A comparison of Web services transaction protocols

A comparative analysis of WS-C/WS-Tx and OASIS BTP

Level: Introductory

Mark Little, Chief architect, Arjuna Technologies Ltd
Thomas Freund, Senior Technical Staff Member , IBM

07 Oct 2003

Level: Introductory

Mark Little, Chief architect, Arjuna Technologies Ltd
Thomas Freund, Senior Technical Staff Member , IBM

07 Oct 2003

Up to August 2003 there were two contenders for the Web services transaction space: OASIS Business Transactions Protocol (BTP), and the Web Services Transactions (WS-Tx) specification. There have been several subjective articles and comments comparing BTP to WS-Tx, attempting to show that BTP can do everything WS-Tx can and ignoring the important differences that exist. This article will try to give an objective comparison of these two specifications and show how they both attempt to address the problems of running transactions with Web services. At the end of the article it should be apparent how and why WS-Tx and BTP are different, while at the same time illustrating where they do have some commonality.

Most workflow and business-to-business collaborative applications require transactional support in order to reach a mutually-agreed outcome. Transactional support ensures this outcome is observed consistently across all of the tasks within the application that comprises the business activity. The results of a task are typically made available before the overall business application or activity completes. For example, an airline reservation system may reserve a seat on a flight for an individual for a specific period of time, but if the individual does not confirm the seat within that period, it will be reclaimed for another passenger.

Thus it is difficult, if not impossible, to incorporate traditional transaction architectures within such environments. Furthermore, most collaborative business process management systems support complex, long-running processes where undoing tasks which have already completed may be necessary in order to effect recovery, or to choose another acceptable execution path in the process.

Web services are specifically about fostering systems interoperability. This presents some interesting issues from a transaction management point of view, particularly the fact that the Web services architecture is deliberately not prescriptive about what happens behind service implementations: Web services are only concerned with the transfer of structured data between parties, plus any meta-level information to safeguard such transfers (for example, by encrypting or digitally signing messages) – yet it is behind a service implementation that you find traditional transaction processing architectures supporting business activities.

However, Web services also offer the possibility of a straightforward solution to a very important transaction problem: interoperability. Ever since transaction processing began, there has been a variety of transaction protocol standards (such as X/Open and the OTS, see Resources) and vendor-specific protocols, with many corresponding implementations. Interoperability between these

various protocols has always proved problematic and there has been limited success. Web services offer a solution to this problem.

Thus there is a paradox: Web services provide a service-oriented, loosely-coupled, and potentially asynchronous means of propagating information between parties (see Resources for WS-Security and BTP references), while the underlying services use traditional transaction processing infrastructures. Furthermore, the fact that transactions in back-end systems are constructed with ACID properties can potentially lead to problems when composing business activities from these services/resources, since it presents opportunities to those parties to lock resources and prevent transactions from making progress. Thus if transactions are to be supported in the Web services architecture, then it is clear that some re-addressing of the problem is required.

In 2001, a consortium of companies including Hewlett-Packard, Oracle and BEA began work on the Organization for Advance Structured Information Systems (OASIS) Business Transaction Protocol (BTP), which was aimed at business-to-business transactions in loosely-coupled domains such as Web services. By April 2002 it had reached the point of a committee specification.

However, others in the industry, including IBM, Microsoft, and BEA released their own specifications: Web Services Coordination (WS-C) and Web Services Transactions (WS-T) (see Resources for appropriate references).

Although we'll examine this in more detail later, they key differences between these specifications can be roughly categorized as follows:

- BTP is not specifically about transactions for Web services – the intention was that it could be used in other environments. As such, BTP defines the transactional XML protocol and must specify all of the service dependencies within the specification. WS-C and WS-Tx are specifically for the Web services environment and hence build on the basic definition of a Web services infrastructure.
- The foundations of WS-Tx are based on traditional transaction infrastructures, where there is a strong separation between the functional aspects of business logic and the non-functional aspects of using transactions within an application. BTP essentially started from scratch and requires business-level decisions to be incorporated within the transaction infrastructure.

In this paper we'll give an objective analysis of these two transaction protocols and compare and contrast the approaches they have taken. Because there are a number of good texts available on OASIS BTP (see Resources for some examples) we will not spend as much time describing that protocol as we will for WS-C and WS-Tx where less information is currently available.

Specification version.

Note that at the time of writing we are comparing the BTP 1.0 committee specification with the preliminary version of the WS-C/Tx specifications. The latter specifications have solicited industry feedback and may undergo additional change if warranted.

Overview

Distributed systems pose reliability problems not frequently encountered in centralized systems. A distributed system consisting of a number of computers connected by a network can be subject to independent failure of any of its components, such as the computers themselves, network links, operating systems, or individual applications, and activities may take an indeterminate duration to execute. Decentralization allows parts of the system to fail while other parts remain functioning, which leads to the possibility of abnormal behavior of executing applications.

Consider the case of a distributed system where the individual computers provide a selection of useful services which can be utilized by an application. It is natural that an application that uses a collection of these services requires that they behave consistently, even in the presence of failures. A very simple consistency requirement is that of *failure atomicity*: the application either terminates normally, producing the intended results, or is aborted, producing no results at all. In the case of an abort, how the state of the system is restored to some predefined state is typically an

implementation choice. This failure atomicity property is supported by traditional transaction processing systems through atomic transactions.

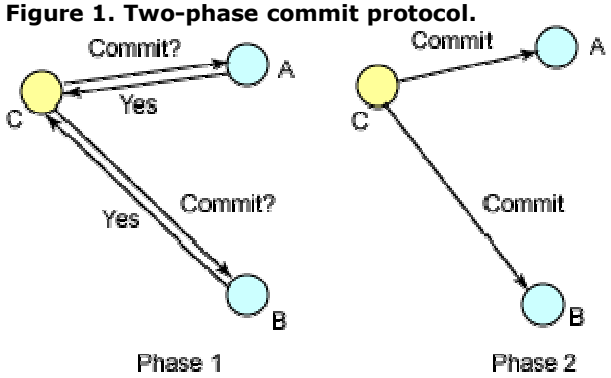
A transaction can be terminated in two ways: *committed* or *aborted* (cancelled). When a transaction is committed, all changes made within it are made durable (forced on to stable storage such as disk). When a transaction is aborted, all changes made during the lifetime of the transaction are undone. Interoperability of existing transaction processing systems is an important part of Web services transactions -- such systems already form the backbone of enterprise level applications and will continue to do so for the Web services equivalent. Business-to-business activities will typically involve back-end transaction processing systems either directly or indirectly, and being able to tie together these environments will be the key to the successful take-up of Web services transactions.

Traditional transaction systems are typically referred to as ACID transactions. An ACID transaction has the following properties:

- *Atomicity*: The transaction completes successfully (commits), or if it fails (aborts), all of its effects are undone.
- *Consistency*: Transactions produce consistent results and preserve application-specific invariants.
- *Isolation*: Intermediate states produced while a transaction is executing are not visible to other transactions. Furthermore transactions appear to execute serially, even if they are actually executed concurrently. This is typically achieved by locking resources for the duration of the transaction so that they cannot be acquired in a conflicting manner by another transaction.
- *Durability*: The effects of a committed transaction are never lost (except by a catastrophic failure).

Traditional transaction systems use a two-phase protocol to achieve atomicity between participants, as illustrated in Figure 1. During the first (*preparation*) phase, an individual participant must make durable any state changes that occurred during the scope of the transaction, such that these changes can either be rolled back or committed later once the transaction outcome has been determined. Assuming no failures occurred during the first phase, in the second (*commitment*) phase participants may "overwrite" the original state with the state made durable during the first phase.

Figure 1. Two-phase commit protocol.



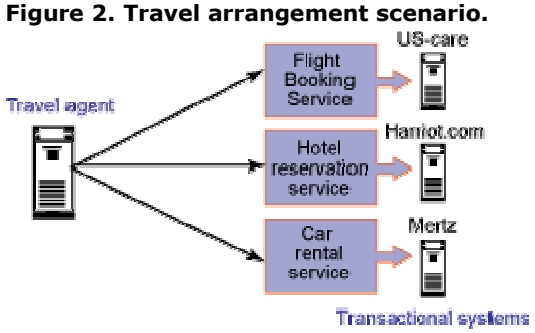
In order to guarantee consensus, two-phase commit is necessarily a blocking protocol: after returning the first phase response, each participant which returned a commit response *must* remain blocked until it has received the coordinator's phase 2 message. Until they receive this message, any resources used by the participant are unavailable for use by other transactions, since to do so may result in non-ACID behavior. If the coordinator fails before delivery of the second phase message, these resources remain blocked until it recovers.

Although most classical transaction systems are implementations of the ACID protocol, the various

properties of an ACID transaction can be relaxed to provide what are typically referred to as *extended transactions* (take a look at the OMG Additional Structuring Mechanisms in Resources); for example, an extended transaction model may relax atomicity to allow partial sets of participants to commit or abort, or it may relax isolation to allow concurrent users to observe partial results. The "classic" ACID protocol can be considered to be a well-formed, two-phase protocol in this spectrum of protocols. As you will see later, other protocols, such as BTP cohesions and WS-Tx Business Activities, fall into the spectrum with varying degrees associated with each of the functionalities.

Composing certain activities from long-running ACID 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 cancelling, however, certain activities may be necessary to restore the system to a consistent state (perhaps performing compensation or counter-effects). Such compensation and fault-handling 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. Thus an extended transaction model is more appropriate for long-duration interactions.

Figure 2. Travel arrangement scenario.



For example, take the relatively simple scenario of arranging travel and accommodation for a conference. In particular, the attendee will require a flight to the city where the conference is being held, a room reservation at a hotel, and possibly a rental car for the duration of the conference.

While locating flight, hotel and car rental options you need to ensure likely options can be reserved as you assemble the required set of reservations required for the trip as a whole. As well as considering the needs of the conference attendee, service providers also need to have some autonomy and maintain control of their own resources (flight, room, and car rental reservations).

The elements required for the booking are interrelated within this domain and yet they are not necessarily pre-determined. Obviously without a flight it makes no sense to book the hotel or to rent a car unless the conference were local, but in other circumstances it may make sense to book the flight and hotel, but if the hotel booking you make is at the same hotel as the conference, it may be possible to do without the car rental.

You may also want to keep your options open by reserving a number of flights while looking for other more direct travel options or other convenient hotels. The customer solicits multiple quotes to determine the lowest-cost supplier. Therefore, conducting the entire travel arrangements within a single classic (ACID) transaction is inappropriate, since in that situation either all of the work occurs or none occurs, which is inappropriate given the travel agent's requirements. With traditional ACID transactions, it would not be possible to have the partial outcomes (relaxed atomicity) that might be required if visiting multiple flight booking services, for example.

Back to top

Functionality of Web services transactions

The fundamental question addressed in this section is what properties must a transaction model possess in order to support business-to-business interactions? To begin to answer that, you might first need to understand what is meant by a *business transaction*.

A *business relationship* is any distributed state maintained by two or more parties, which is subject to some contractual constraints previously agreed to by those parties. A *business transaction* can therefore be considered as a consistent change in the state of a business relationship between parties. Each party in a business transaction holds its own application state corresponding to the business relationship with other parties in that transaction. During the course of a business transaction, this state may change.

In the Web services domain, information about business transactions is communicated in XML documents. However, how those documents are exchanged by the different parties involved (such as email or HTTP) may be a function of the environment, type of business relationship, or other business or logistical factors.

The following sections will consider the characteristics typical for extended transactions and then talk about the specific requirements for business transactions.

Characteristics of extended transactions

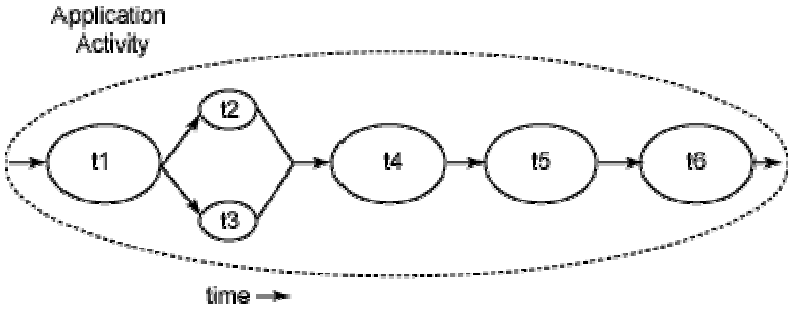
An *activity* is a unit of (distributed) work that may, or may not be transactional. During its lifetime an activity may have transactional and non-transactional periods. Every entity including other activities can be parts of an activity, although an activity need not be composed of other activities. An activity is *created*, made to *run*, and then *completed*. The result of a completed activity is its *outcome*, which can be used to determine subsequent flow of control to other activities.

A *task* is a short-duration unit of work that may be better suited to more traditional transactional semantics. Each task may execute on different, distributed systems or domains, and the internal composition of a task may involve many different machines/domains or sub-tasks. How tasks are implemented to perform the necessary work is typically unimportant to the application.

The structuring mechanisms available within traditional transaction systems are sequential and concurrent composition of transactions. These mechanisms are sufficient if an application function can be represented as a single top-level transaction. Frequently with Web services this is not the case. Top-level transactions are most suitably viewed as *short-lived* entities (tasks), performing stable state changes to the system; they are less well-suited for structuring *long-lived* application functions (such as running for minutes, hours, days, …). Activities implemented using traditional systems may reduce the concurrency in the system to an unacceptable level by holding on to locks for a long time; further, if such a transaction rolls back, much valuable work already performed could be undone.

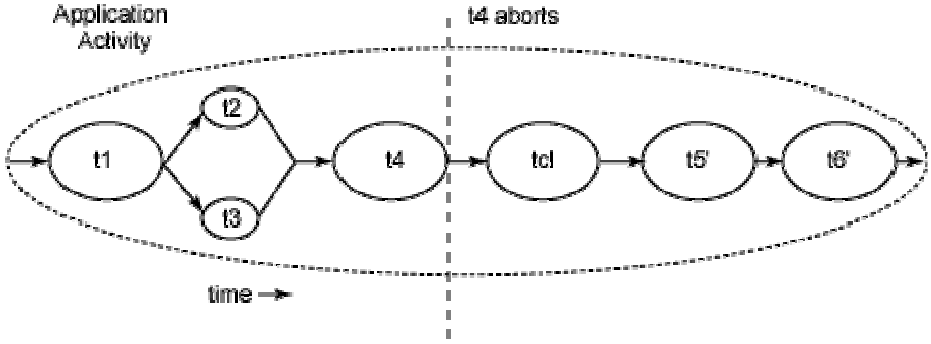
Activities can be structured as many independent tasks to form an overall application. This structuring allows an activity to acquire and use resources for only the required duration within this long-running transactional activity. This is illustrated in Figure 3, where an *activity* (shown by the dotted ellipse) has been split into many different, coordinated, tasks. Assume that 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), and so on. If all of these operations were performed as a single transaction 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.

Figure 3. An example of a logical long-running "transaction", without failure.



In addition, task failures do not necessarily affect the overall activity, unlike traditional ACID transactions. Such long-running applications are generally constructed such that some form of (application-specific) compensation may be required to attempt to return the state of the system to (application-specific) consistency. For example, assume that t4 aborts (Figure 4). Further assume that the activity 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 tasks are started; tc1 which is a compensation task 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 tasks that continue after compensation; for example, if it is not possible to reserve the theater, a ticket at the cinema might be an alternative event to go along with the previously booked restaurant and hotel. Obviously other forms of transaction composition are possible.

Figure 4. An example of a logical long-running "transaction", with failure.



There are several ways in which some or all of the application requirements outlined above could be met. However, it is unrealistic to believe that the "one-size fits all" paradigm will suffice, in other words, a single approach to extended transactions is unlikely to be sufficient for all (or even the majority of) applications. Whereas in case of the last example, a transactional workflow system with scripting facilities for expressing the composition of the activity with compensation (a workflow) may be the most suitable approach; a less elaborate solution might be desirable for the first three examples.

Requirements

As Web Services have evolved as a means to integrate processes and applications at an inter-enterprise level, traditional transaction semantics and protocols have proven to be inappropriate for the reasons mentioned previously. Web services-based transactions differ from traditional transactions in that they execute over long periods, they require commitments to the transaction to be negotiated, and isolation levels have to be relaxed.

Since business relationships imply a level of value to the parties associated by those relationships, achieving some level of consensus between these parties is important. Not all participants within a particular business transaction have to see the *same* outcome; a specific business transaction may

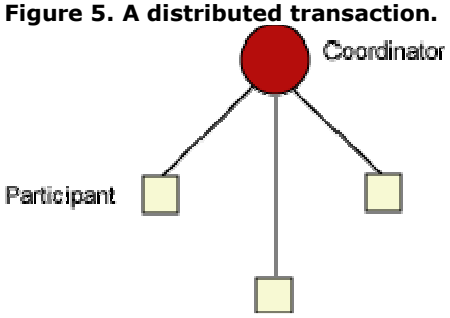
possess multiple different *consensus groups*, with participants in each group observing different outcomes. In addition some consensus groups may allow the atomicity within a specific transaction to be relaxed, allowing subsets of participants to receive different outcomes. This flexibility in the participant list of a group is an important difference between these kinds of extended transactions and traditional transaction systems.

Furthermore, it should be possible for a participant to exit a consensus group when required. There are a number of reasons why a participant may no longer wish to be involved in the consensus decision; for example, the work it has performed can complete safely irrespective of the final outcome of the consensus group, or it may be necessary for a separate task (a different service or domain) to perform a counter-effect in the event the consensus group cancels the work.

Consider the situation depicted in Figure 5, where there is a transaction coordinator and three participants. Assume that each of these participants is on a different machine to the coordinator and each other. Each of the lines connecting the coordinator to the participants also represents the invocations from the coordinator to the participants and vice versa:

- Enroll a participant in the transaction.
- Execute the coordinator termination protocol.

Figure 5. A distributed transaction.



As far as a coordinator is concerned, it does not matter what the participant implementation is -- although one participant may interact with a database to commit the transaction, another may just as readily be responsible for forwarding the coordinator's messages to a number of databases, essentially acting as a coordinator itself.

This technique of using proxy coordinators (or subordinate/sub-coordinators) is known as *interposition*. Each domain (machine) that imports a transaction context may create a subordinate coordinator that enrolls with the imported coordinator as though it were a participant. Interposition is important for a number of reasons, including performance optimization and security. Each subordinate coordinator may represent a separate domain that is responsible for its own security, protocol bridging, etc.

Consensus groups achieve consistent outcomes among participants, but are only part of the picture. Often in business-to-business relationships there are hierarchies of these groups (*scopes of work*), with parent and child relationships existing between them. Typically the work performed by a child is provisional on the successful completion of the parent; for example, the parent scope can perform a counter-effect for the completed child.

It is important to realize that parent-child (activity-task) scopes are *not* equivalent to interposition. In an interposed hierarchy, sub-nodes complete only when instructed to by the completion of their superior nodes. In a nested scope relationship (such as nested transactions), the sub-scopes can complete independently of their parents and may then impose compensation requirements on the parent.

In addition to understanding the outcomes, a participant within a business transaction may need to

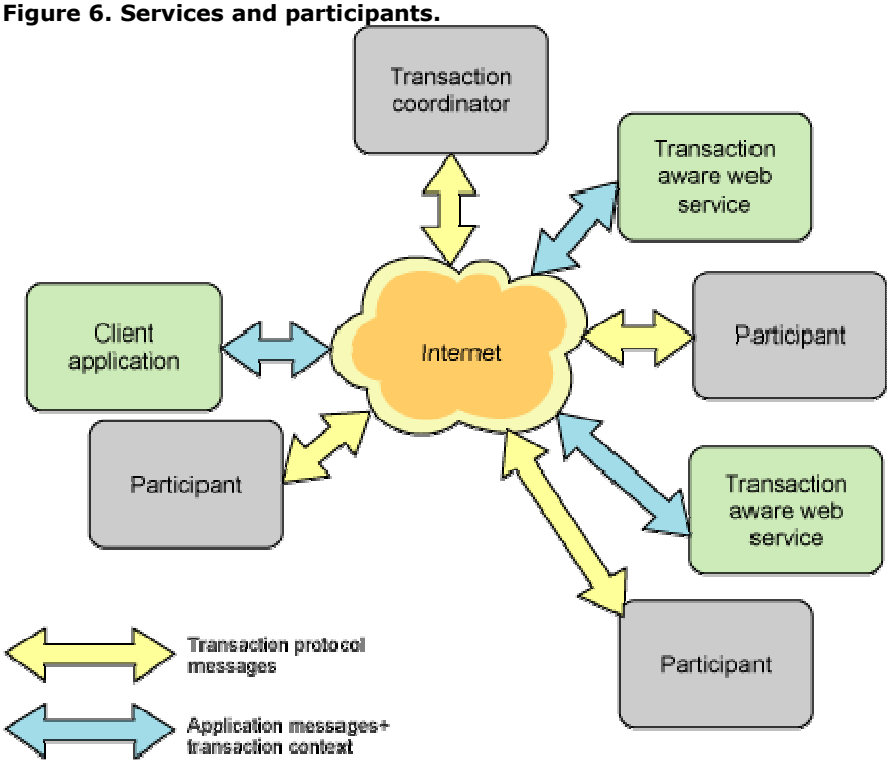
support provisional or tentative state changes during the course of the transaction. Such parties must also support the completion of a business transaction either through confirmation (final effect) or cancellation (counter-effect). In general, what it means to confirm or cancel work done within a business transaction will be for the participant to determine.

For example, an application may choose to perform changes as provisional effect and make them visible to other business transactions. It may store necessary information to undo these changes at the same time. On confirmation, it may simply discard this undo changes or on cancellation, it may apply these undo changes. An application can employ such a compensation-based approach or take a conventional *roll back* approach, for example.

Finally, it's also important for any Web services transactions protocol to have interoperability with existing transaction processing systems. Such systems already form the backbone of enterprise-level applications and will continue to do so for the Web services equivalent.

In the following sections we will discuss whether and how both BTP and WS-Tx have addressed these issues. However, before we do so, it is important to understand that both WS-Tx and BTP allow a distinction to be made between a transactional service and the participants that are controlled by the transaction, as illustrated in Figure 6.

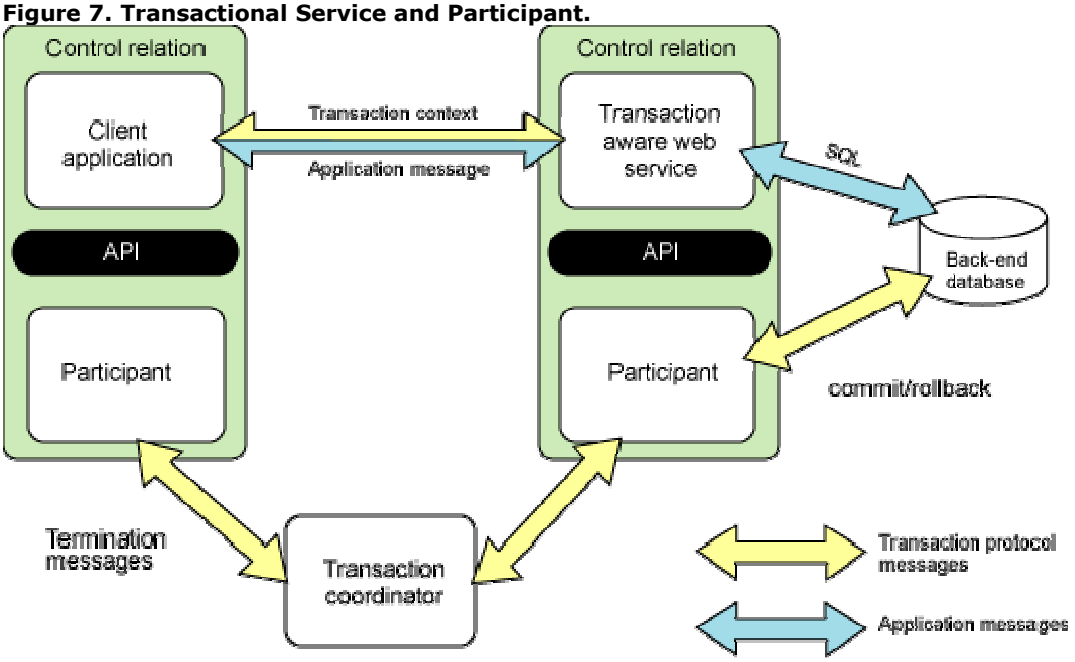
Figure 6. Services and participants.



- *Transactional service*: This service enables the application to conduct work within the scope of a business transaction. The outcome of this work is not finalized until the application instructs the transaction service to either commit or abort. An example of such an object would be a Web service that allows users to place items into a shopping basket, as shown in Figure 7; only if the user decides to confirm the purchase and the application then commits the transaction does the purchase of the items in the basket occur. The responsibility for orchestrating the outcome across the tasks/services that comprise the work is removed from the application and placed under the control to the transaction service.
- *Transactional participant*: This is the entity that controls the outcome of the work

performed by the transactional object. For example, if the online shopping service uses a database to store information on the items in the basket, it will typically access this information via a driver. SQL statements will be sent to the database for processing via the driver, but these statements will be tentative and only commit when (and if) the transaction does so. In order to do this, the driver/database will associate a participant with the transaction and this will inform the database of the outcome. Note that in the case of interposition, this participant may actually be a coordinator as you saw earlier.

Figure 7. Transactional Service and Participant.



Section summary

To summarize what we've discussed in the previous sections, the requirements placed on the use of transactions in Web services mean that any Web services transaction model should support the following functionality:

- Relaxation of ACID properties in a structured, well-defined manner; strict ACID properties, especially atomicity are not appropriate for all applications. Many long-duration activities required are not atomic all-or-nothing (see consensus groups). Likewise, often results of tasks are exposed before the overall activity has terminated (relaxation of isolation).
- Flexible outcomes for consensus groups. For example, open-flat, where the participants in a transaction are exposed to the business logic allowing it to define the relationships of the individual units of work to the transaction task; open-nested, where there are tasks within an activity forming a parent-child relationship of consensus groups.
- Flexible participation in consensus groups; a task can leave an activity (exit the group) prior to outcome processing if it decides it does not affect that processing, in other words, it does not expose results or cause side-effects.
- Activities and tasks should be defined as individual scopes (consensus groups), with clearly-defined relationships between them so that the service can also cleanly delineate responsibilities. Scopes allow the work performed by services or a long-running activity to be clearly demarcated by the application or the service. In addition, termination of scopes resides in the domain of the application and it driven from *top-down*. This is another important distinction between scopes and interposition, where a participant (mapped to a scope, for example) may exit an activity autonomously (in a *bottom-up* manner) and does not give the application the control required to properly manage scopes.

Back to top

The OASIS Business Transactions Protocol

BTP was the first cross-industry attempt to produce an XML standard for business-to-business transactions. It is designed to support applications which are disparate in time, location, and administration and thus require transactional support beyond classical ACID transactions. It is a protocol for orchestrating business processes between loosely-coupled software services to achieve consistent outcomes from the participating business parties.

In BTP, the notion of consensus groups mentioned earlier is obtained through the two transaction protocols that are defined, atoms and cohesions. Both of these transaction types mandate a *two-phase completion protocol* to ensure atomicity between (sub-sets of) participants (you'll see what we mean by this soon). During the first phase (*prepare*), an individual participant must make durable any state changes that occurred within the scope of the transaction, such that those changes can either be undone (*cancelled*) or made durable (*confirmed*) later once consensus has been achieved.

Although BTP uses a two-phase protocol, there is no implication of *ACID semantics* within the BTP. The completion protocol is only concerned with achieving consensus. How participant implementations of the prepare, confirm, and cancel phases are provided is a back-end implementation decision. Issues to do with consistency and isolation of data are also back-end choices and *not* imposed or assumed by BTP; in fact it is not possible to infer from a participant using BTP what back-end choices it has made; for example, there is no Policy Framework such as in the WebServices (see Resources), where behavior is described in policy assertion statements (allowing for interpretation and tooling).

This is good in so far that traditional ACID transactions are not suitable for all types of Web services interactions. However, everything is left up to back-end implementation choices and there is nothing in the protocol (implicit or explicit) to allow a user to determine what choices have been made. Therefore, it is impossible to reason about the ultimate correctness of a distributed application. For example, if you wanted to use BTP for ACID transactions, then of course services could use traditional XA resource managers, for example, wrapped by BTP participants. Unfortunately, there is no way within the BTP for those services to inform external users that this is what they have done so that they can safely be used within the scope of a BTP "ACID" transaction.

Because the traditional two-phase algorithm does not impose any restrictions on the time between executing the first and second phases, BTP took the approach of using this to allow business-logic decisions to be inserted between the phases. What this means is that users are required to drive the two phases explicitly in what BTP terms an *open-top completion protocol*. The application has complete control over when transactions prepare, and using whatever business logic is required, later determine which transactions to confirm or cancel. Prepare becomes an integral part of the service business logic.

This is a significant difference from traditional transaction systems, where an application is only allowed to tell a transaction to commit (confirm) or rollback (cancel); the transaction coordinator then executes the entire two-phase protocol before returning control (and the result) to the application. Being able to control both phases means that the participant and the service on whose behalf it acts, must co-operate closely. The act of being told to prepare by the coordinator is typically reflected by the participant into a business-level decision, such as reserving a quote for a flight. In a traditional transaction system, the reservation would have occurred prior to the commit protocol being executed, and informing the

Open flat

BTP's *open top-completion* is what the industry generals terms Open-flat. The term open-top implies that there is a hierarchy of scopes, but in fact BTP does not support nesting of scopes, only interposition, which as we saw earlier, is not the same.

participant to prepare essentially attempts to make that reservation durable (such as turning the reservation into a booking).

Transaction types

Before going into more detail on why open-top is important to BTP, let's first examine the transaction semantics that are supported within the protocol. BTP introduced two types of *extended transactions*, both using the open-top, two-phase completion protocol:

- *Atom*: an atom is the typical way in which "transactional" work performed on Web services is scoped. The outcome of an atom is guaranteed to be atomic, such that all enlisted *participants* (acting on behalf of their associated Web services) will see the same outcome, which will either be to accept (*confirm*) the work or reject (*cancel*) it. Although at first glance it may seem as though BTP atoms are equivalent to atomic transactions: they are not. We will revisit this in a later section, but it is worth giving some brief details here. BTP did not consider interoperability with existing transaction systems as an important factor. The semantics for an atom (isolation, durability etc.) are not as precisely-defined as those you can expect from an atomic transaction.
- *Cohesion*: this type of transaction was introduced in order to *relax atomicity* and allow for the selection of work to be confirmed or cancelled based on higher-level business rules. Atoms are the typical participants within a cohesion but, unlike an atom, a cohesion may give different outcomes to its participants such that some of them may confirm while the remainder cancel. In essence, the two-phase protocol for a cohesion is parameterized to allow a user to specify precisely which participants (either atoms or stand-alone participants) to prepare and which to cancel. The strategy underpinning cohesions is that they better model long-running business activities, where services enroll in atoms that represent specific units of work, and as the business activity progresses it may encounter conditions that allow it to cancel or prepare these units with the caveat that it may be many hours or days before the cohesion arrives at its ultimate decision and specifies its *confirm-set*: the set of participants that it requires to confirm in order for it to successfully terminate the business activity. Once the confirm-set has been determined, the cohesion collapses down to being an atom: all members of the confirm-set will see the same outcome. As we discussed earlier, this is precisely the kind of weakening of consensus groups required from Web services transactions.

At first glance it may appear that these two transaction models are distinct. However, cohesions in effect present a superset functionality of atoms: if you have a cohesion coordinator then you can use that same implementation to provide support for atoms (though the inverse is not the case).

It is also important to understand that as with a traditional two-phase protocol, there is no ordering implied by the registration of participants in a transaction (atom or cohesion). Therefore, an implementation of a coordinator is free to communicate with participants in any order it wants and any requirement on ordering cannot be enforced within the BTP and so should be avoided by applications or services.

Note that participants in atoms and cohesions are identical and can therefore be enrolled in atoms or cohesions. The context information propagated to services contains sufficient information for a participant to determine within the BTP whether they have enlisted in an atom or a cohesion.

Interposition of coordinators is possible within BTP. Although not prevented in the specification, mixing of the two transaction types in a transaction hierarchy would be difficult to manage, simply because of the differences between atoms and cohesions. In fact, although interposition of atoms is relatively straightforward to reason about (after all, it's similar to interposition in traditional transaction systems), interposition of cohesive transaction coordinators is less straightforward to understand and manage, simply because business-level decisions play a prominent role in the way in which cohesions are terminated. So, for example, if a root cohesion coordinator tells a subordinate cohesion coordinator to confirm, does that mean that all of the enlisted participants with that subordinate should also confirm? The answer may well depend upon which other participants the root coordinator confirmed, but unfortunately this information is not made available

to subordinates within the standard protocol.

You saw earlier how interposition is an important requirement for Web services transactions. However, we also discussed that the notion of parent-child relationships (scopes) is important, especially when structuring large-scale applications from disparate services and domains. Unfortunately BTP does not support nested scopes (nested atoms or cohesions).

Open-top completion and business logic

As we discussed previously, a participant within a business transaction may need to support provisional or tentative state changes during the course of the transaction. Such parties must also support the completion of a business transaction either through confirmation (final effect) or cancellation (counter-effect). In general, what it means to confirm or cancel work done within a business transaction will be for the participant to determine. As such, BTP does not define how prepare, cancel, or confirm should be implemented. This is important because BTP relaxes entirely the durability and isolation aspects of traditional transactions, and this means that, unlike in a traditional transaction, it is entirely possible for concurrent users to interfere or see partial results. Enforcement of such policies is outside the scope of BTP and unfortunately, the protocol does not give any support for standardized mechanisms to assist developers and users.

Being able to control the time between the two phases of the termination protocol is extremely important to BTP. Because there is no implied semantic on a participant's prepare, confirm, or cancel operations, they *typically* become part of the business logic in BTP. When a participant is told to prepare in BTP it makes sense for the participant to perform some business logic. Returning to our flight reservation example: using the open-top completion protocol you can visit each flight reservation center within its own atom and ask for a quote; if you wish to retain the quote until you have determined the best option, then you would prepare the corresponding atom; this prepare would then need to understand the semantics of the work you have performed (obtaining the quote) and translate that into a tentative hold on the corresponding seat.

Although business-level semantics are not required to be associated with the individual participant operations, the explicit control over the time between phases is often cited as the main advantage of the BTP open-top approach. For example, the application has time to choose between alternate tasks that have been prepared before ultimately terminating the transaction. However, as you'll see later, this opening up of the two-phase protocol to allow application time to be "injected" does not really work well in the Web services environment.

Qualifiers

An interesting approach taken by BTP to that of loosely-coupled domains and long-running interactions was of introducing the notion of Qualifiers to the protocol. A Qualifier can be thought of as a caveat to that aspect of the protocol on which it is associated. Essentially a Qualifier is a way of providing additional extended information within the protocol.

Although the BTP specification provided some standard Qualifier types (such as timeouts for how long a participant is willing to remain in a prepared state), it is possible to extend them and provide new implementations that are better suited to the application or participant. Obviously any use or reliance on non-standard Qualifiers will reduce application portability.

Unfortunately, although the concept underlying Qualifiers is sound, their implementation with BTP is flawed. The main reason for this is that in some cases the information contained within Qualifiers is not made available to the entity that can best make use of it. For example, one of the standard Qualifiers in BTP is used during the prepare phase and allows a participant to specify how long it is willing (or able) to remain in a prepared state (and possibly what state it will then transit to). This information is passed to the coordinator, but in reality it is the application that requires it.

Protocol optimizations

There are several optimizations to the basic BTP protocol that are worth considering, especially in light of the open-top completion protocol:

- *Participant resignation*: in a traditional two-phase commit protocol, in addition to indicating success or failure during the preparation phase, a participant can also return a *read-only* response; this indicates that it does not control any work that has been modified during the course of the transaction and therefore does not need to be informed of the transaction outcome. This can allow the two-phase protocol to complete quickly since a second round of messages is not required. The equivalent of this in BTP is for a participant to resign from the transaction (atom or cohesion) it was enrolled in. Resignation can occur at any time up to the point where the participant has prepared and is used by the participant to indicate that it no longer has an interest in the outcome of the transaction.
- *Autonomous participant decisions*: In a traditional two-phase protocol a participant enrolls with a transaction and waits for the termination protocol before it either confirms or cancels. You saw earlier how, in order to achieve consensus, it is necessarily a blocking protocol. Modern transaction-processing systems have augmented two-phase commit with *heuristics*, which allow prepared participants to make unilateral decisions about whether they will commit or roll back. Obviously if a participant makes a choice that turns out to be different to that taken by other participants, non-atomic behavior occurs. BTP has its equivalent of heuristics, allowing participants to make unilateral decisions as well. However, unlike in other transaction implementations, the protocol allows a participant to give the coordinator prior knowledge of what that decision will be and when it will be taken. A participant may prepare and present the coordinator with some caveats (the aforementioned Qualifiers) as to how long it will remain in this state and into what state it will then migrate (for example, "will remain prepared for 10 days and then will cancel the seat reservation"). This information may then be used by the coordinator to optimize message exchange. Although this might sound like a good idea, as we mentioned earlier, the ideal end-point for this sort of information is the application and not the transaction; unfortunately BTP does not provide a means whereby the application can obtain this information.
- *Carrier optimizations*: Typically a participant is enlisted with a BTP transaction when a service invocation occurs. When the service request completes, the response is sent back to the initiator of the request. In some circumstances, it may be possible to compound many of the above messages into a "one-shot" message. For example, the service invocation may cause a state change to occur that means the participant can prepare immediately after the invocation completes. Rather than have to wait for an explicit coordinator message, BTP allows the enroll request and statement of preparation to be compounded within the service response. The receiver is then responsible for ensuring that this additional information is forwarded to the coordinator. (Not necessarily a straightforward operation.)
- *One-phase*: If an atom or cohesion coordinator has only a single participant when it is told to confirm, then it can tell the participant to confirm without having to previously prepare.

Web services and BTP

BTPs approach to Web services is also different to what you might expect. From the outset, the technical committee decided that BTP should be useful outside of Web services. As such, BTP is not Web services-specific; it does not leverage the Web services architecture, contains no WSDL or carrier protocol binding. What this means is that rather than place a requirement on a specific mechanism, BTP chose to define a complete service stack within the transaction protocol.

Unfortunately, mapping BTP into a specific deployment environment, such as Web services, may mean that certain aspects of that stack are not necessary; there is also the potential that the "native" functionality may even interfere with the transaction protocol. For example, the one-shot optimization discussed earlier is meant to allow multiple related BTP protocol messages to be sent back to some end-point in a single carrier message. Most modern-day Web services infrastructures already support this kind of optimization transparently (one-shot requires support from the *BTP infrastructure* at both the sender and receiver).

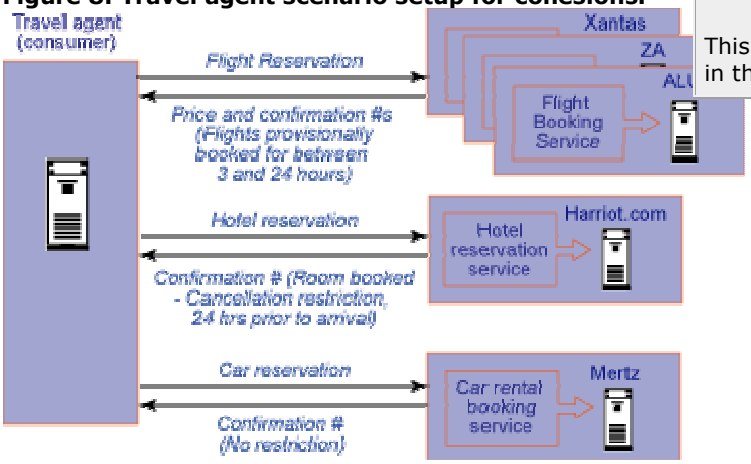
Interestingly all that BTP mandates is the XML message set that is required to conduct the protocol. How that message set is exchanged by the different parties involved may be a function of the

environment (such as email or HTTP), type of business relationship, or other business or logistical factors. The specification does define a binding to SOAP-over-HTTP, but this is not mandated. There is no base interoperability definition for BTP (for the protocol behavior). It is merely a standardization of message content and message sequences.

Example of a cohesive transaction

Let's look at the travel agent scenario and see how cohesions may be utilized, as illustrated in Figure 8. In this example, the travel agent chooses to start a transaction and book a flight to London. One flight option is direct on ALU and the other has two legs and two different carriers ZA and Xantas. Eventually the travel agent has to decide on one of the flights -- either the direct ALU flight or the combined Xantas/ZA flight. By getting commitments for both the ALU flight or the combined Xantas/ZA flight, the travel agent can decide which to take knowing that they will always get the flight they decide upon.

Figure 8. Travel agent scenario setup for cohesions.

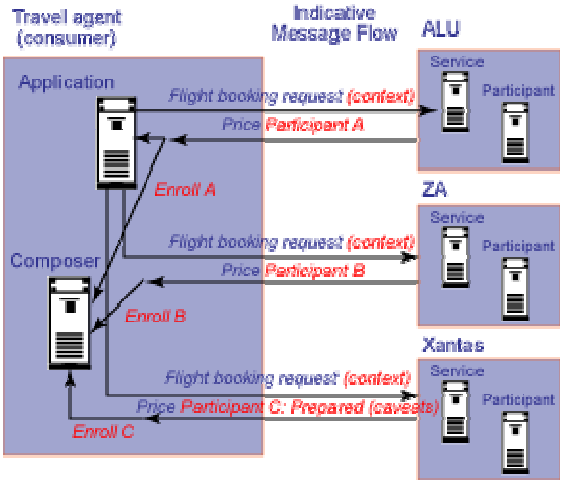


The travel agent scenario.

This example is based on that found in the BTP Primer (see Resources).

If you look at one individual invocation (shown in Figure 9) you can see how commitments are made, managed, and coordinated toward termination. First the travel agent creates a business transaction (*Context*) for the work it wants to perform. It does this through a Composer (the BTP name for a cohesion coordinator). The travel agent then makes the service requests to Xantas.com ALU.com and ZA.com, also propagating the transaction details (*Context*).

Figure 9. Service invocations and context.



Xantas.com ALU.com and ZA.com (*Participants*) all agree to participate in the transaction (Enroll). In this example Xantas also makes a commitment to the transaction (*Prepared*) but ALU and ZA do not.

As shown in Figure 10, based on the prices returned, the travel agent decides to go ahead and book the two-legged flight offered by Xantas and ZA (*Confirm B,C*). Because ALU never made a commitment to the business transaction (*Prepared*), in other words, reserved seats; there is no need to cancel the ALU flight.

Figure 10. Confirming flights.

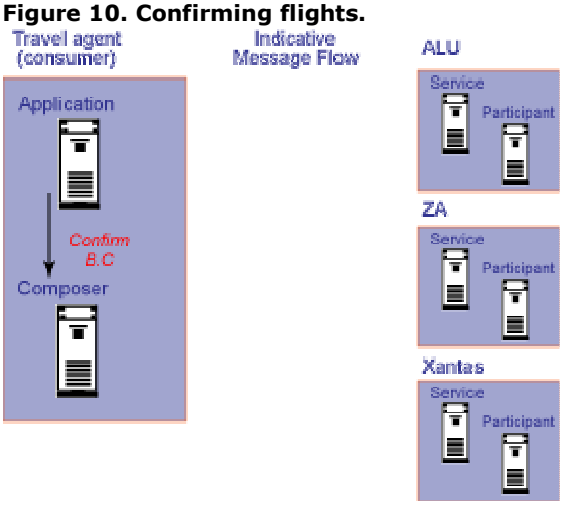
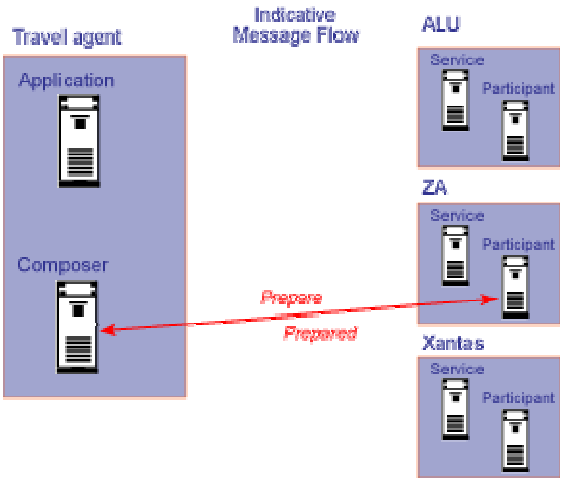


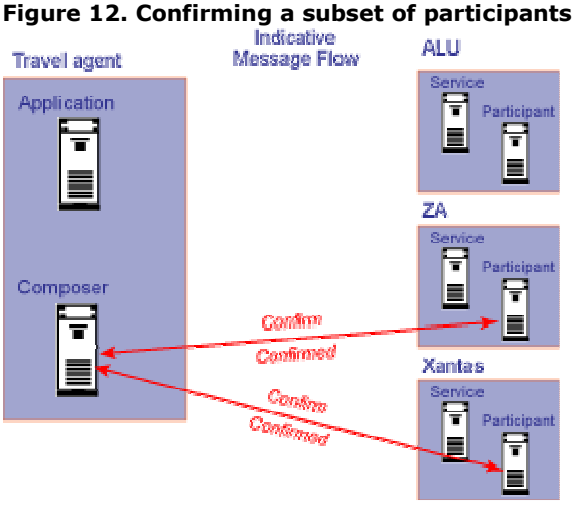
Figure 11 shows that because the flight chosen involves two parties, Xantas and ZA, the transaction the coordinator then asks each participant to make a commitment with regard to the overall business transaction (*Prepare*). Because Xantas has already made a commitment, the coordinator only needs to get a commitment from ZA (*Prepare*).

Figure 11. Preparing the participants.



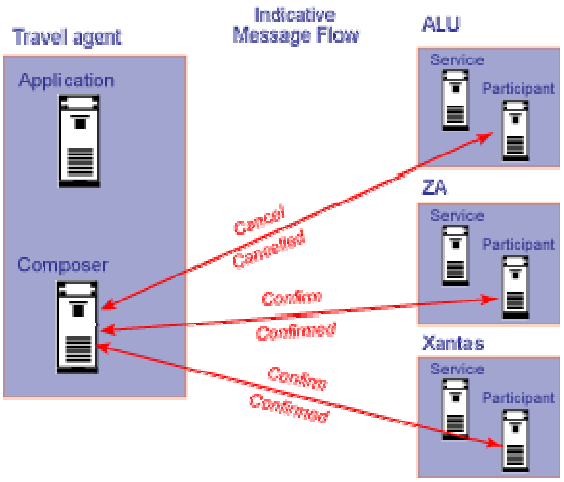
The composer now has received positive commitments from Xantas.com and ZA.com, the requested portions of the business transaction requested by the travel agent. The composer therefore goes ahead and confirms the seat reservations offered by ZA.com and Xantas.com, as shown in Figure 12.

Figure 12. Confirming a subset of participants in the cohesion.



If ALU had made a commitment (*Prepared*) then the composer would need to explicitly cancel the seats reserved by ALU as part of the business transaction, at the same time as confirming the ZA, Xantas flight. The composer finally confirms the successful conclusion of the business transaction back to the travel agent (*Transaction Confirmed*), as illustrated by Figure 13.

Figure 13. Confirming a subset of participants in the cohesion and cancelling others.



As you can see from this example, in BTP, because business logic is encoded within the transaction protocol, it essentially means that a user has to be closely tied to (or perhaps even be) the coordinator. Business information, such as the ability for a participant to remain *prepared* (for example, hold onto a hotel room) for a specific period of time is propagated from the participant to the coordinator, but there is nothing within the protocol to allow this information to filter up to the application/client where it really belongs.

In order to use cohesions it is also necessary for Web services to expose back-end implementation choices about participants. As you saw, in order to parameterize the two-phase completion protocol, the terminator of the cohesion obviously needs to be able to say "prepare A and B and cancel C and D," where A, B, C and D are participants that have been enrolled in the cohesion by services (such as our flight reservation system). Naturally this is something that programmers may not be comfortable with, and it goes against the Web services orthodoxy.

Web Services Coordination and Transaction

This section will examine the overall model used by the Web Services Coordination and Web Services Transactions specifications. This is important in order to understand the differences between WS-C/WS-Tx and BTP.

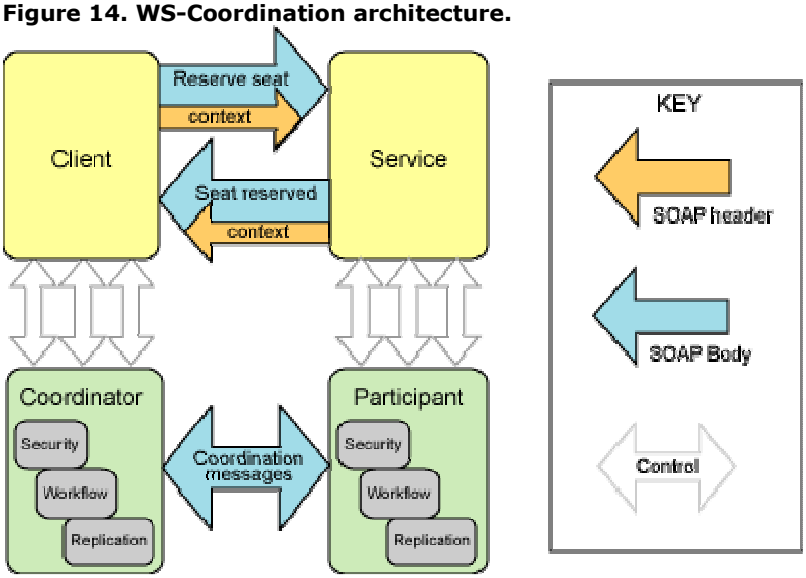
Coordination is a requirement in a variety of different aspects of distributed applications, such as workflow, security, atomic transactions, caching and replication, security, auctioning, and business-to-business activities. For example, coordination of multiple Web services in choreography may be required to ensure the correct result of a series of operations comprising a single business transaction.

Despite the fact that there are many different types of application that require coordination, each use typically manifests as a different type of coordination protocol. In the case of transactions, for example, BTP, Object Management Group’s Object Transaction Service, Microsoft DTC are solutions to specific problem domains and which are not applicable to others since they are based on different architectural styles. Given the domain-specific nature of these coordination protocols, it is unrealistic to provide a "universal" protocol without jeopardizing efficiency and scalability.

Unlike BTP which ties coordination to transactions, the Web Services Transactions protocol leverages a separate protocol aimed solely at outcome determination/processing: Web Services Coordination. The fundamental idea underpinning WS-Coordination is that there is a generic need for a coordination infrastructure in a Web services environment. The WS-Coordination specification defines a framework that allows different coordination protocols to be plugged-in to coordinate work

among clients, services, and participants, as shown in Figure 14.

Figure 14. WS-Coordination architecture.



Note that the Control messages are shown separately only to illustrate the specific interactions between client/coordinator and service/participant. These messages are still Web services messages and hence flow using SOAP.

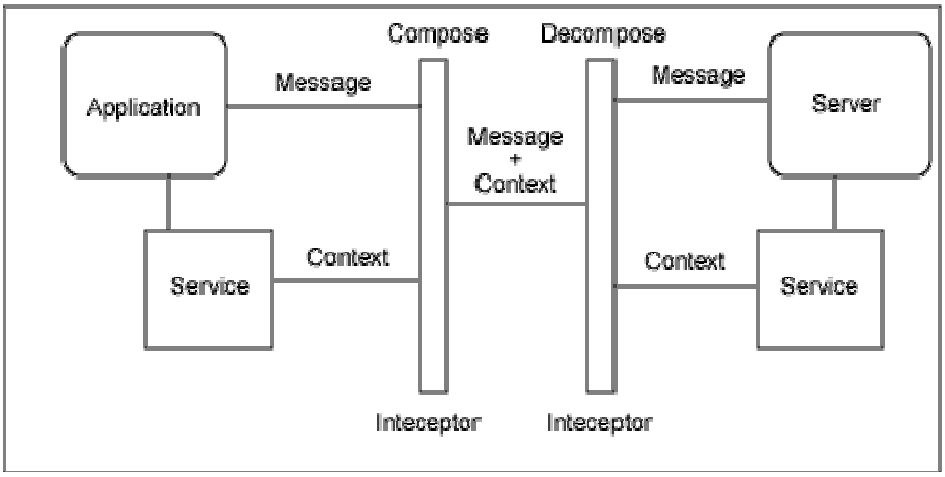
Both WS-C and WS-Tx are intended solely for the Web services environment and as such leverage existing and evolving standards, such as WSDL, WS-Addressing, Web Services Security, and WS-Policy (see Resources). This focuses WS-C and WS-Tx to a WebServices environment and simplifies the specifications and places them as a component in the Web Services architecture. Any advances in performance optimizations for Web services infrastructures can be automatically leveraged by these specifications.

The XML context

In order for coordination to span a distributed number of services/tasks, certain information has to flow between the sites/domains involved in the application. This is commonly referred to as the *context* and typically contains the following information:

- An identifier which guarantees global uniqueness for an individual activity (such an identifier can also be thought of as a *correlation* identifier, or a value that is used to indicate that a task is part of the same work activity).
- The coordinator location or endpoint address so participants can be enrolled.

Figure 15. Services and context flow.



The context information is propagated to provide a flow of context information between distributed execution environments, for example using SOAP header information. This may occur transparently to the client and application services. As has already been mentioned, the context is propagated as part of normal message interchange within an application (for example, as an additional part of the SOAP header).

An important difference between WS-Tx and BTP is that the former differentiates between transactionality requirements and coordination by leveraging the WS-C protocol, whereas the latter ties coordination to transactions. The following section will examine WS-C in order to better understand the type of flexibility this gives WS-Tx over other approaches.

Coordination protocol

Coordination is the act of one agent (the *coordinator*) disseminating information to a number of participants to guarantee that all participants obtain a specific message. A coordinator can also be a participant, creating a tree of sub-coordinators or peer-coordinators that cooperate to further propagate the result. Unlike BTP, interposition is an integral part of the WS-C (and WS-T) models.

Context information flows implicitly (transparently to the application) within normal messages sent to the participants. This information is specific to the type of coordination being performed, for example to identify the coordinator(s), the other participants in an activity, recovery information in the event of a failure, etc. Furthermore, it may be required that additional application-specific context information (for example, extra SOAP header information) flow to these participants or the services which use them.

WS-Coordination defines a generic coordination framework that can support arbitrary coordination protocols. It is extensible at the coordinator level as well as at the level of the context. For example, a coordinator that executes a three-phase commit protocol can be easily plugged in to a WS-C implementation and the basic WS-C context may be enhanced if necessary. The framework is useful for propagating a range of context types including security, workflow, or replication.

The WS-Coordination specification talks in terms of *activities*, which are distributed units of work, involving one or more parties (which may be services, components, or even objects). At this level, an activity is minimally specified and is simply *created*, made to *run*, and then *completed*.

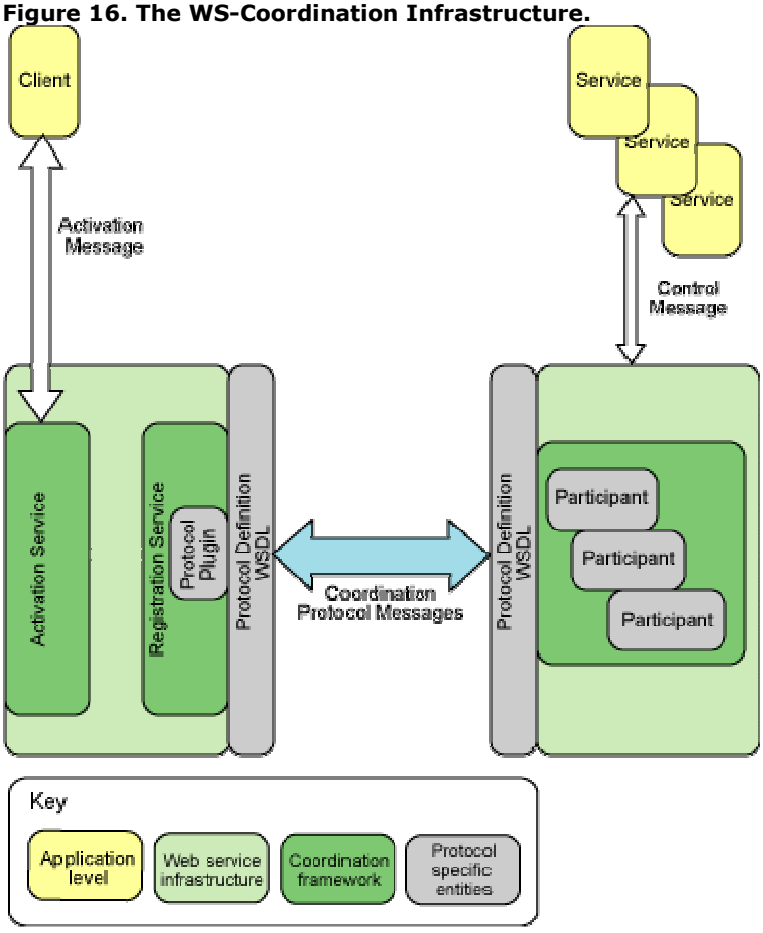
Whatever coordination protocol is used, the same requirements are present:

- Instantiation (or activation) of a new coordinator for the specific coordination protocol, for a particular application instance

- Registration of participants with the coordinator, such that they will receive that coordinator's protocol messages during (some part of) the application's lifetime
- Propagation of contextual information between Web services that comprise the application
- An entity to drive the coordination protocol through to completion.

The first three of these points are directly the concern of WS-Coordination while the fourth is defined in WS-T, usually the client application that controls the application as a whole. These four WS-Coordination roles and their interrelationships are shown in Figure 16.

Figure 16. The WS-Coordination Infrastructure.



The WS-Coordination framework exposes an *Activation Service* which supports the creation of coordinators for specific protocols and their associated contexts. The process of invoking an activation service is illustrated as occurring asynchronously, so the specification defines both the interface of the activation service itself and that of the invoking service: the activation service can call back to deliver the results of the activation, namely a context that identifies the protocol type and coordinator location.

This asynchronous approach reduces the tight coupling between end-points typically seen in other environments, which has the advantage of improved fault-tolerance, modularity, and deployment considerations. For example, although a client may send a completion message to a coordinator, it may make more sense for the response to be sent to some other entity.

Once a coordinator has been instantiated and a corresponding context created by the activation service, a *Registration Service* is created and exposed. This service allows participants to register to

receive protocol messages associated with a particular coordinator. Like the activation service, the registration service assumes asynchronous communication and so specifies WSDL for both registration service and registration requester.

The context is critical to coordination since it contains the information necessary for services to participate in the protocol. It provides the glue to bind all of the application's constituent Web services together into a single coordinated application whole. Since WS-Coordination is a generic coordination framework, contexts have to be tailored to meet the needs of specific coordination protocols that are plugged into the framework. The format of a WS-Coordination context is specifically designed to be third-party extensible, and its contents are as follows:

- A coordination identifier with guaranteed global uniqueness for an individual coordinator in the form of a URI
- An address of a registration service endpoint where parties receiving a context can register participants into the protocol
- A time-to-live value which indicates for how long the context should be considered valid
- Extensible protocol-specific information particular to the actual coordination protocol supported by the coordinator.

This is shown Figure 17, where the schema states that a context consists of a URI that uniquely identifies the type of coordination that is required (`xs:anyURI`), an endpoint where participants to be coordinated can be registered (`wsu:PortReferenceType`), and an extensibility element designed to carry specific coordination protocol context payload (`xs:any`), which can carry arbitrary XML payload.

Figure 17. WS-Coordination Context Schema Fragment.

```
<xs:complexType name="CoordinationContextType" abstract="false" >
  <xs:complexContent>
    <xs:extension base="wsu:ContextType">
      <xs:sequence>
        <xs:element name="CoordinationType"
            type="xs:anyURI" />
        <xs:element name="RegistrationService"
                                    type="wsu:PortReferenceType" />
        <xs:any namespace="##any" processContents="lax"
            minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

We discussed earlier how the OASIS Business Transactions Protocol coordinates participants in either atomic or cohesive transactions in order to achieve consensus. The protocol defined in the BTP specification is an open-top, two-phase completion protocol. However, there is no separation between transactions and coordination in BTP, and all of the protocol assumes two-phase. Attempting to change the type of coordination protocol (for example, to a three-phase protocol)

would require significant modifications to the specification and affect all aspects of coordination and transactions.

However, it is worth noting that it is entirely possible to integrate BTP within WS-C.

Transaction protocol

The WS-Transaction specification plugs into WS-C and proposes two common industry completion patterns (specific coordination protocols), where each supports the semantics of a particular kind of business-to-business interaction:

- *Atomic Transaction (AT)*: This is meant to map to existing transaction standards which have a well-defined behavior for atomicity (well-formed and two-phase), isolation (no dirty reads, repeatable reads) and durability (no lost data), in other words, traditional ACID semantics. The important thing to remember when considering Web services is that they are for interoperability as much as they are for the Web. In the past, making traditional transaction systems talk to one another was a holy grail that was rarely achieved; Web services offer unparalleled support for interoperability in this regard. Traditional transaction systems already form the backbone of enterprise-level applications and will continue to do so for the Web services equivalent. Business-to-business activities will typically involve back-end transaction processing systems either directly or indirectly and being able to tie together these environments will be the key to the successful take-up of Web services transactions (see Resources for an interesting discussion by Vasters and the reference on Interoperability). Finally, AT is useful if only for intra-domain environments where a customer needs to consolidate operations across any number of internal applications. For example, a merger may have resulted in the need to combine the apps of the old and new business.
- *Business Activity (BA)*: This provides flexible transaction properties and is designed specifically for long-duration interactions, where holding on to resources is impossible or impractical. In this model, services are requested to do work (for example, reserving a seat on a flight), and if they can do so in a manner where that work can be later undone, the service may inform the BA. In this way, if the BA later decides it needs to cancel the work, it can inform the service. How services do their work and provide compensation mechanisms is not the domain of the WS-Tx specification: this is an implementation decision for the service provider.

Protocol extensibility.

WS-Tx is meant to be a portfolio allowing other patterns to be added as requirements are defined, such as sessions, delegated transactions, etc.

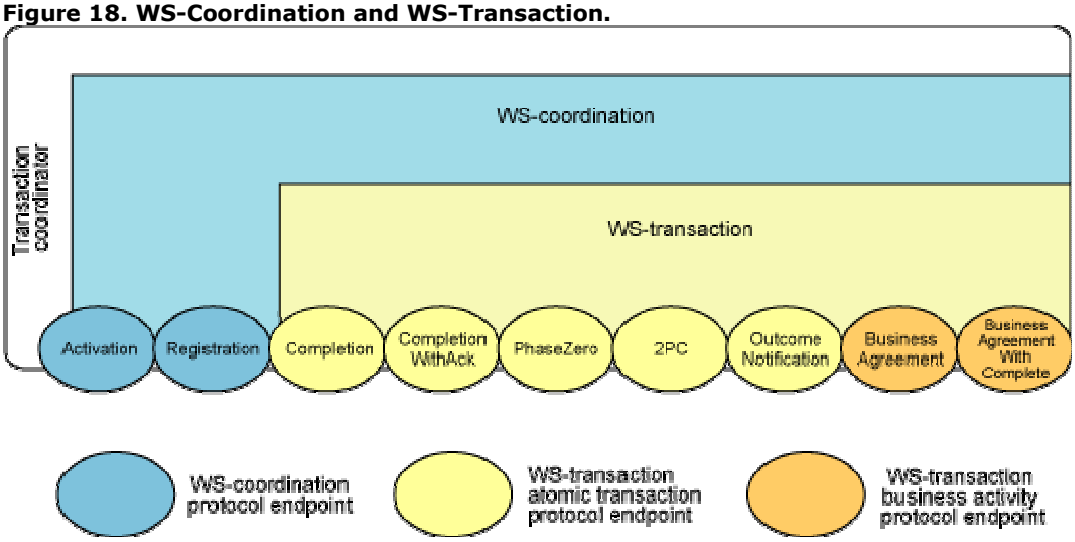
It is important to note that the BA model derives from a specific industry requirement in the BPEL4WS specification (see Resources for the specification). Although ATs and BAs may be sufficient for the current use cases that the specifications are aimed at, it is generally accepted that other protocols may well be needed later. Because WS-Tx leverages WS-C, new protocols can be added to the specification as and when the need arises. Therefore, the WS-Tx specification allows growth if new protocols are required (or identified).

This is yet another important distinction between WS-Tx and BTP: whereas WS-Tx admits the possibility that "one-size doesn't fit all" and other protocols may need to be supported later, the BTP specification is essentially closed and constrained by its two-phase protocol. By making the separation between coordination and transactions explicit within WS-C and WS-T, adding new transaction protocols should be relatively straightforward and not impinge on those that already exist. Unfortunately attempting to do the same with BTP could potentially result in an entirely new specification, since all of the current protocol is tied to two-phase completion coordination.

An important aspect of WS-Transaction that differentiates it from traditional transaction protocols is that a synchronous request/response model is not required. This model derives from the fact that WS-Transaction is, as you see in Figure 18, layered upon the WS-Coordination protocol whose own

communication patterns are asynchronous by default, but can support other interaction patterns.

Figure 18. WS-Coordination and WS-Transaction.



WS-Transaction leverages the context management framework provided by WS-Coordination in two ways. First of all it extends the WS-Coordination context to create a transaction context. Secondly, it augments the activation and registration services with a number of additional services:

- (Completion, CompletionWithAck
- PhaseZero
- 2PC
- OutcomeNotification
- BusinessAgreement
- BusinessAgreementWithComplete)

and two protocol message sets (one for each of the transaction models supported in WS-Transaction).

The Atomic Transaction protocol (AT)

The Atomic Transaction (AT) protocol is a consensus group that enforces strict atomicity among its participants. It is wrong to talk about Atomic Transactions violating the "trust chasm" between Web services; this ignores the central reason for using ATs: interoperability and short-duration interactions. There is a place for traditional transaction systems in Web services and this is precisely what Atomic Transactions are concerned with.

To begin an atomic transaction, the client application may locate a coordinator that supports WS-Transaction. Once located, the client sends a CreateCoordinationContext message to the activation service specifying http://schemas.xmlsoap.org/ws/2002/08/wstx as its coordination type and will get back an appropriate WS-Transaction context. The transaction context has its CoordinationType element set to the WS-Transaction AT namespace and also contains a reference to the atomic transaction coordinator endpoint (the WS-Coordination registration service) where participants can be enlisted.

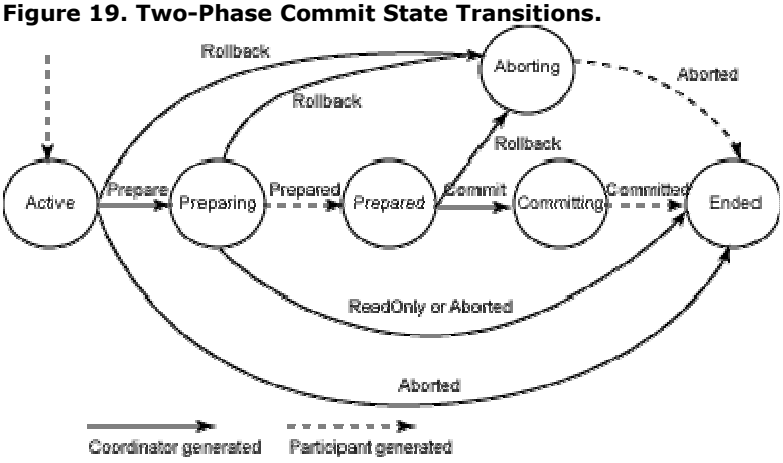
After obtaining a transaction context from the coordinator, the client application then proceeds to interact with Web services to accomplish its business-level work. With each invocation on a business service, the client propagates the context, such that the each invocation is implicitly scoped by the transaction.

Once all the necessary application-level work has been completed, the client can terminate the transaction. To do this, the client application registers its own participant for the Completion or CompletionWithAck protocol. Once registered, the participant can instruct the coordinator either to try to commit or roll back the transaction.

Transaction termination normally uses the two-phase commit protocol (2PC), as described earlier and illustrated in Figure 1. As with BTP, there is no ordering of 2PC participant invocations implied by the WS-Tx specification. If a transaction involves only a single participant, WS-Transaction supports a one-phase commit optimization similar to that in traditional transaction systems (and as you saw earlier, in BTP). Since there is only one participant, its decisions implicitly reach consensus, and so the coordinator need not drive the transaction through both phases. In the optimized case, the participant will simply be told to commit, and the transaction coordinator need not record information about the decision since the outcome of the transaction is solely down to that single participant.

Figure 19 shows the state transitions of a WS-Transaction atomic transaction and the message exchanges between coordinator and participant; the coordinator-generated messages are shown in the solid line, whereas the participant messages are shown by dashed lines.

Figure 19. Two-Phase Commit State Transitions.



Once the coordinator has finished with the transaction, the Completion or CompletionWithAck protocol that originally began the termination of the transaction can complete and inform the client application whether the transaction was committed or rolled back. Note that the CompletionWithAck protocol insists that the coordinator must remember the outcome until it has received acknowledgment of the notification from the participant.

In addition to the 2PC protocol, WS-Tx also provides two other protocols: PhaseZero and OutcomeNotification. Accessing durable storage (whatever its implementation) is often the performance bottleneck, and hence caching of an object's state (such as an entire database table) and operating on that cached state for the duration of a transaction can significantly improve performance over the alternative of continually going back and forth to the database. However, there is obviously a need to force that state back to the original persistence store *prior* to the transaction committing.

In traditional transaction systems this is accomplished through *synchronization participants*. Synchronizations are informed that a transaction is *about* to commit. They are then informed when the transaction has completed and in what state it completed. Synchronizations essentially turn the two-phase commit protocol into a four-phase protocol:

- Before the transaction starts the two-phase commit, all registered synchronizations are informed. Any failure at this point will cause the transaction to roll back.

- The coordinator then conducts the normal two-phase commit protocol.
- Once the transaction has terminated, all registered synchronizations are informed. However, this is a courtesy invocation because any failures at this stage are ignored: the transaction has terminated so there's nothing to affect.

When an Atomic Transaction is terminating, the associated coordinator first executes the PhaseZero protocol if any participants registered for it. All PhaseZero participants are told that the transaction is about to complete and they can respond with either the PhaseZeroCompleted or Error message; any failures at this stage will cause the transaction to roll back.

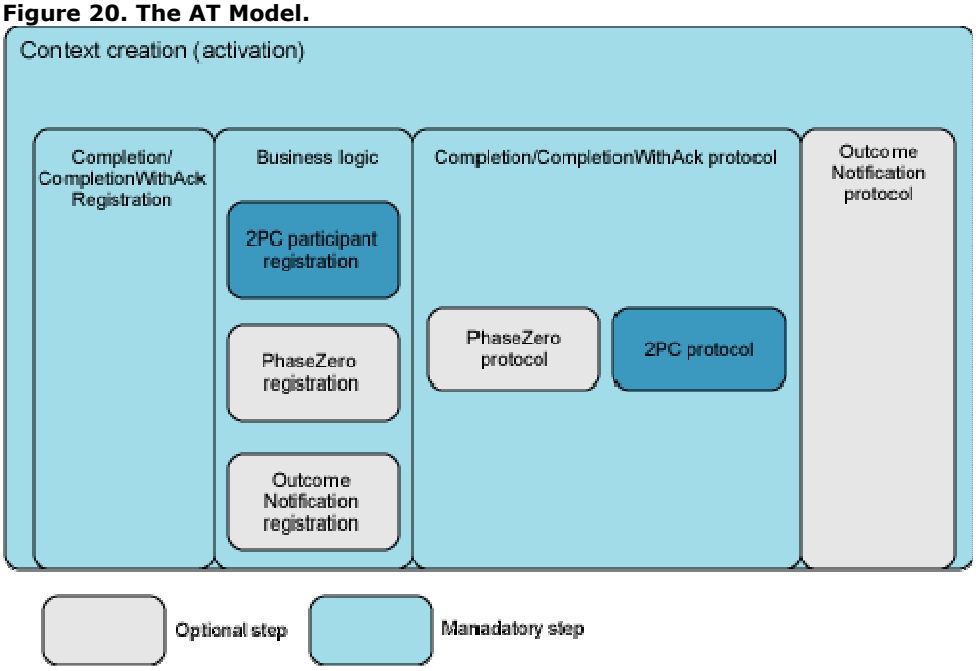
Additionally, some services may have registered an interest in the completion of a transaction and they will be informed through the OutcomeNotificaton protocol after 2PC has completed. Any registered OutcomeNotification participants are invoked after the transaction has terminated and are told the state in which the transaction completed (the coordinator sends either the Committed or Aborted message). Since the transaction has terminated, any failures of participants at this stage are ignored -- OutcomeNotification is essentially a courtesy and has no bearing on the outcome of the transaction.

The fact that there are distinct protocols for synchronization and two-phase commit is as important in AT as it is in traditional transaction systems. Being able to rely upon the order in which certain types of participants will be invoked allows performance optimizations, such as caching, to be supported.

As you saw earlier, BTP has only one type of participant that can be enlisted in an atom or a cohesion, and neither protocol supports any kind of relative ordering. Hence providing an equivalent to synchronizations is not possible within the scope of vanilla BTP.

Finally, after having gone through each of the stages in an AT, you can now see the intricate interweaving of individual protocols that goes to make up the AT as a whole in Figure 20.

Figure 20. The AT Model.



There is another fundamental difference between the AT model and the BTP atom model to which it is often compared: the termination protocol is not open-top and hence the distinction between participants and services is well-defined. The termination protocol does not mix business level

decisions into the commit protocol, overloading what it may mean for a participant to receive a prepare request, for example.

The reason for this is that Web services are typically written to operate in the following way:

- A service receives a document requesting it to perform some work (such as reserving a seat on a specific flight).
- Later that service may be sent another document requesting it to either undo the work or accept it.

If the work is being performed within the scope of a transaction (let's assume it's an atomic transaction), then the interaction between the application and the transaction service should be minimal -- the transaction coordinator only requires access to the participants and they should not require strong interactions with the services on whose behalf they operate.

In the scenario of a flight reservation service, the business-level (service) methods, such as book seat, have already performed the necessary work (such as provisionally reserving the seat). The explicit prepare operation of the open-top protocol is simply not required to have business semantics. The assumed advantages of an open-top approach (allowing decision time between the two-phases) are not required. When the application decides to terminate the business transaction, it wants the work to happen (or not) immediately, and all that is required is to guarantee consensus between the participants.

Business Activities (BA)

Most business-to-business applications require transactional support in order to guarantee consistent outcome and correct execution. These applications often involve long-running computations, loosely-coupled systems, and components that do not share data, location, or administration, and it is difficult to incorporate atomic transactions within such architectures. For example, an online bookshop may reserve books for an individual for a specific period of time, but if the individual does not purchase the books within that period they will be "put back onto the shelf" for others to buy. Furthermore, because it is not possible for anyone to have an infinite supply of stock, some online shops may appear to users to reserve items for them, but in fact may allow others to pre-empt that reservation (in other words, the same book may be "reserved" for multiple users concurrently); a user may subsequently find that the item is no longer available, or may have to be reordered specially for them.

A business activity or BA is designed specifically for these kinds of long-duration interactions, where exclusively locking resources is impossible or impractical. In this model, services are requested to do work, and where those services have the ability to undo any work, they inform the BA such that if the BA later decides to cancel the work, it can instruct the service to execute its undo behavior.

While the full ACID semantics are not maintained by a BA, consistency can still be maintained through compensation, though the task of writing correct compensating actions (and thus overall system consistency) is delegated to the developers of the services under control of the BA. Such compensations may use backward error recovery, but will typically employ forward recovery.

The WS-Transaction Business Activity simply defines a protocol for Web services-based applications to enable existing business processing and workflow systems to wrap their proprietary mechanisms and interoperate across implementations and business boundaries.

Central to WS-Tx is the notion of *scopes* and defining activity-to-task relationships. A business activity may be partitioned into scopes, where a scope is a business task or unit of work using a collection of Web services. Such scopes can be nested to arbitrary levels, forming parent and child relationships. A parent scope has the ability to select which child tasks are to be included in the overall outcome protocol for a specific business activity, and so clearly non-atomic outcomes are possible. A Business Activity defines a consensus group that allows the relaxation of atomicity based on business-level decisions. In a similar manner to traditional nested transactions, if a child task

experiences an error, it can be caught by the parent who may be able to compensate and continue processing.

As you saw earlier, although BTP supports interposition, it does not support nesting of scopes. This is an important difference between WS-Tx Business Activities and BTP. Nested scopes are important for a number of reasons, including:

- *Fault-isolation*: If sub-scope fails (for example, because a service it was using fails) then this does not require the enclosing scope to fail, thus undoing all of the work performed so far.
- *Modularity*: if there is already a scope associated with a call when a new scope is begun, then the scope will be nested within it. Therefore, a programmer who knows that a service requires scopes can use them within the service. If the service's methods are invoked without a parent scope, then the service's scopes will simply be top-level; otherwise, they will be nested within the scope of the client.

When a child task completes it can either leave the business activity or signal to the parent that the work it has done can be compensated later. In the latter case, the compensation task may be called by the parent should it ultimately need to undo the work performed by the child.

Unlike the Atomic Transaction protocol model, where participants inform the coordinator of their state only when asked, a task within a business activity can specify its outcome to the parent directly without waiting for a request. This feature is useful when tasks fail such that the notification can be used by business activity exception handler to modify the goals and drive processing forward without having to meekly wait until the end of the transaction to admit to having failed -- a well-designed Business Activities should be proactive, if it is to be performant.

Underpinning all of this are three fundamental assumptions:

- All state transitions are reliably recorded, including application state and coordination metadata (the record of sent and received messages).
- All request messages are acknowledged, so that problems are detected as early as possible. This avoids executing unnecessary tasks and can also detect a problem earlier when rectifying it is simpler and less expensive.
- As with atomic transactions, a response is defined as a separate operation and not as the output of the request. Message input-output implementations will typically have timeouts that are too short for some business activity responses. If the response is not received after a timeout, it is resent. This is repeated until a response is received. The request receiver discards all but one identical request received.

As with atomic transactions, the business activity model has multiple protocols: BusinessAgreement and BusinessAgreementWithComplete. However, unlike the AT protocol which is driven from the coordinator down to participants, this protocol is driven much more from the participants upwards.

Under the BusinessAgreement protocol, a child activity is initially created in the Active state; if it finishes the work it was created to do and no more participation is required within the scope of the BA (such as when the activity operates on immutable data), then the child can unilaterally send an exited message to the parent; this is equivalent to the participant resigning from the business transaction as is also supported in BTP. However, if the child task finishes and wishes to continue in the BA, then it must be able to compensate for the work it has performed. In this case it sends a completed message to the parent and waits to receive the final outcome of the BA from the parent. This outcome will either be a close message, meaning the BA has completed successfully, or a compensate message, indicating that the parent activity requires that the child task reverse its work.

The BusinessAgreementWithComplete protocol is identical to the BusinessAgreement protocol with the exception that the child cannot autonomously decide to end its participation in the business activity, even if it can be compensated. Rather the child task relies upon the parent to inform it

when the child has received all requests for it to perform work. The parent does this by sending the complete message to the child. The child then acts as it does in the BusinessAgreement protocol.

As with the AT model, another fundamental difference between the BA model and the BTP cohesion model to which it is often compared is that it does not mix business-level semantics with the transaction protocol. The reason for the BA approach is that it's very similar to what traditional workflow systems do and how most Web services are being written today: the compensation work is simply considered as another activity -- it's not special. The work required to compensate is already available from the service (such as unbook seat), and obviously book seat does the work somehow (and this may well be provisional until the application confirms the seat reservation).

Most workflow systems don't distinguish compensate activities from forward progress activities: an activity is an activity and it just does some work. If that work happens to compensate for some previous work then so be it. In addition, most services you'll find already have compensate operations written into their definitions, like "unbook seat" or "cancel holiday" and they don't need to be driven by some other transaction/coordination engine that then sends "prepare" or "commit" or "roll back" to a participant which then has to figure out how to talk to the service to accomplish the same goal.

Protocol optimizations

There are several optimizations to the WS-Tx protocol that are worth considering, especially in light of their equivalents in BTP:

- *Read-only*: As you saw earlier, in a traditional two-phase commit protocol, a participant can also return a *read-only* response to indicates that it does not control any work that has been modified during the course of the transaction, and therefore does not need to be informed of the outcome. The Atomic Transaction protocol supports this optimization.
- *Flexible consensus groups*: As you have seen, the Atomic Transaction protocol provides a strictly atomic consensus group with well-defined ACID semantics. The Business Activity protocol provides a consensus group that allows for the weakening of atomicity; in addition, because of the activity-scope relationships that can be formed in the BA protocol, it is easier to delineate work into different scopes of consensus.
- *Participant resignation*: The equivalent of read-only optimization in Business Activities is for participants to resign (exit) from the activity. This is similar to the BTP participant resignation optimization. Resignation can occur at any time up to the point where the activity is completing and is used by the participant to indicate that it no longer has an interest in the outcome of the BA.
- *Autonomous participant decisions*: Because the Atomic Transaction protocol is based on the traditional (presumed abort) transaction protocol, it allows participants to make autonomous decisions about their ultimate fate. If these decisions are made before the transaction begins to terminate, then the transaction must roll back. If they happen after the participant has prepared, then the decision may lead to a heuristic (non-atomic) outcome, that must be resolved later.
- *Carrier optimizations*: Unlike BTP, the WS-C and WS-Tx protocols rely mainly upon improvements in the Web services architecture and implementations to provide protocol optimizations such as one-shot. This should not be seen as a deficiency in BTP, but rather a property of firmly placing WS-C and WS-Tx into a single deployment domain. Optimizations of this kind are best dealt with by other architecture layers.
- *One-phase*: The Atomic Transaction protocol supports the one-phase commit optimization.
- *Qualifiers*: Additional qualifications to the protocol are handled by WS-Policy, where Policy is a standardized mechanism to advertise the operational characteristics.

Web services and WS-C/WS-Tx

As we have already stated, the Web Services Coordination and Web Services Transactions specifications are firmly placed on the Web services architecture. As such, they are designed to be able to use other Web services specifications such as security, reliable messaging, etc. when and if

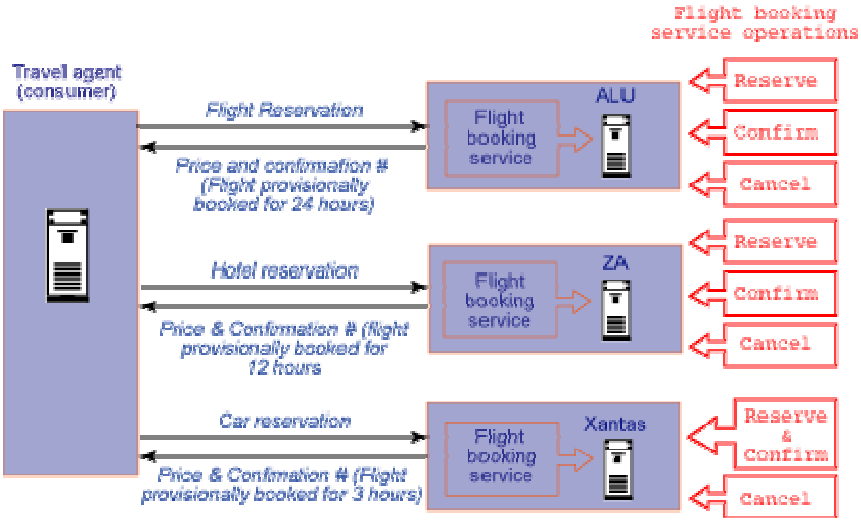
required. However, unlike BTP, these other requirements are clearly delineated within separate specifications.

The travel agent scenario using BAs

This section will show how the travel agent scenario that we previously modeled using BTP cohesions can just as easily be modeled using Business Activities. For simplicity we'll only consider the situation of obtaining several flight quotes and eventually accepting only the cheapest.

This example discusses the requirements of business transactions which need a mechanism to select and manage the tasks that are included in the overall application outcome. Figure 21 essentially reiterates the application configuration: each flight service exposes operations to reserve, confirm, or cancel seats on a specific flight.

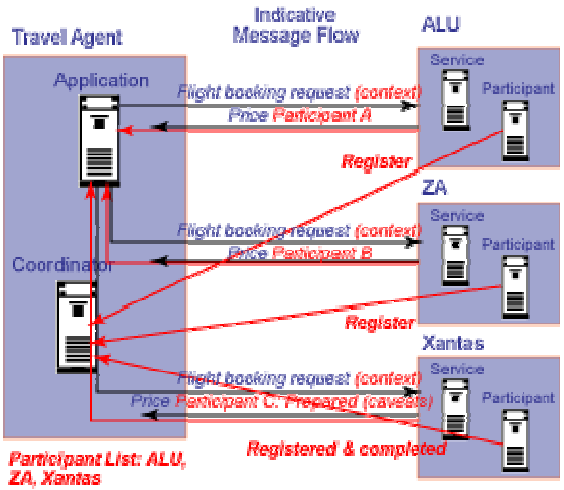
Figure 21. The system configuration.



Note that although not shown in this example, the overall business activity can be comprised of a number of tasks. Each task can be modeled as another business activity within the scope of the overall application. In addition, the task could be implemented as an atomic transaction. WS-Tx allows the application to specify scopes (relationships) without having to build logic within the overall business process to track the relationships.

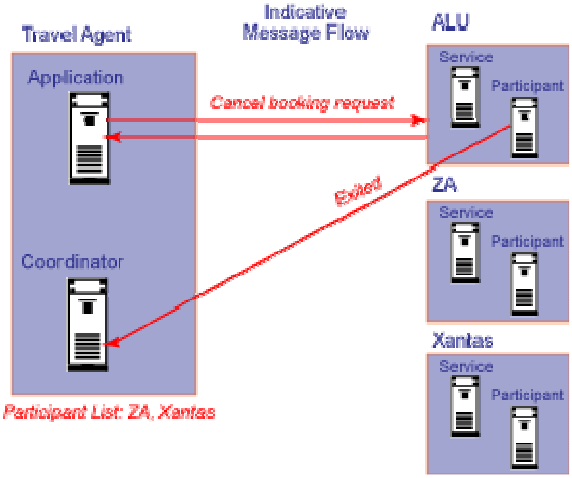
As before, the application makes invocations on each of the services to obtain a quote for a seat on the flight. Xantas.com ALU.com and ZA.com acknowledge the Travel Agent's Flight Booking Requests (or application-level response). ALU and BA provisionally book seats while Xantas actually reserves a seat. Associated with the request, the transaction service manages the tasks that are participating in the applications (in other words, the *Participants* Enroll, indicating the reservation tasks are actively processing, while Xantas also indicates that it has completed the reservation request (shown in Figure 22).

Figure 22. Making the requests.



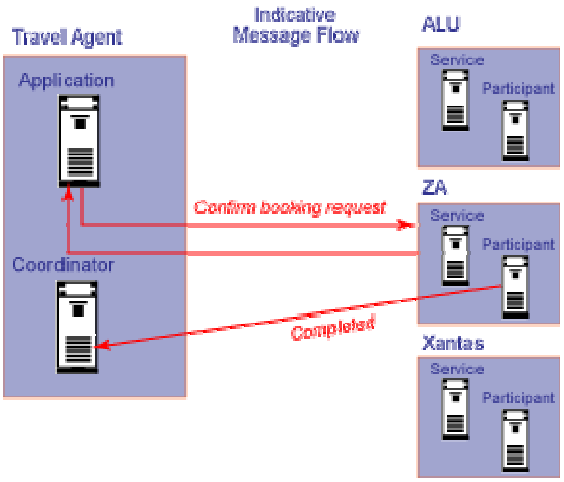
Based on the Prices returned, the Travel Agent decides to go ahead and book the two-legged flight offered by Xantas and ZA, shown in Figure 23. Because ALU never reserved seats, there is no need to cancel the ALU flight. The Travel Agent instructs ALU to cancel the provisional booking. (Optionally the Travel Agent can allow the ALU provisional booking to timeout if the application is constructed with scopes.)

Figure 23. Canceling a quote.



Because the flight chosen involves two parties, Xantas and ZA, the transaction Travel Agent then requests ZA to reserve a seat. ZA acknowledges the reservation. The participant then tells the coordinator that task has completed (Figure 24).

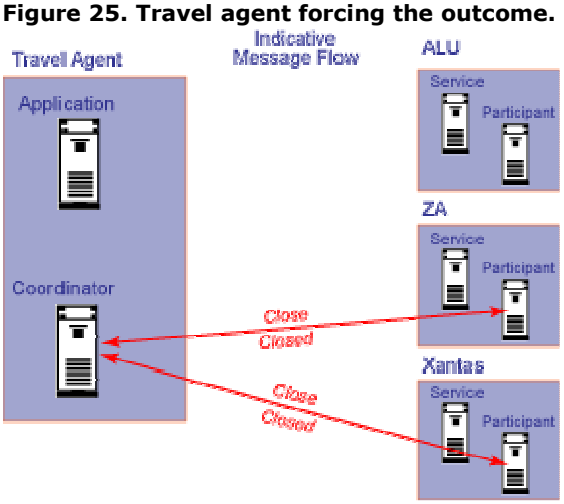
Figure 24. Choosing the right quote.



The application has now chosen the seat reservations that are to be included in the overall booking. You will notice that the final set of participants chosen must terminate atomically. In the example, ZA and Xantas need to make a commitment to the transaction and complete as an atomic set. We could have shown this as an atomic transaction within the scope of the overall business application, but instead we chose to illustrate a more simple scenario where the Travel Agent forces this outcome directly.

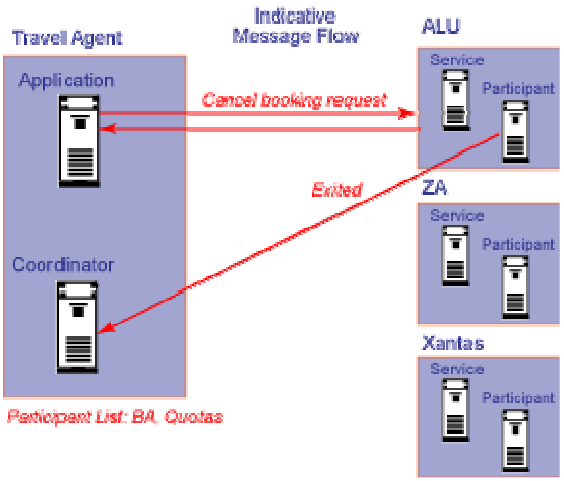
The Coordinator now has received acknowledgements from Xantas.com and ZA.com, and the requested portions of the business transaction Travel Agent have completed. The coordinator therefore goes ahead and confirms (via *close* shown in Figure 25) the seat reservations offered by ZA.com and Xantas.com.

Figure 25. Travel agent forcing the outcome.



If ALU had reserved a seat, then the Travel Agent would need to instruct ALU to cancel the booking. The transaction service would then remove ALU from the tasks participating in the transaction (Participant sends exited to the Coordinator). The Travel Agent would then confirm the reservation for the remaining tasks as shown in Figure 26.

Figure 26. Travel agent confirming the quote.



Comparing and contrasting

Although at first glance it may seem like there is commonality between the two specifications (both support a two-phase completion protocol, for example), as you've seen there are significant differences. This section will re-examine some of the issues that we have already discussed, as well as some that we haven't.

Pros and cons of BTP

As you might expect from a specification that took over a year to develop, on the plus side the BTP specification is well formed and complete. Unfortunately, although the protocol is relatively straightforward, the specification is nearly 200 pages! It is thus not an easy sell for customers or analysts (and sometimes implementers).

What does it mean to be a user of a Web services transaction? Initially it may seem like a good idea to let business logic directly affect the flow of a transaction from within a transaction service, but in practice it doesn't really work. It blurs the distinction between what you would expect from a transaction protocol (guarantees of consistency, isolation etc.) which are essentially non-functional aspects of a business "transaction", with the functional aspects (reserve my flight, book me a taxi, etc.)

In BTP, because business logic is encoded within the transaction protocol, it essentially means that a user has to be closely tied to (or perhaps even be) the coordinator. Business information, such as the ability for a participant to remain *prepared* (for example, hold onto a hotel room) for a specific period of time is propagated from the participant to the coordinator, but there is nothing within the protocol to allow this information to filter up to the application/client where it really belongs.

In order to use cohesions it is also necessary for Web services to expose back-end implementation choices about participants. In order to parameterize the two-phase completion protocol, the terminator of the cohesion obviously needs to be able to say "prepare A and B and cancel C and D," where A, B, C and D are participants that have been enrolled in the cohesion by services (such as a flight reservation system). In a traditional transaction system, users don't see the participants (imagine if you had to explicitly tell all of your XA resource managers to prepare and commit?) Naturally this is something that programmers don't feel comfortable with and it goes against the Web services orthodoxy.

Also, because BTP requires transaction control to use the *open-top* approach, it is difficult to

leverage existing enterprise transaction implementations. Few transaction systems (or their administrators) will feel comfortable exposing their coordinators through the two-phase interface.

Furthermore, the BTP specification expends great efforts to ensure that two-phase completion does not imply ACID semantics. This is good in so far that traditional ACID transactions are not suitable for all types of Web services interactions. However, everything is left up to back-end implementation choices and there is nothing in the protocol (implicit or explicit) to allow a user to determine what choices have been made. Therefore, it is impossible to reason about the ultimate correctness of a distributed application. For example, if you wanted to use BTP for ACID transactions, then of course services could use traditional XA resource managers (for example) wrapper by BTP participants. Unfortunately, there is no way within the BTP for those services to inform external users that this is what they have done so that they can safely be used within the scope of a BTP "ACID" transaction.

As you have seen, each model in WS-Tx clearly defines the semantics within the protocol (Atomic Transaction is ACID, for example). The models in WS-Tx are each aimed at a specific problem domain and is not intended to be used as a global panacea. Unfortunately, BTP does not have such well-differentiated models: the cohesion model is essentially a superset of the atom model.

Therefore, BTP has only one model that must be used to solve a variety of different problems. It does this by relaxing restrictions such as atomicity, durability etc. within the protocol (cohesion or atom) and allowing services to define those semantics outside of the model. Although this may appear at first glance to give developers increased flexibility, this has the disadvantage of not allowing them to be able to reason about an application's overall functionality and behavior. It also makes it difficult to construct applications from arbitrary services since within the protocol, you cannot determine a priori how a service will behave – extra information about the semantics and behavior of the service would have to be available in some implementation/vendor-specific manner.

Pros and cons of WS-Tx

Both the WS-C and WS-Tx specifications are smaller than BTP, at about 45 pages in total. Simplicity and interoperability with existing transaction infrastructures played a key role in their development. As we mentioned at the start of this paper, the WS-C and WS-Tx specifications have not yet been submitted to a standardization body, so errors and omissions are inevitable. However, these issues will all be resolved with subsequent revisions; there are no fundamental flaws in either specification.

On the plus side, the separation of coordination from transactions is good; coordination is a more fundamental requirement and a separate framework offers the chance for a cleaner separation of concerns. Because WS-C does not imply transactionality or a specific protocol implementation, it can therefore be used in more places than other protocols that have use of coordination but are tied to transactions (such as BTP).

The fact that WS-Tx Atomic Transactions are meant specifically for closely-coupled interactions with ACID semantics makes integration with back-end infrastructures easier. Web services are for interoperability as much as for the Internet. As such, interoperability of existing transaction processing systems will be an important part of Web services transactions. Such systems already form the backbone of enterprise-level applications and will continue to do so for the Web services equivalent. Business-to-business activities will involve back-end transaction processing systems either directly or indirectly and being able to tie together these environments will be the key to the successful take-up of Web services transactions.

In addition, because the semantics of Atomic Transactions and their participants are well-defined, it takes away any ambiguity from users and services: they know a priori what semantics to expect because it is a requirement from the protocol. Because BTP essentially only has one type of transaction (atoms are a subset of cohesions), it must allow flexible implementations of participants for long-duration interactions and therefore BTP does not define strict semantics for any participant. It is up to the service/participant implementer to somehow make this information available to users outside the scope of the transaction protocol.

The WS-Tx Business Activity gives service developers freedom to define compensation mechanisms that best suit their services (for example, using Atomic Transactions where necessary), while at the same time providing a simple model for the users of these services. In addition, it ties in well with Web services choreography techniques such as BPEL4WS.

Importantly as you've already seen, because WS-Tx leverages WS-C, it is intended as a portfolio of extended transaction models, each suited for a specific problem domain. Therefore, as use cases appear that cannot be addressed by one of the protocols already within WS-T, new protocols may be added.

The table shows a summary of the various differences and similarities between WS-C/T and BTP.

Comparative analysis

	WS-C/Tx	**BTP**
Coordination framework	WS-C	None – tied to two-phase.
Transaction framework	None, but current defined protocols cover typical patterns (AT and BA); others to be added later.	General protocol, no patterns. Statically defined.
Strict atomic model	**Atomic Transaction** - Atomic Transaction requires strict ACID properties. Specifically for interoperability with traditional transaction systems.	**Atom** - Atom is atomic only, other properties specified by service (and not available via protocol). Uses open-top protocol; makes interoperability with existing transaction systems difficult.
Relaxed model	Business Activity allows flexible participant list.	Cohesions allow flexible participant list. Requires participants to be exposed to application/terminator.
Scopes	Yes. Business Activity manages relationship between scopes. Nested scopes allowed.	No. Cohesion manages relationship within scope.
Flexible outcomes for consensus groups.	Yes, via Business Activity.	Yes, via cohesion.
Flexible participation in consensus groups.	Yes. Participants can exit in Business Activity protocol.	Yes, participants can resign from a cohesion. (Resignation from an atom is equivalent to read-only in Atomic Transaction.)
Service behavior	Defined by the protocols.	Services define behavior (not specified by BTP).
Business logic/coordinator separation	Distinct.	Mixed (open-top protocol requires strong coupling between business logic and coordinator).
Web services-specific	Yes.	No. Requires a lot of extra effort from the specification/protocol.
Failure recovery	Optimized protocol.	Re-drive protocol.

Conclusions

A few years ago the world of Web services and transactions looked like requiring new techniques to

address the problems that it presented, and BTP was seen as the solution to those problems. Unfortunately, with the benefit of hindsight it did not address what users really want: the ability to use existing enterprise infrastructures and applications and for Web services transactions to operate as the glue between different corporate domains.

Although the BTP model has some similarities with WS-Tx, the two specifications differ in some critical areas. For example, transaction interoperability: most enterprise transaction systems do not expose their coordinators through the two-phase protocol. In addition, BTP has many subtle (and some not-so-subtle) impacts on implementations, both at the transaction level and, more importantly, at the user/service level.

Much has been made of the fact that ACID transactions aren't suitable for loosely-coupled environments like the Web. However, very little attention has been paid to the fact that these loosely-coupled environments tend to have large strongly-coupled corporate infrastructures behind them. Any Web services transactions specification should not ask "what can replace ACID transactions?", but rather "how can we leverage what already exists?"

Resources

- Look for X/Open CAE Specification – Distributed Transaction Processing: The XA Specification, X/Open Document Number XO/CAE/91/300 (ISBN 1-872630-24-3).
- Find the Web Services Security specification on *developerWorks* (April, 2002).
- Read over the Web Services Addressing specification on *developerWorks* (March, 2003).
- Get the BTP Committee specification on OASIS (April, 2002).
- Find the Web Services Coordination Specification on *developerWorks* (September, 2003).
- Read over the Web Services Transactions Specification on *developerWorks* (August, 2003).
- Get more information in "The Business Transactions Protocol: Transactions for a New Age" (*Web Services Journal*, November 2002).
- Find the Travel Agent Scenario discussed in this article in The Business Transactions Protocol Primer from OASIS (June, 2002).
- Read this interesting discussion in the Clemens Vasters weblog.
- Read the Business Process Execution Language for Web Services, version 1.1 specification on OASIS (May, 2003).
- Find the Web Services Policy Framework specification on *developerWorks* (May, 2003).

About the authors

Mark Little is Chief Architect, Transactions for Arjuna Technologies Ltd. He has worked in the area of transaction processing for nearly twenty years and has helped develop several specifications in the area of Web services transactions.

Tom Freund is a Senior Technical Staff Member at IBM who has worked extensively in the area of transaction processing. He is currently active promoting transaction processing concepts in areas of emerging technology.