Chapter {14}

Transaction Management

The notion of a transaction is fundamental to business systems architectures. A transaction, simply put, ensures that only agreed-upon, consistent, and acceptable state changes are made to a system—regardless of system failure or concurrent access to the system's resources.

With the advent of Web service architectures, distributed applications (macroservices) are being built by assembling existing, smaller services (microservices). The microservices are usually built with no a priori knowledge of how they may be combined. The resulting complex architectures introduce new challenges to existing transaction models. Several new standards are being proposed that specify how application servers and transaction managers implement new transaction models that accommodate the introduced complexities.

The first sections of this chapter introduce the fundamental concepts behind transactions and explain how transactions are managed within the current Java/J2EE platforms. Later sections discuss the challenges of using existing transaction models for Web services, explain newly proposed models and standards for Web service transactions, and finally, detail proposed implementations of these new models on the Java platform.

> Concepts

A transaction may be thought of as an interaction with the system, resulting in a change to the system state. While the interaction is in the process of changing system state, any number of events can interrupt the interaction, leaving the state change incomplete and the system state in an inconsistent, undesirable form. Any change to system state within a transaction boundary, therefore, has to ensure that the change leaves the system in a stable and consistent state.

A transactional unit of work is one in which the following four fundamental transactional properties are satisfied: atomicity, consistency, isolation, and durability (ACID). We will examine each property in detail.

Atomicity

It is common to refer to a transaction as a "unit of work." In describing a transaction as a unit of work, we are describing one fundamental property of a transaction: that the activities within it must be considered indivisible—that is, *atomic.*

A Flute Bank customer may interact with Flute's ATM and transfer money from a checking account to a savings account. Within the Flute Bank software system, a transfer transaction involves two actions: debit of the checking account and credit to the savings account. For the transfer transaction to be successful, both actions must complete successfully. If either one fails, the transaction fails. The atomic property of transactions dictates that all individual actions that constitute a transaction must succeed for the transaction to succeed, and, conversely, that if any individual action fails, the transaction as a whole must fail.

Consistency

A database or other persistent store usually defines referential and entity integrity rules to ensure that data in the store is *consistent.* A transaction that changes the data must ensure that the data remains in a consistent state—that data integrity rules are not violated, regardless of whether the transaction succeeded or failed. The data in the store may not be consistent during the duration of the transaction, but the inconsistency is invisible to other transactions, and consistency must be restored when the transaction completes.

Isolation

When multiple transactions are in progress, one transaction may want to read the same data another transaction has changed but not committed. Until the transaction commits, the changes it has made should be treated as transient state, because the transaction could roll back the change. If other transactions read intermediate or transient states caused by a transaction in progress, additional application logic must be executed to handle the effects of some transactions having read potentially erroneous data. The *isolation* property of transactions dictates how concurrent transactions that act on the same subset of data behave. That is, the isolation property determines the degree to which effects of multiple transactions, acting on the same subset of application state, are isolated from each other.

At the lowest level of isolation, a transaction may read data that is in the process of being changed by another transaction but that has not yet been commit-

ted. If the first transaction is rolled back, the transaction that read the data would have read a value that was not committed. This level of isolation—*read uncommitted*, or "dirty read"—can cause erroneous results but ensures the highest concurrency.

An isolation of *read committed* ensures that a transaction can read only data that has been committed. This level of isolation is more restrictive (and consequently provides less concurrency) than a read uncommitted isolation level and helps avoid the problem associated with the latter level of isolation.

An isolation level of *repeatable read* signifies that a transaction that read a piece of data is guaranteed that the data will not be changed by another transaction until the transaction completes. The name "repeatable read" for this level of isolation comes from the fact that a transaction with this isolation level can read the same data repeatedly and be guaranteed to see the same value.

The most restrictive form of isolation is *serializable.* This level of isolation combines the properties of repeatable-read and read-committed isolation levels, effectively ensuring that transactions that act on the same piece of data are serialized and will not execute concurrently.

Durability

The *durability* property of transactions refers to the fact that the effect of a transaction must endure beyond the life of a transaction and application. That is, state changes made within a transactional boundary must be persisted onto permanent storage media, such as disks, databases, or file systems. If the application fails after the transaction has committed, the system should guarantee that the effects of the transaction will be visible when the application restarts. Transactional resources are also recoverable: should the persisted data be destroyed, recovery procedures can be executed to recover the data to a point in time (provided the necessary administrative tasks were properly executed). Any change committed by one transaction must be durable until another valid transaction changes the data. ▷

Transaction Manager

In a simple Java application that interacts with a database management system (DBMS), the application can demarcate transaction boundaries using explicit SQL commits and rollbacks. A more sophisticated application environment, with multiple transactional resources distributed across a network, requires a dedicated component to manage the complexity of coordinating transactions to completion.

A *transaction manager* works with applications and application servers to provide services to control the scope and duration of transactions. A transaction manager also helps coordinate the completion of global transactions across multiple transactional resource managers (e.g., database management systems), pro-

▷ **Isolation Levels and Locking**

Traditionally, transaction isolation levels are achieved by taking locks on the data that they access until the transaction completes. There are two primary modes for taking locks: optimistic and pessimistic. These two modes are necessitated by the fact that when a transaction accesses data, its intention to change (or not change) the data may not be readily apparent.

Some systems take a *pessimistic* approach and lock the data so that other transactions may read but not update the data accessed by the first transaction until the first transaction completes. Pessimistic locking guarantees that the first transaction can always apply a change to the data it first accessed.

In an *optimistic* locking mode, the first transaction accesses data but does not take a lock on it. A second transaction may change the data while the first transaction is in progress. If the first transaction later decides to change the data it accessed, it has to detect the fact that the data is now changed and inform the initiator of the fact. In optimistic locking, therefore, the fact that a transaction accessed data first does not guarantee that it can, at a later stage, update it.

At the most fundamental level, locks can be classified into (in increasingly restrictive order) shared, update, and exclusive locks. A *shared* lock signifies that another transaction can take an update or another shared lock on the same piece of data. Shared locks are used when data is read (usually in pessimistic locking mode).

An *update* lock ensures that another transaction can take only a shared lock on the same data. Update locks are held by transactions that intend to change data (not just read it).

If a transaction locks a piece of data with an *exclusive* lock, no other transaction may take a lock on the data. For example, a transaction with an isolation level of read uncommitted does not result in any locks on the data read by the transaction, and a transaction with repeatable read isolation can take only a share lock on data it has read.

Locking to achieve transaction isolation may not be practical for all transactional environments; however, it remains the most common mechanism to achieve transaction isolation.

vides support for transaction synchronization and recovery, and may provide the ability to communicate with other transaction manager instances.

A transaction context contains information about a transaction. Conceptually, a transaction context is a data structure that contains a unique transaction identifier, a timeout value, and the reference to the transaction manager that controls the transaction scope. In Java applications, a transaction manager associates a transaction context with the currently executing thread. Multiple threads may be associated with the same transaction context–dividing a transaction's work into parallel tasks, if possible. The context also has to be passed from one transaction manager to another if a transaction spans multiple transaction managers (see Figure 14.4).

Two separate but interconnected Java specifications pertain to the operation and implementation of Java transaction managers. These are detailed in the next sections. ▷

Two-Phase Commit and Global Transactions

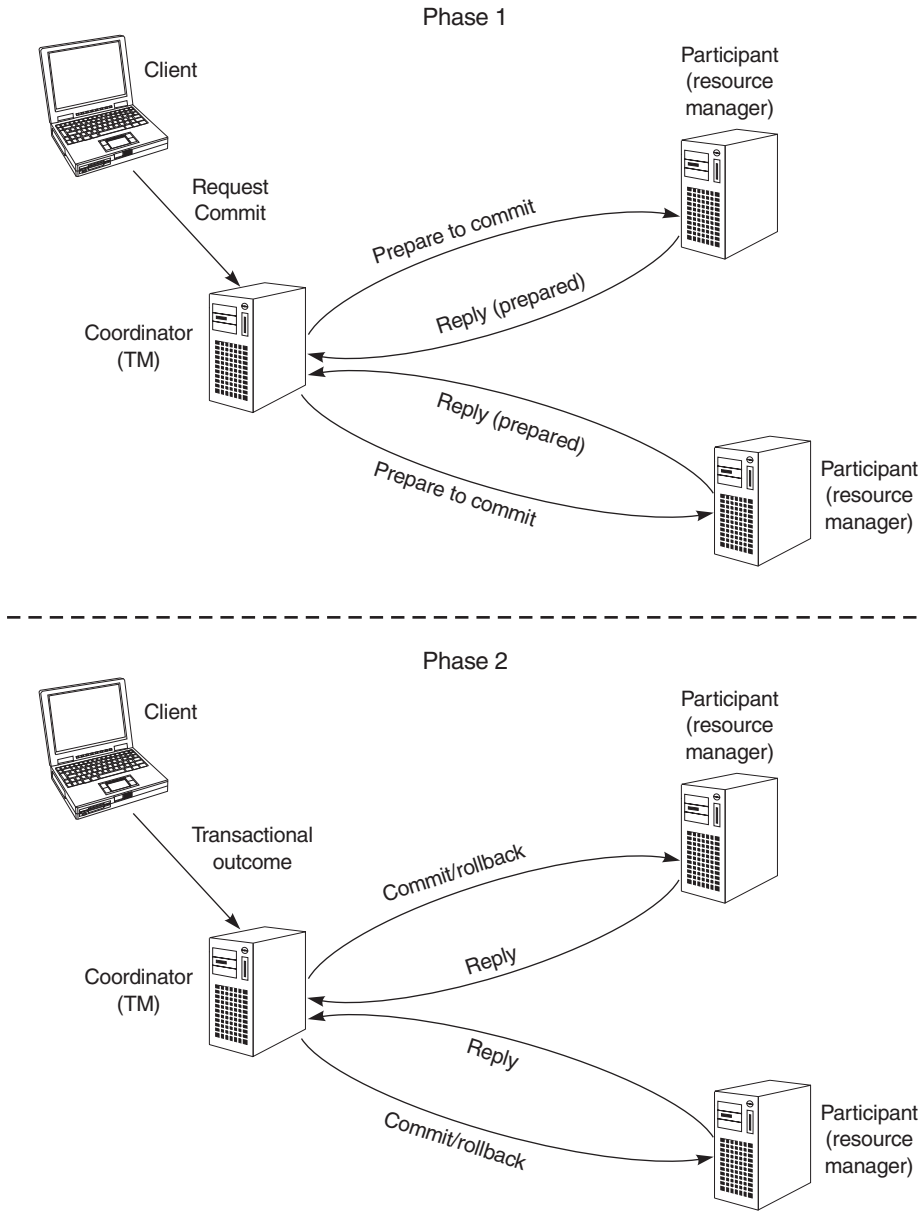
Global transactions span multiple resource managers. To coordinate global transactions, the coordinating transaction manager and all participating resource managers should implement a multiphased completion protocol, such as the *two-phasecommit (2PC) protocol* (Figure 14.1). Although there are several proprietary implementations of the this protocol, X/Open XA is the industry standard. Two distinct phases ensure that either all the participants commit or all of them roll back changes.

During the first, or *prepare phase,* the global coordinator inquires if all participants are prepared to commit changes. If the participants respond in the affirmative (if they feel that the work can be committed), the transaction progresses to the second, or *commit phase,* in which all participants are asked to commit changes.

▷ **Transaction Manager versus TP Monitor**

Transaction processing (TP) monitors, such as CICS and IMS/DC, enhance the underlying operating system's scalability and its ability to manage large transaction volumes, by taking on some of the roles of the underlying operating system. For example, a TP monitor, in addition to managing transactions, also performs connection pooling and task/thread pooling and scheduling. Transaction management is only one function of a TP monitor. In today's J2EE environment, application servers perform a similar function and may be thought of as modern equivalents of TP monitors.

Figure 14.1 Two phase commit and Global transactions
The two-phase commit protocol



The two-phase commit protocol ensures that either all participants commit changes or none of them does. A simplified explanation follows of how this happens in a typical transaction manager. To keep the discussion brief, we examine only a few failure scenarios. Once a transaction starts, it is said to be *in-flight*. If a

machine or communication failure occurs when the transaction is in-flight, the transaction will be rolled back eventually.

In the prepare phase, participants log their responses (preparedness to commit) to the coordinator, and the state of the transaction for each participant is marked *in-doubt*. At the end of the prepare phase, the transaction is in-doubt. If a participant cannot communicate with the global coordinator after it is in the in-doubt state, it will wait for resynchronization with the coordinator. If resynchronization cannot take place within a predefined time, the participant may make a heuristic decision either to roll back or commit that unit of work. (Heuristic or arbitrary decisions taken by the participant are a rare occurrence. We emphasize this because it is a conceptual difference between current transaction models and those such as BTP, discussed later in the chapter).

During the second phase, the coordinator asks all participants to commit changes. The participants log the request and begin commit processing. If a failure occurs during commit processing at one of the participants, the commit is retried when the participant restarts.

Transaction Models

Transactions managers can provide transactional support for applications using different implementation models. The most common is the flat transaction model. A transaction manager that follows the flat transaction model does not allow transactions to be nested within other transactions. The flat transaction model can be illustrated by examining the transaction model employed in J2EE application servers today.

As an example, Flute Bank provides a bill payment service. The service is implemented by an EJB that interacts with two other EJBs the account management EJB (to update account balance) and the check writing EJB (to write out a check to the payee).

Table 14.1a illustrates the scope of transactions in the scenario where all three EJBs are deployed with declarative transaction attribute of `Required`.

Table 14.1b illustrates the scope of transactions where the bill payment service EJB and account management EJB are deployed with transaction attribute `Required` but the check writing EJB is deployed with a transaction policy of `RequiresNew`. In this scenario, when the check writing EJB method is executed, the container suspends T1 and starts a new transaction, T2. Check writing occurs within the scope of the second transaction. When control returns to the bill payment EJB, T2 is committed, and the previous transaction is activated. So, in a flat transaction model, two transactions are executed in different scopes–T2's scope is not within T1's scope.

Table 14.1a All EJBs Deployed with Transaction Attribute `Required`

Component	`BillPayment` EJB	`AccountMgmt` EJB	`CheckWrite` EJB	`BillPayment` EJB
Transaction	T1	T1	T1	T1 (terminates)

Table 14.1b Account Management EJB with Transaction Attribute `RequiresNew`

Component	`BillPayment` EJB	`AccountMgmt` EJB	`CheckWrite` EJB	`BillPayment` EJB
Transaction	T1	T1	T2 (started, then terminated)	(T1 resumed, then terminated)

In effect, one business activity (bill payment) is executed under the scope of two separate transactions. In a flat transaction model, the only way to correctly control the scope of transactions that span multiple services is to reach an agreement beforehand on how these services will be combined for a business activity and to apply appropriate transaction policies for the services by agreement.

What if the check writing service were created and hosted by a different company? What if the check writing company did not want its deployment attributes to be dictated by Flute Bank? What if other customers (Flute Bank's competitor) of the check writing company wanted a contradictory transaction policy? Would it not be in Flute Bank's interest to have its bill payment transaction control the outcome of the business activity, regardless of whether the check writing company decided to start a new transaction or not?

A *nested transaction model,* shown in Figure 14.2, is one solution to the above problem. It allows transactions to consist of other transactions: a top-level transaction may contain subtransactions. In a *nested transaction model,* with respect to the above example, the bill payment service would start a top-level transaction (*t*). Both the account management service and the check writing service are free to start new transactions (*ts1* and *ts2*). But both these subtransactions are within the scope of the top-level transaction (*t*). Transactions in a nested transaction model also adhere to the ACID properties of transactions–that is, *t* completes successfully only if *ts1* and *ts2* both complete successfully. If either subtransaction fails, the top-level transaction would fail, thereby guaranteeing atomicity.

A nested transaction model allows services to be built independently and later combined into applications. Each service can determine the scope of its transaction boundaries. The application or service that orchestrates the combination of services controls the scope of the top-level transaction.

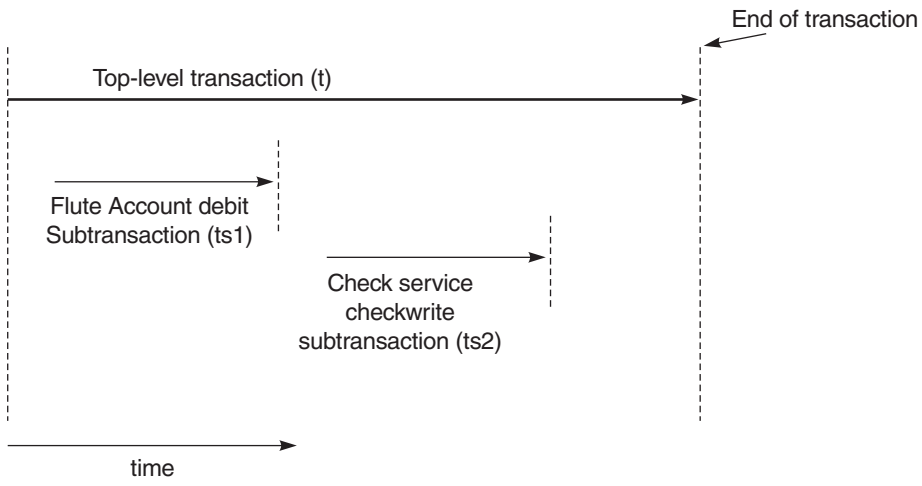


Figure 14.2 Nested transactions

Java Transaction API (JTA)

In a J2EE environment, the transaction manager has to communicate with the application server, the application program, and the resource managers, using a well-defined and standard API. The Java Transaction API (JTA) is defined precisely for this purpose. It does not specify how the transaction manager itself has to be implemented but how components external to the transaction manager communicate with it. The JTA defines a set of high-level interfaces that describe the contract between the transaction manager and three application components that interact with it: the application program, resource manager, and application server. These are described below. The next section describes the Java Transaction Service (JTS) specification—a related specification that deals with the implementation details of transaction managers.

- **JTA contract between transaction managers and application programs.** In J2EE applications, a client application or server EJB component whose transaction policy is managed in the bean code (`TX_BEAN_MANAGED`) can explicitly demarcate (i.e., start and stop) transactions. For this purpose, the application gets a reference to a user transaction object that implements the `javax.transaction.UserTransaction` interface. This interface defines methods, among others, that allow the application to commit, roll back, or suspend a transaction. The transaction that is started or stopped is associated with the calling user thread.

- In a Java client application, the `UserTransaction` object is obtained by looking up the application server's JNDI-based registry. There is no standard JNDI name for storing the `UserTransaction` reference, so the client application must know beforehand (usually by reading a configuration file) how to obtain the `UserTransaction` object from the application server's JNDI registry. In an EJB, the `UserTransaction` object is exposed through the `EJBContext`.
- **JTA contract between transaction managers and application servers.** J2EE containers are required to support container-managed transactions. The container demarcates transactions and manages thread and resource pooling. This requires the application server to work closely with the transaction manager. For example, in an EJB where the container has to manage transaction demarcation, the EJB container has to communicate with the transaction manager to start, commit, and suspend transactions. The J2EE application server and transaction manager communicate via the `javax.transaction.TransactionManager` interface.
- **JTA contract between transaction managers and transactional resource managers.** Resource managers provide an application access to resources, which can be databases, JMS queues, or any other transactional resource. An example of a global transaction is the one that changes a Flute Bank employee's address: it involves updating the employee master database (Oracle) and posting a message to a JMS queue (MQSeries) for the external payroll company (because payroll taxes can change based on employee address). For the transaction manager to coordinate and synchronize the transaction with different resource managers (in this example, Oracle and MQSeries), it must use a standards-based, well-known transaction protocol that both resource managers understand (such as X/Open XA).
- JTA defines the `javax.transaction.xa.XAResource` interface, which allows a transactional resource manager, such as a DBMS or JMS implementation, to participate in a global transaction. The XAResource interface is a Java mapping of the industry-standard X/Open XA. J2EE applications communicate to the resource managers via a resource adapter (e.g., JDBC connection), and JDBC 2.0 extensions support distributed transactions. JDBC 2.0 provides two interfaces that support JTA-compliant resources: the `javax.sql.XAConnection` and `javax.sql.XADataSource`.
- Similarly, JMS providers implement the `javax.jms.XAConnection` and the `javax.jms.XASession` interfaces in supporting JTA transaction managers.

Java Transaction Service

The JTA specification's main purpose is to define how a client application, the application server, and the resource managers communicate with the transaction manager. Because JTA provides interfaces that map to X/Open standards, a JTA-compliant transaction manager can control and coordinate a transaction that spans multiple resource managers (*distributed* transactions). JTA does not specify how a transaction manager is to be implemented, nor does it address how multiple transaction managers communicate with each other to participate in the same transaction. That is, it does not specify how a transaction context can be propagated from one transaction manager to another. The Java Transaction Service (JTS) specification addresses these concepts.

JTS specifies the implementation contracts for Java transaction managers. It is the Java mapping of the CORBA Object Transaction Service (OTS) 1.1 specification. The OTS specification defines a standard mechanism for generating and propagating a transaction context between transaction managers, using the IIOP protocol. JTS uses the OTS interfaces (primarily `org.omg.CosTransactions` and `org.omg.CosTSPortability`) for interoperability and portability. Because JTS is based on OTS, it is not unusual to find implementations that support transactions propagating from non-Java CORBA clients as well.

Within a JTS-compliant transaction manager implementation, a communication resource manager component handles incoming and outgoing transaction requests. Details of the interaction between the application server and a JTS transaction manager can be found in the JTS specification document.

In addition to being a Java mapping of OTS, the JTS specification also mandates that a JTS-compliant transaction manager implement all JTA interfaces–that is, a JTS-based transaction manager interfaces with an application program, the application server, and resource managers, using the JTA. Figure 14.3 shows the relationship between the application components involved in a transaction.

> A Transaction Model for Web Services

So far, we have examined the properties, transactions, and specifications for transaction managers in the current Java/J2EE environment. Most, if not all, of today's Java transaction managers implement the flat transaction model, in compliance with JTA and/or JTS. Also, most transactions today are executed within

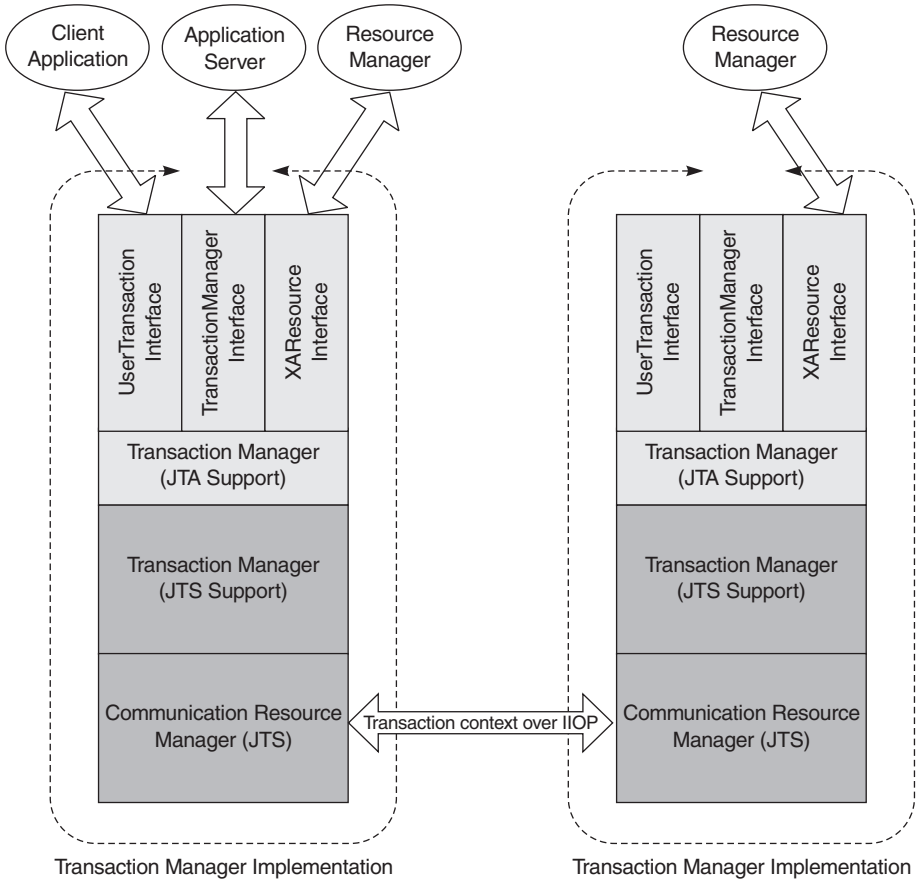


Figure 14.3 JTS transaction manager components

the scope of one enterprise, within one trust domain, and with all resources under the control of one transaction manager. A *business transaction* may be defined as a consistent state change in the business relationship among *two or more parties,* with each party maintaining an independent application system (or Web service) that maintains the state of each application. Business transactions form a good number of transaction types in Web services.

While the flat transaction model is well suited for traditional business computing environments, a Web services environment can create new, interesting challenges for flat transactions:

- **Loose coupling among services.** A Web service application is a loose coupling of constituent Web services and is often constructed by combining Web services from multiple organizations. In this scenario, each service may implement a standalone business process and therefore demarcate transaction

boundaries within the service. Because transactions are started and committed within each service, it is not possible to use a simple flat transaction model within the combined application.

For example, in Flute Bank's bill payment service, Flute works with an external check writing service to mail payment checks on behalf of its customers. That is, when a Flute customer transacts with Flute's bill payment service, the service has to accomplish two separate functions as a part of one atomic transaction: it has to debit the customer's checking account, and it has to transact with the external check-writing service and send details of the payee (payee address, payment amount, etc.) If the external check-writing service demarcates its own transaction, a flat transaction model may be inadequate to guarantee atomicity to the Flute consumer's action. (The consumer's interaction is internally implemented as two distinct transactions).

It should be said that a nested transaction model will accommodate this situation, but it is not the only model that can do so. A flat transaction model that supports interposition also can accommodate this situation; we look at interposition in the later sections that explain the Business Transaction Protocol (BTP).

- **Long-running transactions.** Many business transactions are long-running—some may run for hours or days. As transaction isolation is achieved by holding locks on resources, long-running transactions may reduce transaction concurrency to unacceptable levels. Moreover, in a business transaction, the actions of one party affect how long another's resources are locked, opening the door for denial-of-service attacks. It is clear that the isolation property (which controls locking) must be relaxed in such transactions.

  Because a business transaction can take a long time to complete, businesses often impose a time limit on certain types of transactions. For example, an airline may hold a tentative reservation for a travel agency for two days. When the time limit is reached, the airline may decide to either cancel or confirm the reservation automatically. The time period for timeout and the result (confirm or cancel) should be a business agreement between the two parties. The transaction protocol should anticipate such events and allow for negotiation between the participants.

- **Optional subtransactions.** In some business transactions, a subset of activities may be optional. An example of a fictitious travel agency will better illustrate such a situation.

  Assume that the travel service will search with multiple airline services to find flights, based on a consumer's request. Further assume that the airlines all take a pessimistic approach to their commitment: when the travel agent ser-

vice requests seats on a particular flight, if enough seats are available, the airlines lock the seats for the travel agent and marks the seats unavailable to other requests. Figure 14.4 depicts the participants in this example.

In such a scenario, a transaction initiated by a consumer request will result in the agency requesting seats from one or more airline systems (in the example airlines A, B, and C), a rental car agency, and a hotel. Some airline systems may decline to participate in the transaction (perhaps because no seats are available, as is the case with airline C), and more than one may lock seats for the request (airlines A and B both have seats and confirm them).

Airline C's failure to commit seats cannot be considered a failure of the reservation transaction as a whole, because the application can determine the *atomic subset* of participants, based on business logic. As a result, the atomic property of the transaction need not be satisfied. Moreover, should multiple airlines lock seats (Airlines A and B), a compensating transaction that releases previously booked seats will have to be generated for Airline B, so that only one airline eventually reserves the required seats. This, in effect, conceptually relaxes the durability property of the reservation transaction initiated by the travel service. For this solution to work, application logic has to be written to

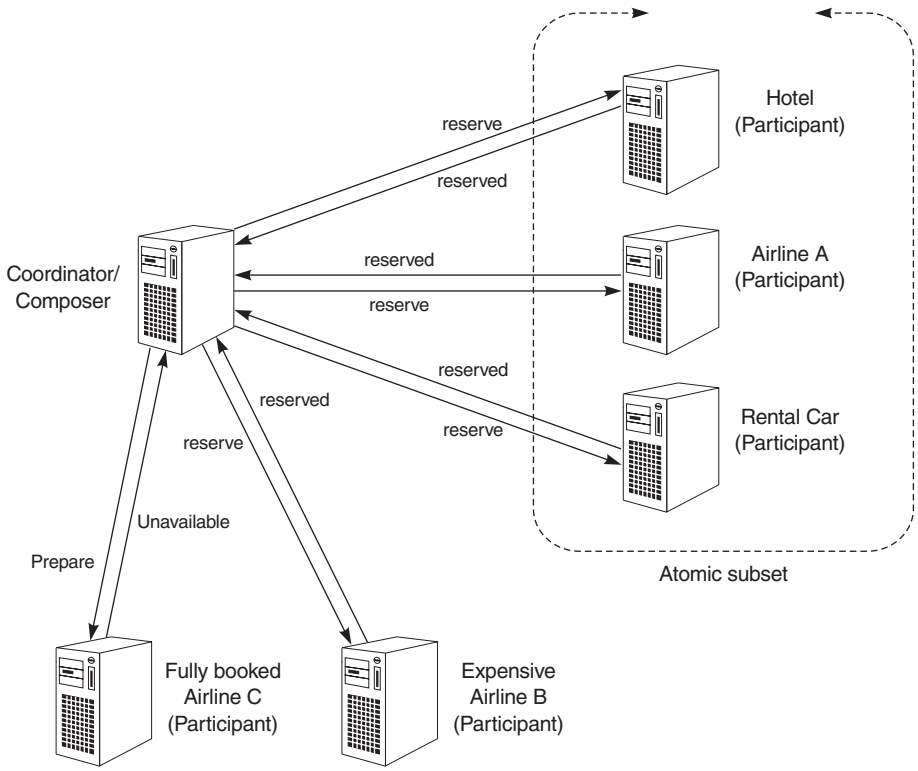


Figure 14.4 Atomicity relaxed, travel agency example

select the subset of activities that must be considered atomic and also to generate compensation transactions.

- **Transaction context propagation.** A Web service application that combines Web services from different organizations will have transactions that span the services. The constituent services may be implemented using different technologies on different platforms (e.g., UNIX, Windows) and may store state in heterogeneous databases. When a transaction spans these disparate services, the transaction context must be passed from service to service. In a JTS environment, JTS specifies how a compliant transaction manager propagates the context from one transaction manager to another over IIOP. A Web service transaction manager must accommodate XML-based transaction contexts propagated over Web service protocols.

This section has illustrated the need for a new model for transactions and transaction management. In the new model, a transaction is viewed as a series of business activities with the context of the transaction made available to each activity. The execution of activities is coordinated, but individual activities may decide to commit the effects of the transaction and make the effects visible before the larger transaction is completed. However, the flat transaction model is adequate for a large number of Web service applications–those that feature short transactions and those in which individual services persist data into a central database or databases within the same enterprise. The new model therefore has to support both the properties of ACID transactions and the relaxed ACID properties demanded by loosely coupled, long-running business transactions. ▷

Table 14.2 summarizes the differences between traditional transaction models and the new model.

> New Transaction Specifications

In this section, we discuss three important proposed standards that implement new transaction models: Business Transaction Protocol (BTP) from OASIS, WS-Transaction from IBM and Microsoft, and Activity Service from OMG. Their purpose is to coordinate Web services into applications that provide reliable out-

▷ Strictly speaking, adoption of this new model is not dictated primarily by Web service architectures but by the fundamental long-running nature of business transactions. Web service architecture has simply accentuated the need for it.

Table 14.2 Properties of Traditional and Business Transaction Models

Property	Traditional transactions	Business transactions
Atomicity	Required; all or nothing.	Depends; sometimes desirable, sometimes may be applicable only to a subset of functions.
Consistency	Required.	Required; temporary inconsistencies rectified.
Isolation	Required; state change is not visible until transaction completes.	Relaxed; each service controls degree of visibility.
Durability	Required; effects persist.	Required, but based on atomicity property; some parts may be volatile.
Context propagation	Relaxed; may not be required.	Required.

comes. Of the three, BTP is the most mature and sophisticated. We examine it first and in the most detail.

Business Transaction Protocol (BTP)

In May 2002, the OASIS Business Transaction Technical Committee (BTTC) published a specification, BTP 1.0, for coordinating transactions between applications controlled by multiple autonomous parties. BTP 1.0 is the work of several companies (BEA, IBM, Sun, HP, Choreology, ORACLE, and others) that addresses the challenges posed by business transactions to traditional transaction models.

Examining BTP messages and interactions in detail is beyond the scope of this book. Our intention is to describe the concepts and motivations behind BTP. The full BTP specification can be found at *www.oasis-open.org/committees/business-transactions*.

BTP recognizes that in a business transaction, no single party controls all resources needed. In such an environment, parties manage their own resources but coordinate in a defined manner to accomplish the work scoped by a transaction. Individual service providers either agree to join a transaction or not. If they agree, they are required to provide a mechanism to confirm or cancel their commit-

ments to the transaction. They may, however, autonomously decide when to unlock resources they hold and/or whether to use compensating transactions to roll back transient states that were persisted.

The BTP specification did not arise out of the need created by Web services architecture but was formed to address the needs of interorganizational transactions and of workflow systems. However, the authors realized early on the limitations of similar coordination protocols tied to communication protocols.

BTP defines an abstract message set and a binding to communication protocols. Its ability to coordinate transactions across multiple autonomous services and its use of XML messages makes it particularly suited for adoption in Web service architectures. BTP is designed such that the protocol may be bound to any carrier protocol, and BTP implementations bound to the same carrier protocols should be interoperable. The current specification describes a SOAP 1.1/HTTP binding.

Transaction and security aspects of an application system are often related, but the BTP specification consciously does not address how a BTP transaction will integrate with a security system, because Web services security standards are still evolving (independently of the transaction specifications).

Application and BTP Elements

BTP is a *protocol*—a set of well-defined messages exchanged between the application systems involved in a business transaction. Each system that participates in a business transaction can be thought of as having two elements—an *application element* and a *BTP element* (Figure 14.5).

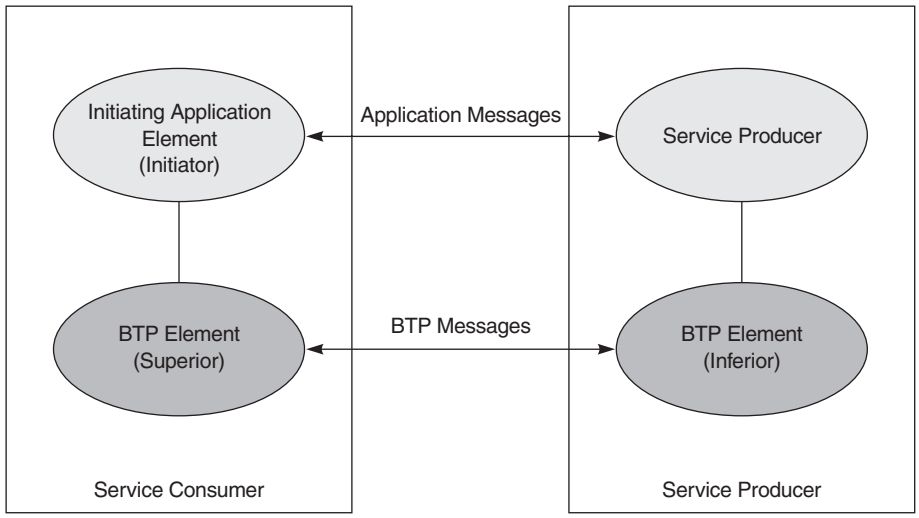


Figure 14.5 Application and BTP elements overview

The application elements exchange messages to accomplish the business function. When Flute Bank's bill payment service sends a message to the check writing service with details of the payee's name, address, and payment amount, the application elements of the two services are exchanging a message.

The BTP elements of the two services also exchange messages that help compose, control, and coordinate a reliable outcome for the message sent between the application elements.

The application element pertains to the service consumer and service producer components the application programmer deploys–that is, application/business logic. The BTP elements are supplied by the BTP vendor. The separation of system components into BTP and application elements is a logical one. These elements may or may not coexist in a single address space.

With respect to a BTP transaction, application elements play the role of *initiator* (the Web service that starts the transaction) and *terminator* (the Web service that decides to commit or end the transaction). The initiator and terminator of a transaction are usually played by the same application element.

BTP elements play either a *superior* or *inferior* role. The BTP element associated with the application element that starts a business transaction is usually assigned the *superior* role. The superior informs the inferior when to prepare to terminate the transaction and waits for the inferior to report back on the result of its request. The following sections detail roles played by BTP elements and the nature and content of BTP messages.

Types of BTP Transactions

Table 14.2 summarized the transactional properties business transactions must satisfy. In all types of business transactions, the isolation property is relaxed. Some business transactions require the entire transaction to be treated as an atomic operation, while another class of business transactions requires that the atomic property also be relaxed. BTP accommodates both types of transaction needs. BTP Atomic Business Transactions, or *atoms,* are like traditional transactions, with a relaxed isolation property. ▷

▷ In traditional transactions, a transaction manger will cancel (roll back) a transaction if any resource manager participating in the transaction cannot commit or cannot prepare. In BTP, this is not *always* the case; the set of participants that must confirm before a transaction can be committed is called a *confirm-set.* The confirm-set may be the set of all participants or a subset of participants.

BTP Cohesive Business Transactions, or *cohesions,* are transactions where both isolation and atomicity properties are relaxed.

Atoms *Atoms* are business transactions where all participants have to agree before a transaction can be committed that is, all participants in an atom are guaranteed to see the same ending to the transaction. If any participant cannot confirm, the entire transaction is canceled. Because BTP transactions do not require strict isolation, it is up to each participating service to determine how to implement transaction isolation.

Figure 14.6 depicts a Web service consumer invoking two business methods on two different services, within the scope of a transaction. If the overall transaction is an atom, the BTP element (superior) at the service consumer end is called an *atom coordinator* or simply a *coordinator.* The BTP element plays the coordinator role and coordinates a BTP atomic transaction. It does this by exchanging BTP messages with the BTP elements associated with the two service producers when the application asks it to complete the transaction.

As Figure 14.6 also shows, inferior BTP elements are called *participants.* The participant is in charge of persisting the state change made by the associated application element (service producer), which it does by following instructions (via BTP messages) from the superior (coordinator). If either participant informs the superior that it cannot confirm the transaction, the transaction is rolled back–that is, the confirm-set in this example is "participant a" and "participant b." ▷

Cohesions *Cohesions* are transactions where not all involved parties must agree to commit their changes before a transaction is committed. Only some subset of the parties may need to agree. The subset of parties that need to agree before a transaction can be completed is determined using business logic and is called the confirm-set.

In the example illustrated by Figure 14.4, airline C refuses to reserve seats on the requested flight, because it is fully booked. This refusal to participate in the transaction does not mean that the transaction should be rolled back, because the application logic is able to determine that two other airlines have reserved seats on their flights. Furthermore, because two airlines have reserved seats, the application element must instruct the BTP superior to cancel the more expensive reservation. The BTP superior that composes the transaction based on instructions from the initiating application element is called a *cohesion composer.* The

▷ An atom is a BTP transaction whose confirm-set is the set of *all* inferiors—that is, in an atom, any inferior has power to veto the transaction.

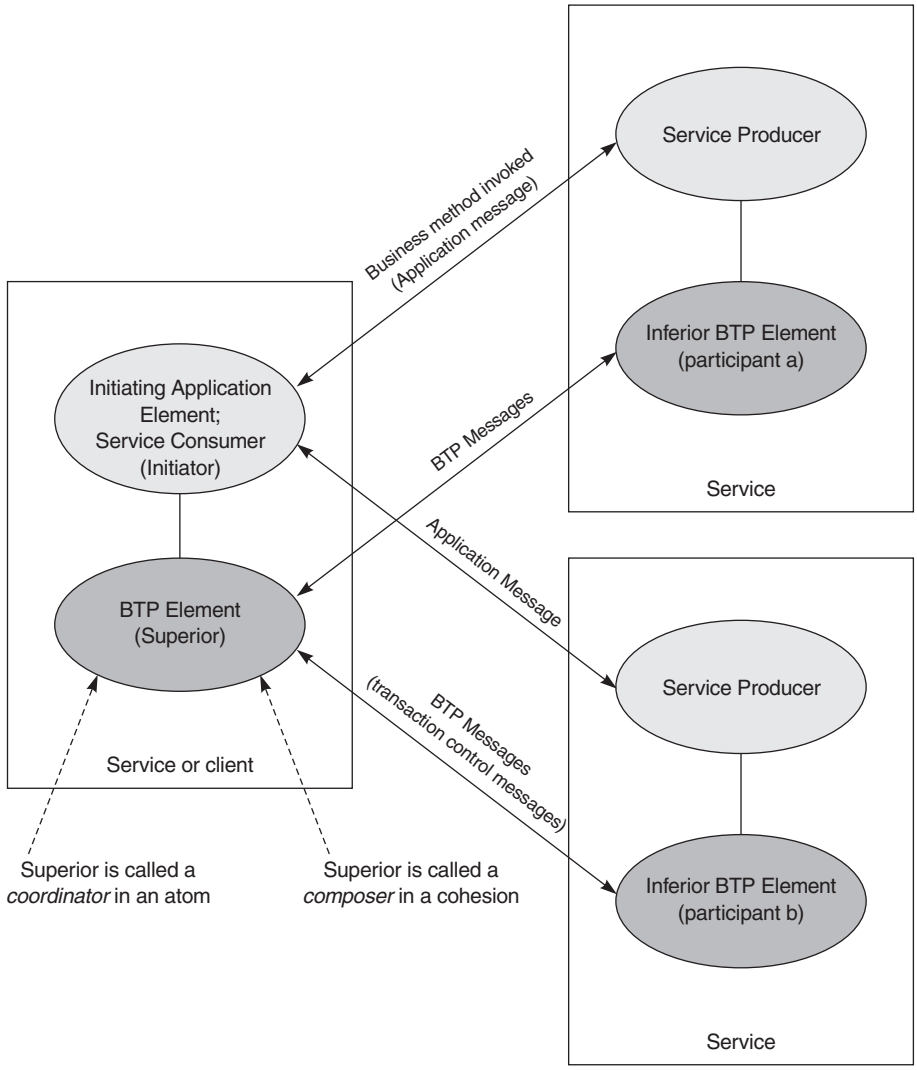


Figure 14.6 Application and BTP elements in an atom/cohesion

confirm-set in the example is the BTP elements associated with the Web services of Airline A, the rental car agency, and the hotel.

Referring to Figure 14.6, in a *cohesion* scenario, the BTP element (superior) at the service consumer end is called a *cohesion composer* or simply a *composer.* The BTP element associated with the service producer is called a *participant.* In a cohesion scenario, the business logic in application element (initiating element) can determine whether the transaction can be completed–that is, whether "participant a" only need confirm, "participant b" only need confirm, or both must confirm. If only one participant must confirm but both eventually confirm, the composer will ask the unwanted participant to cancel.

In BTP, the actions of transaction coordinator or composer can be influenced by application elements (i.e., business logic). In a cohesion, the initiating application element determines which subset of activities is to be included in the overall transaction by providing that information to the superior. Application elements can also influence the control and coordination of the transaction by providing the superior with additional context information (via *qualifiers;* see next section), such as transaction time limits and other application-specific values. ▷

BTP Transactions and Locking

For both atoms and cohesions, the isolation level for cohesions is left up to each service. Isolation can be achieved by:

- Making changes but applying locks, as in traditional transactions
- Deferring changes until a transaction commits (perhaps by writing a log of changes that will be applied later)
- Making changes and making the interim results visible to others

If the third option is chosen, the effect of making interim changes visible is called the *provisional effect.* If a service makes visible provisional changes and the transaction is ultimately rolled back, new transactions (compensations) may have to be generated to undo the changes made (This is known as the *counter effect*).

BTP Actors, Roles, and Messages

As mentioned earlier, the BTP element associated with the initiator plays the superior role. The initiator is also usually the *terminator* of the initiated transaction. Depending on the type of business transaction, superiors are either coordinators or composers of the business transaction–an atom is coordinated by an *atom coordinator (coordinator),* and a cohesion is composed by a *cohesive composer (composer).* All other BTP elements are inferiors to this superior. In the simplest case, with only one superior and one inferior (i.e., only two parties are involved in the business transaction), the inferior is called a participant.

▷ A cohesion is a transaction whose confirm-set is a subset of all inferiors participating in the transaction—that is, only inferiors in the confirm-set have the power to veto the transaction. This subset is determined by application logic, and the application element passes the confirm-set to the composer before asking the composer to complete the transaction.

Figure 14.7 shows a more detailed version of Figure 14.5. The application *(initiator)* first asks a BTP element called the *factory* to create the coordinator/composer. The *factory* creates the superior and returns the transaction *context*. The *initiator* then invokes the business method on the service consumer and passes the context to the service.

How the context is passed depends on the protocol binding. It is, for example, stuffed as a header block in a SOAP message. At the other end, the invoked service asks a BTP element called *enroller* to enroll in the transaction, passing the received context. The enroller creates the inferior (*participant*) and enrolls in the transaction with the *superior.* Finally, the service provides the response to the business method and passes along the *context reply.* The message sequence diagram in Figure 14.8 details the exchange of messages between the different elements.

BTP messages must be bound to a protocol such as SOAP. Because we have not yet described the BTP binding to SOAP, the following section shows only abstract forms of BTP messages.

All BTP messages have an associated schema. The `CONTEXT` message shown below is an example of a BTP message.

```
<btp:context id>
<btp:superior-address> address</btp:superior-address>
<btp:superior-identifier> URI </btp:superior-identifier>
```

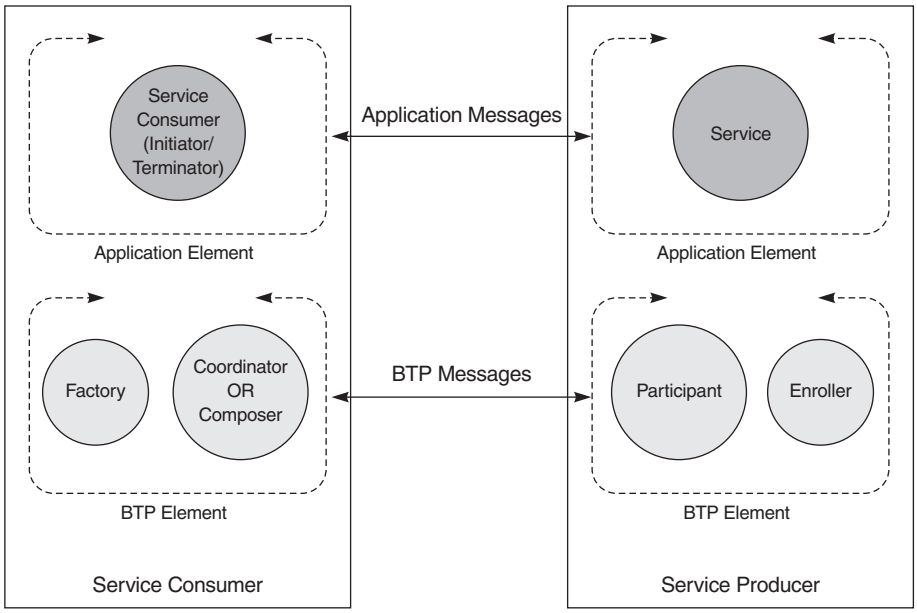


Figure 14.7 BTP and application elements

```
<btp:superior-type>atom</btp:superior-type>
<btp:qualifiers> qualifiers </btp:qualifiers>
<btp:reply-address> address </btp:reply-address>
</btp:context>
```

The `superior-address` element contains the address to which `ENROLL` and other messages from an inferior are to be sent. Every BTP address element (superior-address, reply-address, etc.) has the following XML format:

```
<btp:superior-address>
    <btp:binding-name> </btp:binding-name>
    <btp:binding-address></btp:binding-address>
    <btp:additional-information>information ... </btp:additional-
    information>
</btp:superior-address>
```

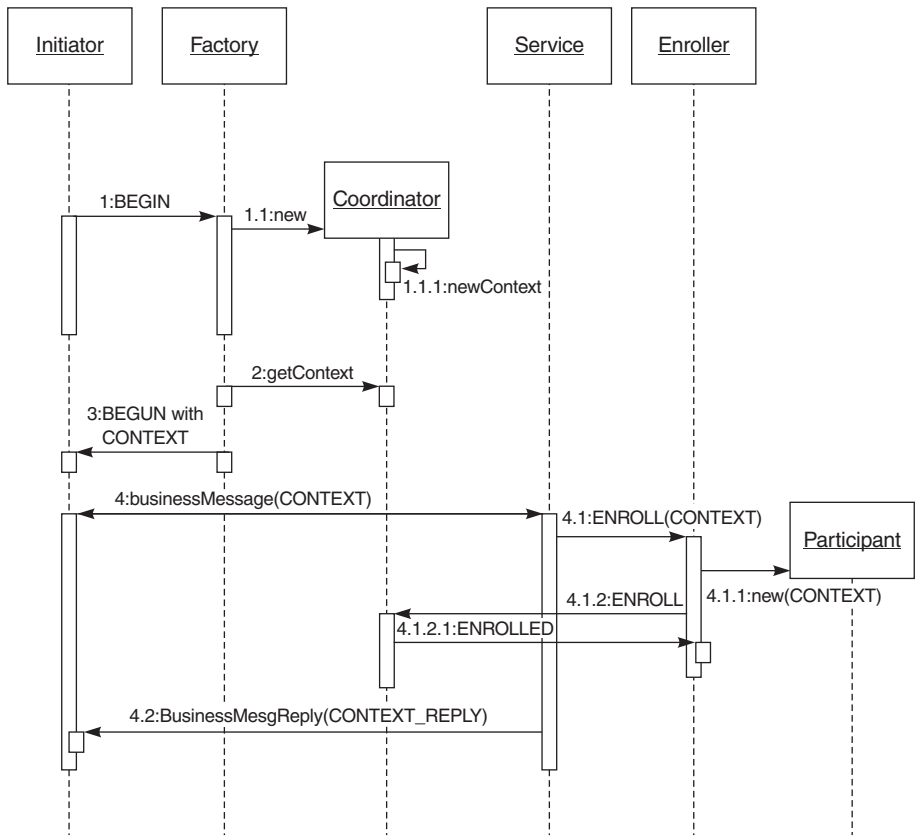


Figure 14.8 BTP actors and messages overview

`superior-identifier` contains a unique identifier (URI) for the superior. `superior-type` indicates whether the context is for a transaction that is an atom or a cohesion. The `qualifiers` element provides a means for application elements to have some control over transaction management. Qualifiers are data structures whose contents can influence how the transaction coordinator/composer controls the transaction. BTP defines a few standard qualifiers (such as transaction time limit), but BTP vendors can define more such parameters.

The `reply-address` element contains the address to which a `CONTEXT_REPLY` message is to be sent (this is required only if the BTP message is not sent over a request-response transport).

BTP Two-Phase Protocol

Once the initiating application decides to terminate the transaction, it asks the BTP superior to confirm the transaction. The BTP elements then follow a two-phase protocol to complete the transaction. This is quite similar to the two-phase protocol used in the flat transaction model, but there are some key differences, discussed later in this section. Figure 14.8 illustrated how a transaction is started. Figure 14.9 shows how such a transaction is terminated using the two-phase protocol.

On receiving a `PREPARE` message, an inferior (in Figure 14.9, the participant) can reply with a `PREPARED`, `CANCEL`, or `RESIGN` message. In Figure 14.9, because only one inferior exists, the participant must reply with a `PREPARED` message if the

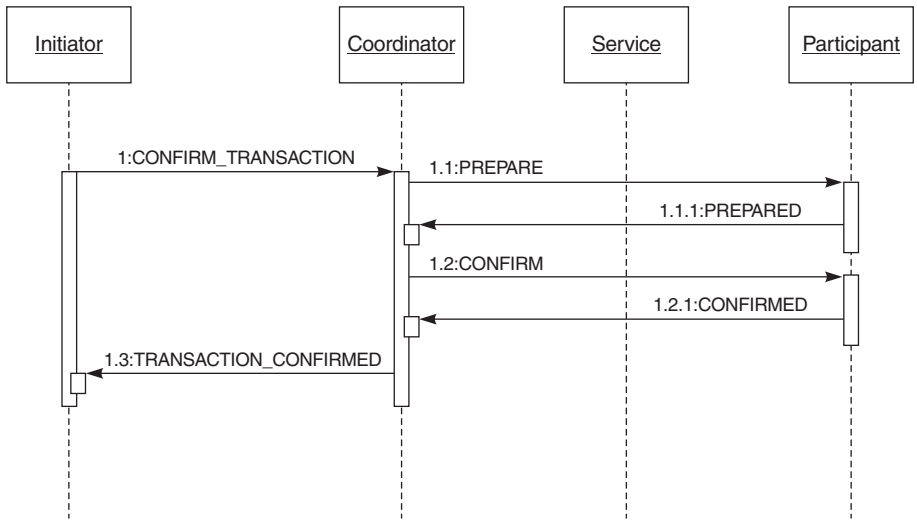


Figure 14.9 A simple atom example illustrating the BTP two-phase protocol

transaction is to be confirmed and progress to phase 2 (CONFIRM). An example of the BTP message for PREPARE is shown below:

```
<btp:prepare id>
    <btp:inferior-identifier> URI </btp:inferior-identifier>
    <btp:qualifiers>qualifiers</btp:qualifiers>
    <btp:target-additional-information>
        additional address information
    </btp:target-additional-information>
    <btp:sender-address>address</btp:sender-address>
</btp:prepare>
```

As explained previously, the qualifiers element contains a set of standard or application-specific qualifiers. The timeout for inferiors is one of the qualifiers that should be sent for a PREPARE message. target-address points to the address of the inferior that was ENROLLed. The PREPARE message will be sent to that address. The sender-address points to address of the superior.

The effect on the outcome of a final transaction of having multiple inferiors depends on whether the transaction is a cohesion or is an atom. The set of inferiors that must eventually return CONFIRMED to a CONFIRM message for the transaction to be committed is called a *confirm-set.* For an atomic transaction, the set consist of *all* of a superior's inferiors. For a cohesion, the confirm-set is a subset of all its inferiors. The subset is decided by the application element associated with the superior (this implies that business logic is involved).

Figure 14.10 illustrates how a composer with multiple participants confirms a cohesion with the two-phase protocol. The application element (the *initiator* and the *terminator* of the transaction) decides that only participants 1 and 2 should confirm—that the *confirm-set* consists of participants 1 and 2. To accomplish this,

1. The terminator sends a CONFIRM_TRANSACTION with the IDs of the participants in the *confirm-set.*
2. The *decider* (composer) sends PREPARE messages to participants 1 and 2 and a CANCEL message to participant 3.
3. As soon as PREPARED messages return from participants in the confirm-set, the decider sends out CONFIRM (phase 2) messages.
4. When the confirm-set replies with CONFIRMED messages, the transaction is confirmed.

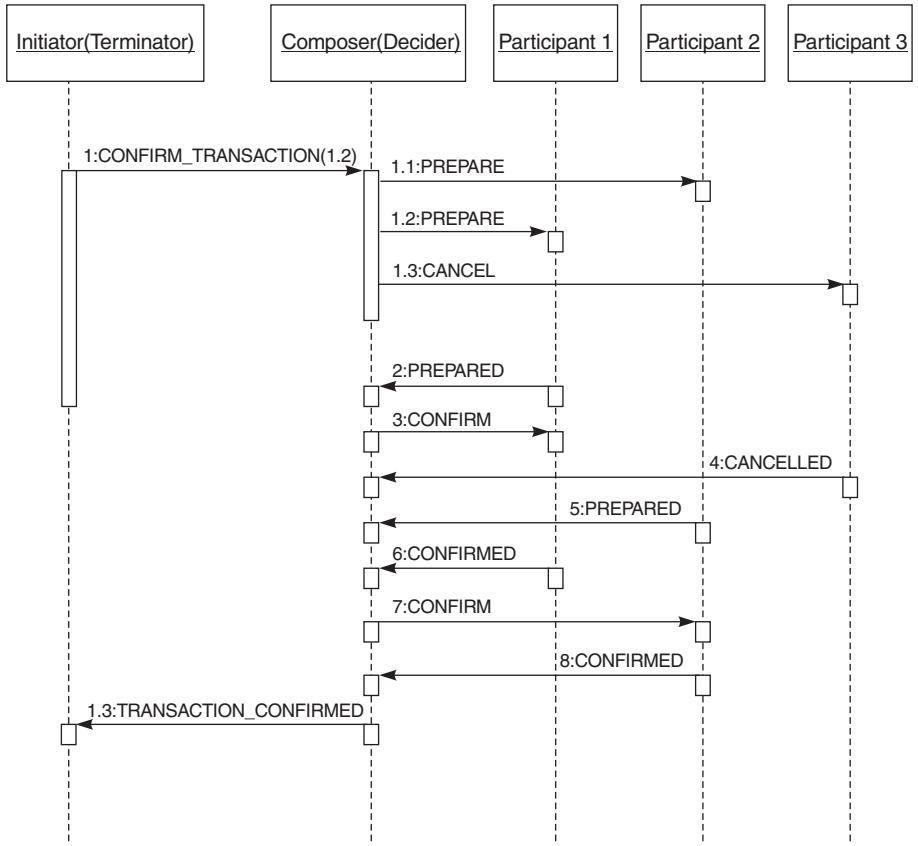


Figure 14.10 Cohesion completion

How the confirm subset is passed to the decider is better understood by examining the `CONFIRM_TRANSACTION` message structure:

```
<btp:confirm-transaction id>
    <btp:transaction-identifier> ... URI ... </btp:transaction-identifier>
    <btp:inferiors-list>
        <btp:inferior-identifier> inferior URI</btp:inferior-identifier>
        <btp:inferior-identifier> inferior URI</btp:inferior-identifier
    </btp:inferiors-list>
    <btp:report-hazard>true</btp:report-hazard>
    <btp:qualifiers>qualifiers</btp:qualifiers>
    <btp:target-additional-information>
        info
    </btp:target-additional-information>
    <btp:reply-address> decider address</btp:reply-address>
</btp: confirm_transaction>
```

Note that `inferiors-list` contains only the confirm-set of inferiors. If this element is absent, *all* inferiors are part of the confirm-set. For an atom, because all participants are in the confirm set, this element must not be present.

The `report-hazard` element defines when the decider informs the application that the transaction is conformed (`TRANSACTION_CONFIRMED` message):

- If `report-hazard` is `true`, the decider waits to hear from all inferiors, not just the confirm-set, before informing the terminator.
- If `report-hazard` is `false`, the decider must wait for all elements (even elements that receive a `CANCEL` message) to reply before communicating the outcome of the transaction to the terminator.

`report-hazard` is useful when the application element wants to know if there was a hazard (problem) with any inferior. ▷

Differences between the BTP and Current Two-Phase Commit Protocols

As discussed in the "Isolation Levels and Locking" sidebar earlier, BTP does not require a participant waiting for a confirm message to hold locks for maintaining application state. A prepared participant can choose to apply application state changes to the database and make this interim "provisional effect" visible. In doing so, it has to hold a compensating transaction ready to be applied (the counter effect), in case of the need to cancel.

Because Web service transactions can span networks prone to communication failure or congestion, a participant and its superior may not be able to communicate readily. BTP recognizes this and accommodates possible communication failures. A participant can limit the promise made in sending `PREPARED` back to its superior. by retaining the right to autonomously confirm or cancel application state changes it controls. This autonomous decision is very much like the heuristic decision made by in-doubt resources in a normal two-phase commit protocol. The difference is that whereas other transaction models consider heuristic decisions rare occurrences, BTP anticipates such problems and allows participants to indicate how long they are willing to be in a prepared state.

▷ If a coordinator or composer has only one inferior, it may decide to use a single-phase confirm operation and skip the two-phase protocol. Instead of a `PREPARE + CONFIRM` message exchange, it may send a `CONFIRM_ONE_PHASE` message to the inferior.

The two-phase protocol used in BTP ensures that either the entire transaction is canceled or that a consistent set of participants is confirmed.

What happens when a participant, having informed its superior that it is prepared to confirm, makes an autonomous decision to cancel, because it has not received the phase-two `CONFIRM` message within the allotted time? What if, after the autonomous decision is made to cancel, the superior sends out a `CONFIRM` message? In such cases, the superior will eventually recognize and log a contradiction, inform management/monitoring systems, and send a `CONTRADICTION` message to the participant. By sending a contradiction message, the superior acknowledges the contradiction and informs the inferior that it knows and has tried to cope with it.

The differences between BTP 2PC and current 2PC can be summed up as follows:

- In BTP, the application determines the confirm-set.
- BTP provides a means for applications to "tune" the two-phase protocol, by passing qualifiers to the coordinator. These qualifiers include standard BTP qualifiers, such as how long a participant is to wait before making an autonomous decision. Qualifiers can also be BTP vendor- or application-specific.
- BTP does not require two-phase locking. This can lead to contradictions if participants make autonomous decisions. Unlike other models, however, which treat heuristic decisions as rare occurrences, BTP anticipates such problems and tries to accommodate them.

Complex BTP Element Relationships

So far, we have examined the BTP elements, roles, and message in a rather simple scenario. A business transaction can span multiple organizations and multiple services, and BTP accommodates such complex scenarios by describing a tree structure for relationships between BTP elements. Figure 14.11 illustrates such a complex relationship. The BTP elements associated with services B and C are both superiors to other inferiors but are inferiors to the superior (composer A, also known as the *decider*). BTP element B is a composer of the transaction cohesion Tc1. As B is an inferior to A, B is called a *sub-composer.* Similarly, BTP element C coordinates the atom Ta1 and is a superior to F and G but an inferior to A , so C is a *sub-coordinator.*

A sub-composer or sub-coordinator is not a participant; it controls other participants and reports to its superior. On receiving `PREPARE` and `CONFIRM` messages from its superior, the sub-composer or sub-coordinator asks its confirm-set to

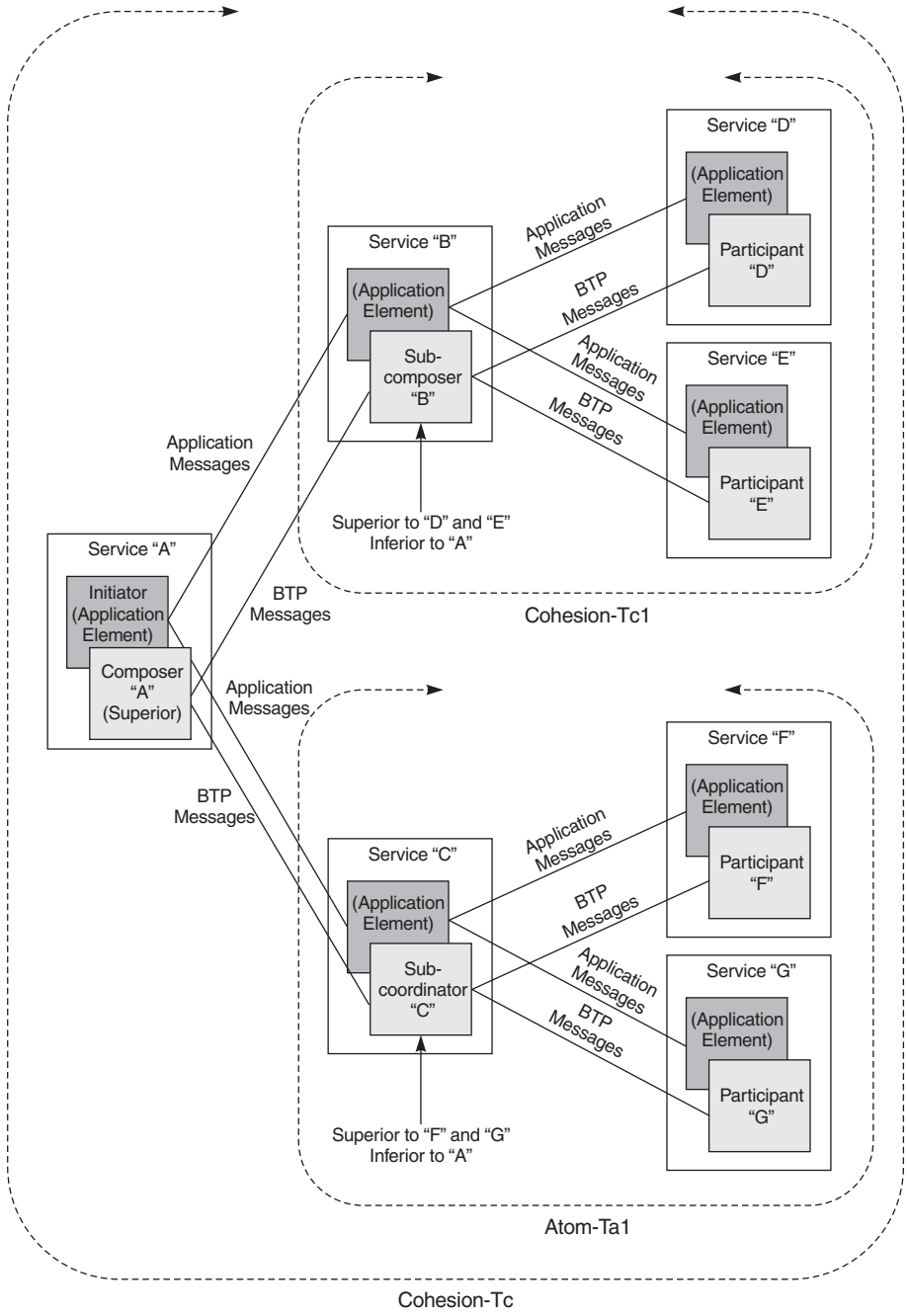


Figure 14.11
BTP element tree

PREPARE and CONFIRM and reports the result to its superior. In this way, BTP supports the creation of relationship trees of arbitrary depth and width. ▷

To summarize, if an *inferior* is a *superior* to other nodes, it may be a *sub-composer* or a *sub-coordinator*. A *sub-coordinator* treats its inferior's actions as atomic, while a *sub-composer* treats only a subset of its inferiors as atomic. A *participant* is a BTP inferior element that has no children; it is responsible for applying application state changes to a persistent store. Figure 14.12 shows the different roles played by BTP elements and the relationships between them.

BTP and SOAP Binding

BTP defines an abstract message set and a protocol that are independent of underlying communication protocols over which the messages are to be transmitted. For BTP to be viable for Web services, BTP messages must be transmitted over the de facto messaging standard for Web services: SOAP. BTP specifications define a binding for SOAP (and SOAP with Attachments).

As discussed above, most BTP messages are exchanged between BTP elements and are not mixed with any application-element business messages (e.g., PREPARE, CONFIRM). All such BTP messages are sent within the SOAP body under

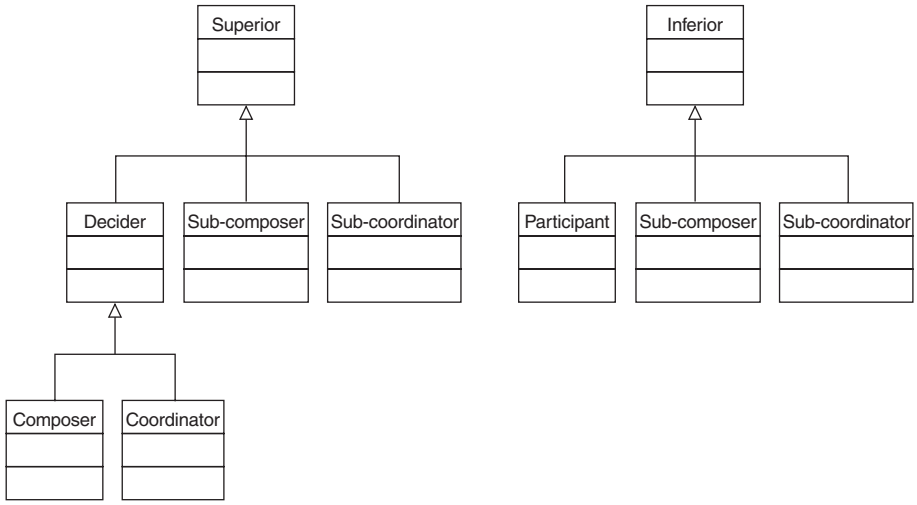


Figure 14.12 Roles played by superiors and inferiors

▷ Sub-composers and sub-coordinators are *interposed* (as in *injected* or *inserted*) into the transaction. By default, BTP supports an interposed transaction model, not a nested transaction model.

the btp:messages element. A few BTP messages are sent along with application messages.

One example of this type of message is CONTEXT. The BTP CONTEXT is propagated from a BTP superior to a BTP inferior when the initiating application element invokes a business method on a service. The BTP CONTEXT message, therefore, must be sent along with the application business message. For such messages, the BTP message is sent in the SOAP header under a single btp:messages element, and the application message is sent within the SOAP body. All BTP messages are sent without any encoding (application messages may be encoded).

The following example of a SOAP message starts a BTP atomic transaction in Flute Bank's bill payment service. The business method invoked is getLastPayment. As a new transaction is started, the BTP CONTEXT message is passed, along with the application message. The CONTEXT message is passed as a part of the SOAP header. The header is not encoded, but the body, which carries the application message, is RPC/encoded.

```
<?xml version="1.0" encoding="UTF-8"?>
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/"
    env:encodingStyle="">

    <env:Header>

       <btp:messages
            xmlns:btp="urn:oasis:names:tc:BTP:1.0:core">
            <btp:context>
                <btp:superior-address>
                    <btp:binding>soap-http-1</btp:binding>
                    <btp:binding-address>
                           http://www.flute.com/btpengine
                    </btp:binding-address>
                </btp:superior-address>

                <btp:superior-identifier>
                    http://www.flute.com/btp01
                </btp:superior-identifier>
```

```
                <btp:qualifiers>
                    <btpq:transaction-timelimit>
        xmlns:btpq="urn:oasis:names:tc:BTP:1.0:qualifiers">
                    <btpq:timelimit>500</btpq:timelimit>
                    </btpq:transaction-timelimit>
                </btp:qualifiers>
            </btp:context>
        </btp:messages>
    </env:Header>

    <env:Body>
        <ans1:getLastPayment
            xmlns:ans1="http://com.flute.webservice/billPay/wsdl/billPay">
            env:encodingStyle=" http://schemas.xmlsoap.org/soap/encoding/"

            <String_1 xsi:type="xsd:string">horizon_wireless </String_1>
        </ans1:getLastPayment>
    </env:Body>

</env:Envelope>
```

Although application architects and developers rarely concern themselves with messages exchanged between BTP elements, let us look at more examples, to help visualize the transaction coordination. The listing below shows the `BEGIN` BTP message over SOAP from the initiating application element to the associated BTP factory (Figure 14.13). This `BEGIN` message starts a cohesion.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<soap-env:Body>
     <btp:begin transaction-type="cohesion"
                                 xmlns:btp="urn:oasis:names:tc:BTP:xml"/>
 </soap-env:Body>
</soap-env:Envelope>
```

The following listing shows a SOAP message with a `CANCEL` request from a transaction coordinator to a participant.

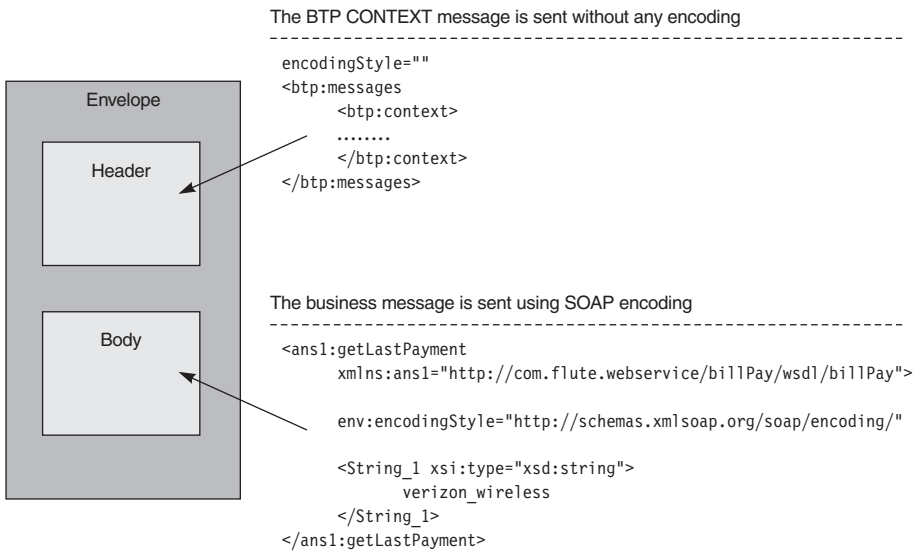


Figure 14.13 BTP SOAP message shown pictorially

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<soap-env:Body>
    <btp:cancel xmlns:btp="urn:oasis:names:tc:BTP:xml">
    <btp:inferior-identifier>
    http://www.flute.com/btp01/Fluteparticipant/-74c4a3c4:8dd-6ce8fec8:1
    </btp:inferior-identifier>
    <btp:superior-identifier>
    http://www.flute.com/btp01/TransactionManager/-77e2b3e4:8aa-3ce8fec8:9
    </btp:superior-identifier>
    </btp:cancel>
</soap-env:Body>
</soap-env:Envelope ▷
```

▷ **HP Web Services Transactions (HP-WST)**

HP's Web Services Transactions (WST) is the first implementation of a transaction manager based on the BTP specifications. The implementation, available at *www.hpmiddleware.com,* supports both cohesive and atomic transactions. HP-WST includes an HP-WST coordinator component, Java libraries for transaction participants, and libraries for Java clients.

WS-Transaction

WS-Transaction (WS-TX) and the related WS-Coordination specifications are relatively new (released in August 2002). They describe mechanisms to coordinate the executions of individual Web services into reliable applications. The two specifications, created jointly by Microsoft, IBM, and BEA, rely on or are built on existing Web services specifications of WSDL, SOAP, and WS-Security.

The execution of a Web service application is seen as a series of activities, each of which may be executed in a different Web service. WS-Coordination describes a general mechanism to create, register, and coordinate those activities across multiple Web services. When a participant Web service creates and registers the activity, it also identifies the coordination protocol–that is, the way collaboration occurs between activities. The specification does not describe protocols for coordination,. which can be achieved in many ways. Some coordination protocols may be transactional; some may not.

WS-Coordination provides a generalized framework that allows Web services to specify the coordination protocol. WS-Transaction describes protocols for two common transactional coordination patterns that may be used with the WS-Coordination framework: the atomic transaction coordination pattern and the business activity transaction pattern. Figure 14.14 shows the relationship between WS-Coordination and WS-Transaction.

Before describing the WS-Transaction coordination protocols, a brief explanation of the coordination framework described in WS-Coordination is necessary. The latter describes a standard coordination framework, consisting of an

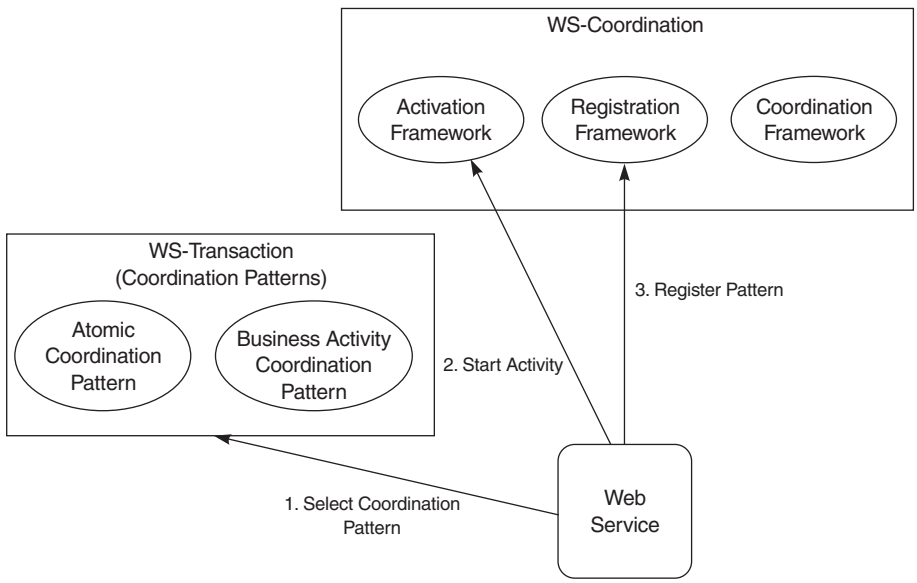


Figure 14.14 Relationship between WS-Coordination and WS-Transaction

activation service, which helps create a new activity; a *registration service,* to register an activity's participants; and a *coordination service,* to process an activity's completion.

An application contacts the activation service to create an activity, which is identified by a coordination context. The context is a container (defined by an XML schema) with elements for an ID, a context expiration time, the coordination type (the coordination protocols to be used), and other extensible elements. Web services that participate in the same activity receive application messages with the context attached. Web services then use the context to identify the registration service and register as participants to the original activity. The coordination service controls the completion of the activity, based on the selected coordination protocol.

A WS-TX atomic transaction coordination pattern preserves the ACID properties of transactions. The application requests that a new activity be created by the activity service. The activity service creates a coordination context that has the `coordinationType` element set to the URI `http://schemas.xmlsoap.org/ws/2002/08/wstx`. The coordination context allows for coordination-protocol-specific elements to be added to the context, using the extensibility mechanism (e.g., a context for an atomic transaction may also carry the isolation level value).

The Web service that created the transaction also registers with the registration service the coordination or completion pattern service for the completion protocol. The coordination service controls the completion of the activity, based on the registered completion protocol.

The BTP and WS-Transaction/WS-Coordination specifications provide for similar transactional needs. One important difference between them is that although BTP provides bindings for SOAP 1.1, it is independent of current de facto Web service standards, although WS-Transaction is built on or relies on SOAP, WSDL, and WS-Security.

Activity Service

Activity Service framework is a submission to OMG in response to OMG's call for additional structuring mechanisms for Object Transaction Service (OTS). OTS, you will recall, is the specification for the CORBA transaction manager. The Activity Service specification submission, like the BTP specification, addresses the transactional needs of complex, long-running transactions. The submission can be found at *http://cgi.omg.org/cgi-bin/doc?orbos/2000–06–19*.

This specification, unlike BTP, is directly tied to and based on the OMG Object Request Broker (ORB) and OTS specifications. The lack of support for XML-based messages and Activity Service's tie to OTS may mean that Web services implemented in compliance to it may not transact transparently with services writ-

ten on, say, the .NET platform. However, because most current Java transaction managers implement JTS (which is a Java mapping of OTS), Activity Service is likely to have a high impact on how Java transaction managers are implemented.

JSRs for Web Service Transaction Support

As described earlier, JTA- and JTS-based transaction managers are limited, in that they are meant to manage and coordinate distributed, ACID transactions. A new specification is needed to address the complex distributed transactions prevalent in Web service and workflow applications.

JSR-95, *J2EE Activity Service for Extended Transactions*, proposes a low-level API for J2EE containers and transaction managers. JSR-95 borrows from the OMG Activity Service submission and leverages concepts defined by JTA and EJB specifications.

JSR-156, *XML Transactions for Java (JAXTX),* also proposes support for extended transaction support in J2EE systems. Unlike JSR–95, JSR-156 proposes a closer relationship to the BTP specification, using XML messaging for transaction management and coordination. This proposed Java API also has a loftier long-term goal: to isolate application programmers and application servers from the underlying transaction manager implementation (BTP, WS-Transaction, and Activity Service). ▷

▷ The acronym "JAXTX," in an attempt at consistency with other JAX* APIs, is, in our opinion, misleading. The other JAX* APIs (e.g., JAXM) are meant to be used primarily by application programmers. The J2EE specifications have tried to minimize programmatic transaction demarcation, making the **UserTransaction** interface the only JTA interface exposed to the application.

We recognize that coordinated, non-atomic transactions (e.g., cohesive BTP transactions and business transactions in the WS-Transaction specification) will require application logic to generate compensation transactions. We also recognize that in determining the subset of activities treated as atomic, the interface exposed to application programmers must be expanded beyond the current JTA **UserTransaction** interface. But the vast majority of transaction coordination messages are meant to be exchanged by components below the application layer. Therefore, we feel that "JAXTX" is misleading.

Summary

J2EE servers are required only to implement the flat transaction model. This model is sufficient in situations where the transactional resources are centralized and under the control of one party. The ACID properties of such transactions can be maintained without compromise. This transactional model is adequate for Web service applications with short-running transactions and those in which transactions do not span organizational boundaries.

The flat transaction mode is inadequate to address the transactional needs of some business transactions in Web services, which frequently involve the assembly of disparate services. These disparate services may have been written with their own, autonomous transaction boundaries and may control their own resources. Web services for business-to-business integration can also be long-running, causing inefficiency as a result of the locking mechanism used for achieving isolation in traditional transaction models.

Relaxing the isolation property of transactions and allowing subtransactions to make visible state-change effects before the larger transaction completes solves the problem of inefficient resource locking in long-running transactions. Allowing a transaction to complete even if only a subset of subtransactions completes (i.e., relaxing the atomic property) allows a business transaction to be composed based on application needs.

The BTP, WS-TX, and Activity Service specifications address the needs of coordinating long-running business transactions that span autonomous, loosely coupled services while accommodating the needs of short-running, centralized transactions. BTP specifies the working of atomic and cohesive transactions, and WS-TX describes the atomic and business activity patterns for coordination. The Activity Service submission to OMG also provides a specification for business transactions but does not reference XML-based messages specifically. (Because Activity Service is based on OMG OTS, it does provide IDL specifications.)

Two JSRs, JSR-95 and JSR-156 propose an extended transaction model for the J2EE platform. As of now, no implementations of BTP or Activity Service transaction managers work seamlessly with J2EE servers. It will be some time before the relevant JSRs are fleshed out and we see servers with embedded, standards-based transaction support for extended transaction models. Until then, our current JTA- and JTS-based flat transaction managers will have to suffice.