporation into an OASIS specification, and acknowledge(s) that the requirements of OASIS IPR policy section 3.1 relate to this contribution.
Other Contributors: Dave Ingham (dave.ingham@arjuna.com), Savas Parastatidis (savas@arjuna.com), Jim We bber (jim.webber@arjuna.com), Stuart Wheater (stuart.wheater@arjuna.com)

1. Introduction

An increasingly large number of distributed applications are constructed by being composed from existing applications. The resulting applications can be very complex in structure, with complex relationships between their constituent applications. Furthermore, the execution of such an application may take a long time to complete, and may contain long periods of inactivity, often due to the constituent applications requiring user interactions. In a loosely coupled environment like the Web, it is inevitable that long running applications will require support for fault-tolerance, because machines may fail or services may be moved or withdrawn. A common technique for fault-tolerance is through the use of atomic transactions, which have the well know ACID properties, operating on persistent (long-lived) objects [1]. Transactions ensure that only consistent state changes take place despite concurrent access and failures.

1.1 Problem statement

Traditional transaction processing systems are sufficient if an application function can be represented as a single top-level transaction. Frequently this is not the case. Top-level transactions are most suitably viewed as “short-lived” entities, performing stable state changes to the system; they are less well suited for structuring “long-lived” application functions (e.g., running for minutes, hours, days, …). Long-lived top-level transactions may reduce the concurrency in the system to an unacceptable level by holding on to resources (e.g., locks) for a long time; further, if such a transaction aborts, much valuable work already performed could be undone [2].

Several enhancements to the traditional flat-transaction model have been proposed. One enhancement (supported by the Object Transaction Service [3]), permitting a finer control over recovery and concurrency, is to permit *nesting* of transactions [4]; furthermore, nested transactions could be concurrent. The outermost transaction of such a hierarchy is typically referred to as the top-level transaction. The permanence of effect property is only possessed by the top-level transaction, whereas the commits of nested transactions (subtransactions) are provisional upon the commit/abort of an enclosing transaction. This allows for failure confinement strategies, i.e., the failure of a subtransaction does not necessarily cause the failure of its enclosing transaction. Resources (e.g., locks) acquired within a subtransaction are inherited (retained) by parent transactions upon the commit of the subtransaction, and (assuming no failures) only released when the top-level transaction completes, i.e., they are retained for the duration of the top-level transaction.

Nested transactions allow the construction of modular applications: the builder of an object can use transactions within its methods if those methods need to be transactional, and if they are subsequently invoked from within another transaction they will simply be nested. They also allow for fault containment: if a failure occurs within a subtransaction then it can be rolled back without forcing the enclosing transaction to roll back. This latter property may be particularly useful within a loosely coupled environment such as the Web.

Another possible enhancement is to introduce *type specific concurrency control*, which is a particularly attractive means of increasing the concurrency in a system. Concurrent read/write or write/write operations are permitted on an object from different transactions provided these operations can be shown to be non-interfering (for example, for a directory object, reading and deleting different entries can be permitted to take place simultaneously). Object-oriented systems are well suited to this approach, since semantic knowledge about the operations of objects can be exploited to control permissible concurrency within objects [5]. Additional work may be needed when working with procedural systems.

In addition, although strict two-phase locking protocols are typically implemented by transaction systems to ensure that locks are (apparently) released instantaneously when the transaction terminates, this is not a requirement. As long as failures do not occur that would cause the transaction to rollback, locks can be released early. However, if the transaction rolls back, it may cause a cascade rollback scenario, where other transactions that have acquired these early released locks are told to rollback as well.

Note, that traditional transaction systems tie persistence and concurrency control together at the level of the database. This is not a transaction requirement. In object-oriented systems, typically persistence and concurrency control are separated, allowing finer granularity locking to occur.

Structuring certain activities from long-running transactions can reduce the amount of concurrency within an application or (in the event of failures) require work to be performed again. For example, there are certain classes of application where it is known that resources acquired within a transaction can be released "early", rather than having to wait until the transaction terminates; in the event of the transaction rolling back, however, certain compensation activities may be necessary to restore the system to a consistent state. Such compensation activities (which may perform forward or backward recovery) will typically be application specific, may not be necessary at all, or may be more efficiently dealt with by the application. It is important to realise that these types of application can be implemented using traditional transaction systems, but simply in a less efficient manner than may be possible given other techniques.

The following discussion is background describing some typical applications where extended transaction structuring techniques may be beneficial. For simplicity we assume that all applications require some form of durability, but this is not necessarily the case, and transactions can still be used to control non-durable state changes.

1.1.1 Arranging a night out

Consider the following long running business transaction, illustrated by Figure 1. The application activity is concerned with booking a taxi (*t1*), reserving a table at a restaurant (*t2*), reserving a seat at the theatre (*t3*), and then booking a room at a hotel (*t4*). If all of these operations were performed as a single transaction (shown by the dotted ellipse), then resources acquired during *t1* would not be released until the top-level transaction has terminated. If subsequent activities *t2*, *t3* etc. do not require those resources, then they will be needlessly unavailable to other clients.

Long-running applications and activities can be structured as many independent, short-duration top-level transactions, to form a long-running business transaction. This structuring allows an activity to acquire and use resources for only the required duration of this long-running transactional activity. Therefore, as shown the business transaction may be structured as many different, coordinated, short-duration top-level transactions.

(c) Copyright 2001 Hewlett-Packard Company

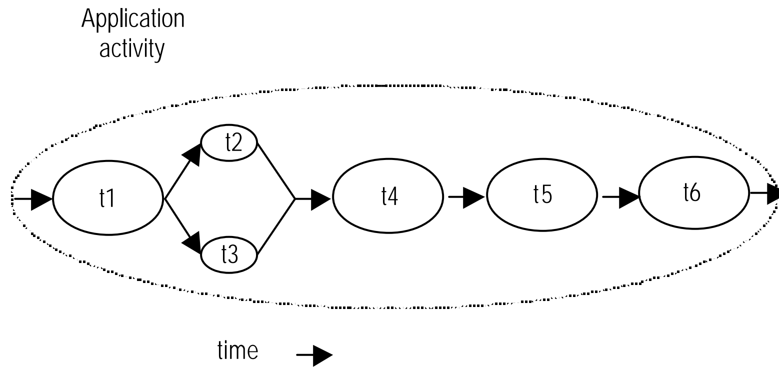

Figure 1: An example of a logical long-running "transaction", without failure.

However, if failures and concurrent access occur during the lifetime of these individual transactional activities then the behaviour of the entire "logical long-running transaction" may not possess ACID properties. Therefore, some form of (application specific) compensation may be required to attempt to return the state of the system to (application specific) consistency. Just as the application programmer has to implement the transactional work in the non-failure case, so too will programmers typically have to implement compensation transactions, since only they have the necessary application specific knowledge. Note, for simple or well-ordered work it is possible to provide automatic compensations.

For example, let us assume that *t4* has failed (rolls back). Further assume that the application can continue to make forward progress, but in order to do so must now undo some state changes made prior to the start of *t4* (by *t1*, *t2* or *t3*). Therefore, new activities are started; *tc1* which is a compensation activity that will attempt to undo state changes performed, by say *t2*, and *t3* which will continue the application once *tc1* has completed. *tc5'* and *tc6'* are new activities that continue after compensation, e.g., since it was not possible to reserve the theatre, restaurant and hotel, it is decided to book tickets at the cinema. Obviously other forms of transaction composition are possible.

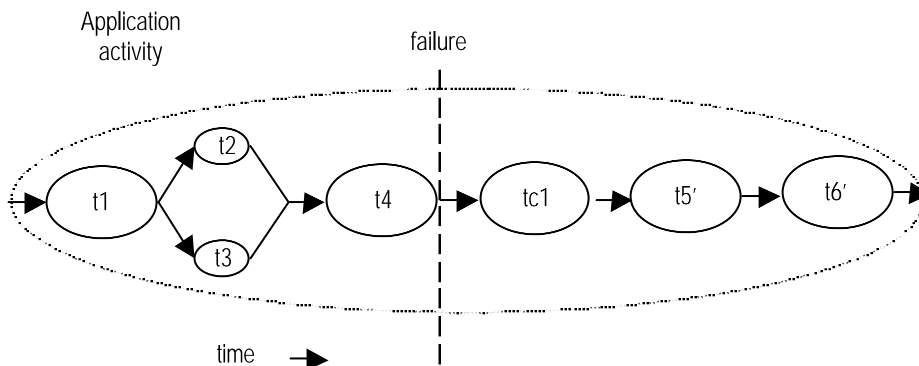

Figure 2: An example of a logical long-running "transaction", with failure.

(c) Copyright 2001 Hewlett-Packard Company

It should be noted that even with suitable compensations, it can never be guaranteed to make the entire activity transactional: in the time between the original transaction completing and its compensation running, other activities may have performed work based upon the results of the yet to be compensated transaction. Attempting to undo these additional transactions (if this is possible) can result in an avalanche of compensations that may still not be able to return the system to the state it had prior to the execution of the first transaction. In addition, compensations may (continually) fail and it will then be extremely important to inform users (or system administrators). Note, it will be application specific as to whether or not a compensation should be tried again if it does fail. For example, consider the situation where a transaction sells shares and the compensation is to buy them back; if the compensation fails it may be inappropriate (and expensive) to try it again until it does eventually succeed if the share price is going up rapidly.

1.1.2 Home entertainment system

Let us assume that we are interested in building our own customised home entertainment system consisting of TV, DVD player, hi-fi and video recorder. Furthermore, rather than purchase each of these from the same manufacturer we want to shop around and get the best of each from possibly different sources.

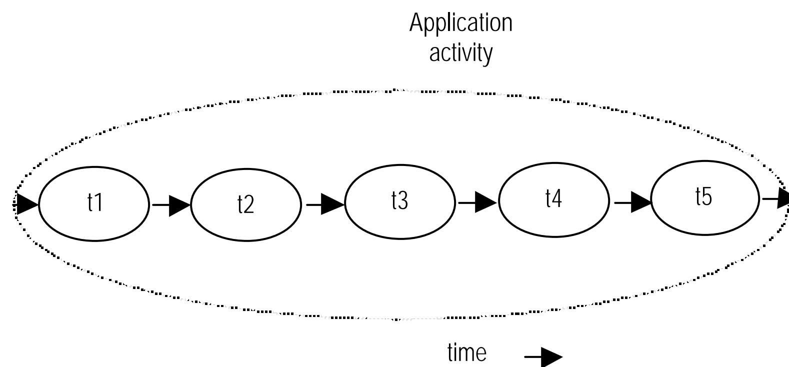

Figure 3: Building a home entertainment system via the Web.

When we visit the TV site we wish to start a transactional activity, t1, that will allow us to search and provisionally reserve (obtain transactional locks on) a number of different televisions (set A) that match our requirements. Before t1 terminates, we select a subset of the televisions, B, we are interested in, and provide a reference to our online bank account which the television site may contact to check that we have sufficient funds. t1 then ends, any locks obtained that are not members of B are released as normally for a transaction, (allowing other users to acquire them immediately if necessary), and all other locks are obtained and passed to t2.

The sequence of operations for t2, t3, and t4 are identical, where only subsets of items (DVD player, hi-fi, video) we have transactionally locked are released when each activity terminates. By the time t5 is executed there is a list of items that are locked and under it's control, possibly also including the on-line bank account. Therefore, t5 is responsible for committing or rolling back the final purchase order. All locked resources are atomically handed off to the next atomic action, and failures do not require compensation: the atomicity property of t1, t2, t3, t4 and t5 ensures that either the purchase happens, or it does not (and all resources will be released).

1.1.3 Web services coordination

Web services (components, objects, …) will typically not open up two-phase commit protocols to be driven by external coordinators. As a result, coordinating multiple Web services within a single transaction can *never* give the same ACID guarantees as multiple two-phase commit resources: in a single-phase model if a failure occurs after having committed some resources it is not possible to undo those that have committed. There are two solutions to this:

1) Wrap these one-phase objects in a two-phase wrapper which ignores the "prepare" phase and register them with a traditional transaction manager.
2) Treat this as an extended transaction model that requires specific coordination and error treatment.

In terms of implementation, there is no difference between 1 and 2: the resources are still one phase and may fail in exactly the same manner. However, on a conceptual level there is a significant difference: when using a transaction manager, programmers expect an all-or-nothing effect, especially since applications typically do not know about one-phase/two-phase restrictions of resources and may mix them in the same transaction; raising heuristic exceptions is the only possible solution for the transaction manager that finds it cannot undo committed operations, and then the application (or typically the administrator) has to deal with the outcome [1].

Giving applications a specific extended transaction model that clearly defines how resources behave *and* does not allow one-phase and two-phase resources to be registered in the same atomic action, gives a better understanding to users: the programmer must make a conscious choice as to the model that is being used, and the entire application is structured accordingly. In the end these types of applications are not transactional, and a traditional two-phase aware transaction system is inappropriate for them.

This notion of having a different extended transaction model for each use-case makes applications aware of the issues involved and helps to categorize objects and activities into which type of model they support. So, for example, one object may be used successfully within a two-phase and one-phase model, whereas another may only be used

within two-phase. It is important to reduce complexity in developing "transactional" internet application by not over overloading a given model (e.g., ACID transactions).

1.1.4 Timed transactions

Many business transactions have specific "real-time" deadlines within which they must operate (e.g., purchasing of shares). After the deadline has elapsed, if the business transaction has not completed in a normal manner, there will typically be application specific ways in which it must terminate (e.g., purchase the shares at the current price if it is less than a certain value). However, while this business transaction is running, its constituents may still require some or all of the ACID properties associated with transactions.

1.1.5 Arranging a meeting

The requirement is to arrange a date for a meeting between a group of people; it is assumed that each user has a *personal diary object* which records the dates of meetings etc., and each diary entry (slot) can be locked separately. The application starts by informing people of a forthcoming meeting and then receiving from each a set of preferred dates. Once this information has been gathered, it will be analyzed to find the set of acceptable dates for the meeting. This set is then broadcast to the users to get a more definitive idea of the preferred date(s). This process is repeated until a single date is determined. To reduce the amount of work which must be re-performed in the event of failures, and to increase the concurrency within the application, it would be desirable to execute each "round" of this protocol as a separate top-level transaction. However, to prevent concurrent arrangement activities from conflicting with each other, it would be beneficial to allow locks acquired on preferred diary entries to be passed from one transaction to another, i.e., the locks remain acquired on only those entries which are required for the next "round".

1.2 Conclusions

These applications share a common feature that as viewed by external users, in the event of successful execution (i.e., no machine failures or application-level exceptional responses which force transactions to rollback), the work performed possesses all ACID features of traditional transactional applications. If failures occur, however, non-ACID behaviour is possible, typically resulting in non-serializability. For some applications this does not result in application-level inconsistency, and no form of compensation for the failure is required. However, for other applications compensation may be required to restore the system to a consistent state from which it can then continue to operate.

As mentioned previously, compensation activities will typically be written for specific applications (and specific operations). However, even if compensation activities are required, it may be more efficient to exploit application-level semantics and allow the

application programmer to drive compensation, rather than rely upon system-driven compensation. Additionally, it me be impossible to guarantee that compensations will succeed, especially in the presence of failures.

There are several ways in which some or all of the requirements outlined above could be met [6][7][8][9]. We shall briefly outline two approaches:

- *independent top-level transactions*: with this mechanism it is possible to invoke a top-level transaction from within another transaction [7]. If the invoking transaction rolls back, this does not lead to the automatic rollback of the invoked transactions, which can commit or rollback independently of its invoker, and hence release resources it acquires. Such transactions could be invoked either synchronously or asynchronously. In the event that the invoking transaction rolls back, and compensation is required, compensating transactions may be invoked automatically by the transaction system or by the application.
- *structured top-level transactions*: long-running top-level transactions can be structured as many independent, short-duration top-level transactions, to form a "logical" long-running transaction [6]. This structuring allows an activity to acquire and use resources for only the required duration of this long-running transactional activity. In the event of failures, to obtain transactional semantics for the entire long-running transaction may require compensation transactions which can perform forward or backward recovery. Structuring applications which have long-running transactions out of many smaller transactions may require additional techniques (e.g., scripting languages) to ease the burden on application programmers.

Several industrial transaction systems already exist which support at least one of the above mentioned techniques. Experiences from these systems suggest that users benefit from the capability to relax strict ACID properties in a *structured, and well defined manner*. The ability to reason about applications structured from these techniques in both failure-free and failure-prone situations is extremely important. Traditional transaction semantics are a well understood concept, which users find useful. What these various techniques show is that one-size does not fit all, and different applications will require different qualities of "transactional" service. However, it is our belief that these different models share a core model based upon *activity signalling and distribution.*

To differentiate between ACID transactions and extended (non-ACID) transaction models we shall use the following terminology:

- *Atomic transaction (transaction)*: a traditional ACID transaction.
- *Atomic action*: an activity, or group of activities, that does not necessarily possess the guaranteed ACID properties. An extended action still has the "all or nothing" effect, i.e., failure does not result in partial work.

- *Business transaction*: an activity, or group of activities, that is responsible for performing some application specific work. A business transaction may be an atomic transaction or an atomic action.

1.3 Protocol configuration and negotiation

Just as we believe that one extended transaction model does not suit all application domains, it is possible that Web Service components may support multiple different extended transaction models (possibly representing different qualities of service). Either when the Web application is created, or when one component initially interacts with another, some level of protocol negotiation will be necessary to determine which transaction model will be used. If the component does not support the required extended transaction model then it will be up to the application to determine whether or not it makes sense to continue to use the component. For example, it may make sense for a transactional application to refuse to work with any service that does not support transactional semantics, i.e., does not accept (and use) transaction contexts that may be sent to it.

In addition, we do not assume that a single remote invocation mechanism (e.g., CORBA IIOP) will be the natural communication medium for all Web Services. How participants within and between activities appear to each other is not central to this discussion. They may be CORBA objects, communicating via IIOP, or they may be coarser grained Web Services objects, communicating via SOAP, for example. We assume that they will use the most appropriate invocation protocol for the application, e.g., it is unlikely that there will be much real-time video streaming over SOAP/HTTP. This does not preclude a given application from using multiple object models and communication protocols simultaneously. Note, in an internet environment where network and machine failures are relatively common (compared to, say, a LAN), the use of a slow invocation mechanism is more likely to result in transaction failures, particularly during the two-phase commit protocol. Since SOAP over HTTP is much slower than IIOP, it is likely that critical transactional applications (obviously a subjective term) will not use SOAP for transactional invocations (or at least for completing transactions).

We assume that protocol negotiation will occur on many levels (e.g., which transaction models are supported by a Web Service and in use by the invoker, which communications protocols are provided, business level issues, etc.) We do not prescribe when such negotiation occurs, but rather assume that any (or all) of the following mechanisms will be supported:

- *Statically*: prior to using a Web Service, an invoker may contact a naming/location service (e.g., UDDI), and from this obtain information about protocols etc. that the service provides, e.g., typically in the form of an XML document.

- *Dynamically*: when the Web Service is initially contacted there may be some initial set up (handshake) protocol during which protocol information in the form of XML documents will be exchanged. This may be necessary if a reference to a Web Service is returned from another service.
- *Caching*: once protocol information has been obtained about a Web Service, it may be cached and used again. Obviously cache information may become invalid, and cache invalidation protocols, or dynamic set ups may be used to solve this problem.

How protocol information is subsequently used to determine which transaction model, or communication protocol to use, for example, is beyond the scope of this paper, and will typically depend upon the application. For example, business considerations, or quality of service considerations may play a large part in this: SOAP may be the default communication protocol, however, if company A is prepared to pay for extra bandwidth then a different protocol may be used.

2. The proposed framework

Rather than construct each different extended transaction implementation from scratch, another approach would be to construct a generic *framework* on which many extended transaction models may be built [9][10][11]. This is the approach taken in [9] by the Object Management Group, in which it is shown that extended transaction models share a common requirement of coordination and control. This has several advantages: (i) the flexibility of such a framework allows for the reuse of components from one model in another, (ii) the framework adopted by the OMG to standardise extended transaction models is now also a JSR from Sun for extensions to J2EE [9][10], and will eventually be available in many servers and enterprises; leveraging this technology could help advance the development and use of enterprise level Web Services. Note, members of the Hewlett-Packard Arjuna Labs played a key role in specifying this OMG specification along with IBM and Iona Technologies.

In the rest of this paper we shall briefly describe the framework specified in [9][10] which we believe is sufficient to allow middleware to manage complex *business transactions* that extend the concept of *transaction* from the well-understood, short-duration atomic transaction. The different extended transaction models can be mapped onto specific implementations that use this framework such that they may interoperate and allow such transactions to span a network of systems connected indirectly by some distribution infrastructure. Owing to space limitations we can only give an overview of this work, and the interested reader is referred to [9] for a more detailed description. Note, although the framework in [9] is CORBA specific, the ideas it describes are not. Furthermore, CORBA implementations of it may be better suited to intranet

(c) Copyright 2001 Hewlett-Packard Company

environments, with gateways or bridges to equivalent Web Services implementations in an internet setting.

There are several ways in which some or all of the application requirements outlined in Section 1 could be met. However, it is unrealistic to believe that the "one-size fits all" paradigm will suffice, i.e., a single high-level model approach to extended transactions is unlikely to be sufficient for all (or even the majority of) applications. The area of business transactions is relatively new, and is still evolving, necessitating the use of a flexible, extensible protocol. Therefore, we shall describe a low-level infrastructure to support the coordination and control of abstract, application specific entities. As we shall show, these entities (*activities*) may be transactional, they may use weaker forms of serializability, or they may not be transactional at all; the important point is that we are only concerned with their control and co-ordination, leaving the semantics of such activities to the application programmer.

2.1 Activities

An *activity* is a unit of (distributed) work that may, or may not be transactional, and during its lifetime an activity may have periods of transactionality and periods of non-transactionality. An activity is *created*, made to *run*, and then *completed*. An *outcome* is the result of a completed activity, and this can be used to determine subsequent flow of control to other activities. Activities may run over long periods of time (minutes, hours, days, …) and can thus be *suspended* and then *resumed* later.

A very high level view of the role of the Activity Service is shown in Figure 4.

(c) Copyright 2001 Hewlett-Packard Company

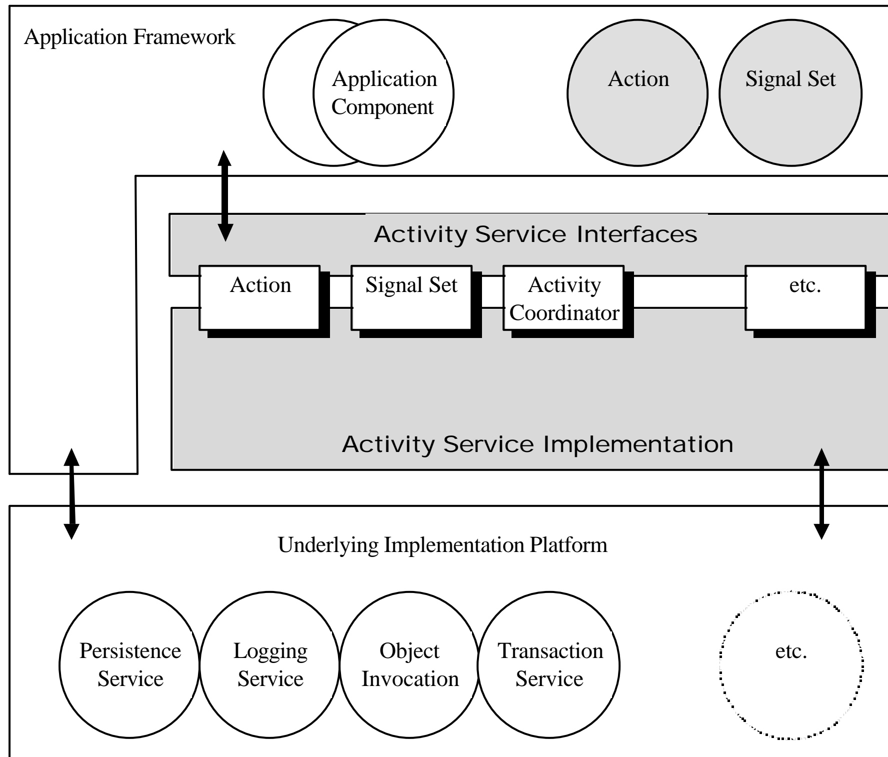

Figure 4: The role of the Activity Service.

Note, it is not expected that the operations in the Activity Services will be used directly by end-user programmers. In this context, the term *application programmers* refers to those who write for example, workflow managers or component management systems or who are extending the functionality of the Containers of Enterprise Java Beans (EJBs). Extended transactions like Sagas and Compensations have a complex structure and are intended to last over quite long intervals. Therefore a significant amount of middleware is required to manage the progress and recovery of an extended transaction. This should not be in the domain of an application programmer employed to write business software rather than middleware.

We shall define the *Application Framework* to be that middleware supplied by specialist vendors required to manage the progress of workflows and long-running business transactions in a variety of business domains, and further define *Application Component* to describe the components that "plug in" to such a framework. The Workflow Management Coalition (WfMC) and the OMG's Workflow Management Facility use the term Activity to describe a step in a path through a workflow digraph. An Application Component, if it is to be reusable, has to maintain a degree of independence from the Application Framework in which it runs. Thus a workflow manager might associate one or more Activities with a single Application Component each time giving it different properties that will serve to parameterize the enactment of the workflow [8]. For

example, an Application Component that holds a conversation with a graphical user interface (GUI) may be associated with two Activities, one of which has a property "the end-user language is French", the other which has "the end-user language is English", or again, the Application Component could be composed into a larger Activity that must run as a business transactions whose effects can be undone in some situations, whereas it could also be composed into another, or used stand-alone where no transactional behaviour is involved at all.

2.2 Activity coordination and control

An activity may run for an arbitrary length of time, and may use transactions at any number of points during its lifetime. For example, consider Figure 5, which shows a series of connected activities co-operating during the lifetime of an application. The solid ellipses represent transaction boundaries, whereas the dotted ellipses are activity boundaries. Activity *A1* uses two top-level transactions during its execution, whereas *A2* uses none. Additionally, transactional activity *A3* has another transactional activity, *A3'* nested within it. The framework is responsible for distributing both the activity and transaction contexts between execution environments in order that the hierarchy can be fully distributed.

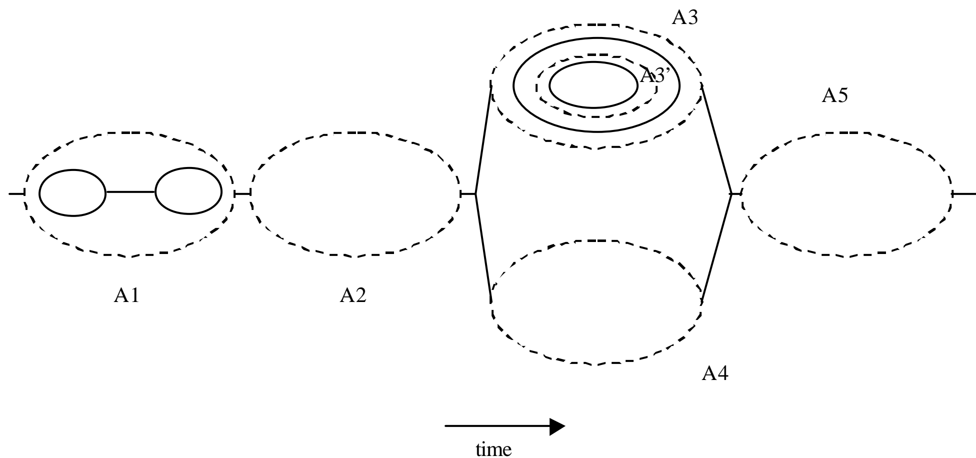

Figure 5: Activity and transaction relationship.

Just as a thread of control may require transactional and non-transactional periods and can suspend and resume its transactionality, so too may it require periods of non-activity related work. Thus, it is possible for a thread to perform some work outside the scope of the activity before returning to activity related work. In the example diagram above, if the thread performing activity *A3'* decided to perform some non-activity related work, it could do so outside the scope of *A3'* and *A3*.

(c) Copyright 2001 Hewlett-Packard Company

2.2.1 Actions and Signals

Demarcation signals of any kind are communicated to registered entities (*actions*[1]) through *signals*. For example, the termination of one activity may initiate the start/restart of other activities in a workflow-like environment. Signals can be used to infer a flow of control during the execution of an application. The information encoded within a Signal will depend upon the implementation of the extended transaction model and is allowed to be any arbitrary information.

An Action will use the Signal in a manner specific to the extended transaction model and return a result of it having done so. For example, upon receipt of a specific Signal, an Action could start another activity running (e.g., a compensation activity); another Action could commit any modifications to a database when it receives one type of Signal, or undo them if it receives another type.

2.2.2 Signal factories

To drive the Signal and Action interactions an *activity coordinator* is associated with each activity. Activities that require to be informed when another activity sends a specific Signal can register an Action with that activity's coordinator. When the activity sends a Signal (e.g., at termination time), the coordinator's role is to forward this signal to all registered Actions and to deal with the outcomes generated by the Actions.

The implementation of the coordinator will obviously depend upon the type of extended transaction model being used. For example, if a Sagas type model is in use then a compensation Signal may be required to be sent to Actions if a failure has happened, whereas a coordinator for a strict transactional model may require to send a Signal informing participants to rollback. Therefore, to enable the coordinator to be configurable for different transaction models, the coordinator delegates all Signal generation and control to the *SignalSet*.

The SignalSet is one of the keys to the extensibility of this framework: it's implementation is peculiar to the kind of extended transaction being used. The SignalSet is essentially a *Signal factory*, that generates signals that are sent to participants and processes the results returned to determine which signal to send next. Signals are associated with SignalSets and it is the SignalSet that generates the Signals the coordinator then passes to Actions. The set of Signals a given SignalSet can generate may change from one use to another, for example based upon the current status of the Activity or the responses from Actions. The intelligence about which Signal to send to an Action is hidden within a SignalSet and may be as complex or as simple as is required.

[1] Not to be confused with Atomic Action.

Importantly, a SignalSet is dynamically associated with an activity, and each activity can have a different SignalSet controlling it.

With the exception of some predefined Signals and SignalSets, the majority of Signals and SignalSets will be defined and provided by the higher-level applications that make use of this Activity Service framework. To use the generic framework provided it is necessary for these higher-level applications to impose application specific meanings upon Signals and SignalSets, i.e., to impose a structure on their abstract form.

For example, suppose we have two SignalSets to represent the possible outcomes for a transaction, Rollback and Commit, and register Actions with the Activity as the transactional resources; as with the OTS, it is up to the users of the Activity Service to ensure that appropriate Actions are registered at appropriate times.

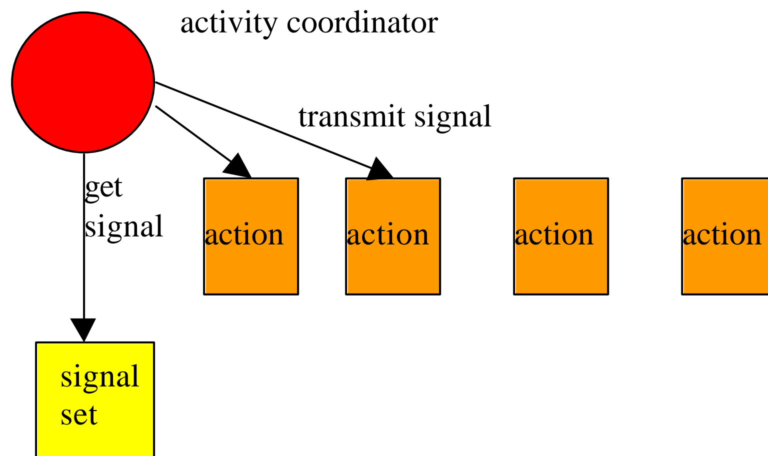

Figure 6, Activity coordinator signaling actions.

The Signal associated with the Rollback SignalSet would simply be “rollback”, whereas the Commit SignalSet would have “prepare”, “commit”, and “rollback” Signals. If the application decides to commit, then when called by the Activity Coordinator the SignalSet would generate the “prepare” Signal to be sent to the registered Actions, as shown in Figure 6. The Activity Coordinator would then send this Signal to each Action, and inform the SignalSet of the result. Assuming none of the Actions returns an exceptional response to this Signal, then when all Actions have received the “prepare” Signal, and the Activity Coordinator asks the SignalSet for the next Signal, it will return the “commit” Signal. However, if during the “prepare” phase, an Action returns a response which indicates that there is no point in sending the “prepare” Signal to further Actions, the Activity Coordinator will be required to obtain a new Signal from the SignalSet (the “rollback” Signal in this case), and send this to all registered Actions. As stated previously, the intelligence about which Signal to send, and about interpreting

outcomes from Actions, resides within the SignalSet, allowing implementations of the framework to be highly configurable, to match application requirements.

As new types of extended transaction emerge, so will new SignalSet instances and associated Actions. This allows a single implementation of this framework to serve a large variety of extended transaction models, each with its own idea of extended transactions, each with its own action and Signal Set implementations. The *Activity Service implementation* does not need to know the behaviour which is encapsulated in the Actions and SignalSets it is given, merely interacting with their opaque interfaces in an entirely uniform and transparent way.

2.3 Composite activities

An activity, which contains component activities, may impose a requirement on the Activity Service implementation for managing these component activities. It must be determined whether these component activities worked as specified or failed and how to map their completion (or non-completion) to the enclosing activity's outcome. This is true whether the activities are strictly parallel, strictly sequential or a complex structure. In general, an activity that needs to co-ordinate the outcomes of component activities has to know what state each component activity is in:

- which are active
- which have completed and what their outcomes were
- which activities failed to complete

Another activity may be required to handle the sub-activity outcomes so that control flows can be made explicit. The activity determines the collective outcome of the component activity in the light of the various component failure and success situations. The framework does not specify how the activities should be coordinated, only providing interfaces for coordination to occur. The coordination may therefore be performed in a manner most suitable to the application or extended transaction model. For example, a suitable scripting language may be required to assist the application programmer to define the roles of outcome manager and activities [6].

2.4 Activity failures

The failure of an individual activity may produce application specific inconsistencies depending upon the type of activity.

- if the activity was involved within a transaction, then any state changes it may have been making when the failure occurred will eventually be recovered automatically by the transaction service.
- if the activity was not involved within a transaction, then application specific compensation may be required.

- an application that consisted of the (possibly parallel) execution of many activities (transactional or not) may still require some form of compensation to "recover" committed state changes made by prior activities. For example, the application shown in Figure 2.

Rather than distinguish between compensating and non-compensating activities, compensation is considered to be the role of another activity: a compensating activity is simply performing further work on behalf of the application. Just as application programmers are expected to write "normal" activities, they will therefore also be required to write "compensating" activities, if such are needed. In general, it is only application programmers who possess sufficient information about the role of data within the application and how it has been manipulated over time to be able to compensate for the failure of activities.

2.5 State management and recovery

It is inherently complex to recover applications after failures (e.g., machine crashes); for example, the states of objects in use prior to the failure may be corrupt. The advantage of using transactions to control operations on persistent objects is that transaction systems ensure the consistency of the objects, regardless of whether or not failures occur. A transaction system guarantees that regardless of (non-catastrophic) failures, all transactions that were in flight when the failure occurred will either be committed or rolled back, making permanent or undoing any changes to objects.

Rather than mandate a particular means by which objects should make themselves persistent, many transaction systems simply state the requirements they place on such objects if they are to be made recoverable, and leave it up to the object implementers to determine the best strategy for their object's persistence. The transaction system itself will have to make sufficient information persistent such that, in the event of a failure and subsequent recovery, it can tell these objects whether to commit any state changes or roll them back. However, it is typically not responsible for the application object's persistence.

In a similar way, the Activity Service specification does not mandate a specific persistence and recovery mechanism. Rather it states what the requirements are on such a service in the event of a failure, and leaves it to individual implementers to determine their own recovery mechanisms. In a distributed application, where an individual activity may run on different implementations of the Activity Service during its lifetime, recovery is the responsibility of these different implementations. Each implementation may perform recovery in a completely different manner, forming *recovery domains*.

Unlike in a traditional transactional system, where crash recovery mechanisms are only responsible for guaranteeing consistency of object data, applications that use extended transaction implementations will typically also require the ability to recover the activity

structure that was present at the time of the failure, enabling the application to progress onwards.

Some of the recovery requirements are outlined below:

- *application logic*: the logic required to drive the activities during normal runtime is required during recovery in order to drive any in-flight activities to application specific consistency. Since it is the application level that imposes meaning on Actions, Signals, and SignalSets, it is predominately the application that is responsible for driving recovery.
- *application object consistency*: the states of all application objects must be returned to some form of application specific consistency after a failure.

If Activities and transactions co-operate within a given application, then the respective recovery mechanisms will also be required to co-operate. Obviously it is not necessary for a user of the Activity Service implementation to use transactions at all, in which case only Activity recovery will be required in the event of a failure, i.e., it is possible to have recovery domains that do not require a transaction service implementation at all.

3. Conclusions and further issues

From the previous discussions it should be evident that there are a range of applications that require different levels of transactionality. Many types of business transaction do not have the simple commit or rollback semantics of an ACID transaction, and may complete in a number of different ways that may still be interpreted as successful but which do not imply everything that the business transaction did has occurred.

We have shown that a flexible and extensible framework for extended transactions is necessary, then in addition to standardising on the interfaces to this framework, we also need to work on *specific extended transaction models that suit the Web*. We would not expect applications to work at the level of Signals, Actions and SignalSets, as these are too low-level. Higher-level APIs are required to isolate programmers from these details. However, from experience we have found that this framework helps to clarify the requirements on specific extended transaction implementations.

We have given examples of the types of Web applications that have different requirements on any transaction infrastructure, and from these we believe it should be possible to obtain suitable extended transaction models. Other issues that will need to be considered when implementing many business transactions include:

1) Security and confidentiality: any business transaction involving buying or selling items, whether they be hotel rooms or newspapers, requires guarantees that the buyer/seller is who they appear to be, and that no one can “snoop” the connection and obtain information they are not entitled to.

(c) Copyright 2001 Hewlett-Packard Company

2) Audit trail: maintaining a log of the actions performed during a business transaction can be useful for a number of reasons, not least that of non-repudiation in the case of legal action.
3) Protocol completeness guarantee: even in the presence of failures, the correctness guarantee for the application relies upon the structure of the atomic action (application activity) being followed. The information about which activity to invoke when and under what circumstances must reside in, for example, a highly available repository, such that failure of the original "controller" (that entity which was responsible for parsing and driving the activities) does not cause the activity to stop, or for branches of it to be ignored.
4) Quality of service: some Web Services may support different types of extended transaction model as well as different communication protocols. The selection of which model to use may depend upon quality of service requirements.

How these fit into an extended transaction model will be one of the areas of future research and development.

4. References

[1] J. N. Gray, "The transaction concept: virtues and limitations", Proceedings of the 7th VLDB Conference, September 1981, pp. 144-154.

[2] D. J. Taylor, "How big can an atomic action be?", Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986, pp. 121-124.

[3] OMG, *CORBAservices: Common Object Services Specification*, Updated July 1997, OMG document formal/97-07-04.

[4] J. E. B. Moss, "Nested Transactions: an approach to reliable distributed computing", Ph.D. Thesis 260, MIT, Cambridge, MA, April 1981.

[5] P. M. Shwarz and A. Z. Spector, "Synchronizing Shared Abstract Types", ACM Transactions on Computer Systems, Vol. 2, No. 3, August 1984, pp. 223-250.

[6] Nortel, supported by the University of Newcastle upon Tyne, "OMG document bom/98-03-01", submission for the OMG Business Object Domain Task Force (BODTF): Workflow Management Facility, 1998.

[7] B. Liskov and R. Scheifler, "Guardians and actions: linguistic support for robust distributed programs", ACM TOPLAS, Vol. 5, No. 3, July 1983, pp. 381-404.

[8] C. Pu, G. Kaiser and N. Hutchinson, "Split-transactions for open-ended activities", Proceedings of the 14th VLDB Conference, Los Angeles, September 1988, pp. 26-37.

[9] H. Garcia-Molina and K. Salem, "Sagas", Proceedings of the ACM SIGMOD International Conference on the Management of Data, 1987.

(c) Copyright 2001 Hewlett-Packard Company

[6] OMG, *Object Constraint Language Specification*, Revision 1.1, September 1997, OMG document ad/97-08-08.

[7] C. T. Davies, "Data processing spheres of control", IBM Systems Journal, Vol. 17, No. 2, 1978, pp. 179-198.

[8] J. J. Halliday, S. K. Shrivastava, and S. M. Wheater, "Implementing Support for Work Activity Coordination within a Distributed Workflow System", Proceedings of the Third International Conference on Enterprise Distributed Object Computing (EDOC '99), September 1999, pp. 116-123.

[9] OMG, *Additional Structuring Mechanisms for the OTS Specification*, September 2000, document orbos/2000-04-02.

[10] JSR-000095, "J2EE Activity Service for Extended Transactions", http://java.sun.com/aboutJava/communityprocess/jsr/jsr_095_activity.html

[11] R. Barga and C. Pu, "A Practical and Modular Method to Implement Extended Transaction Models", Proceedings of the 21st VLDB Conference Zurich, Switzerland, 1995.