Use Cases for the Business Transaction Protocol

OASIS Business Transactions Technical Committee
Models / Use Cases Technical Subcommittee
bt-models@lists.oasis-open.org

Introduction

This document presents different scenarios in which the Business Transaction Protocol (BTP) could be used. It attempts to capture the diversity of transactional requirements that may be imposed by applications.

Use Cases

Status of this Document

Revision 0.14 – Last updated 12/04/2001 23:28

1. Arranging a Night-Out

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

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

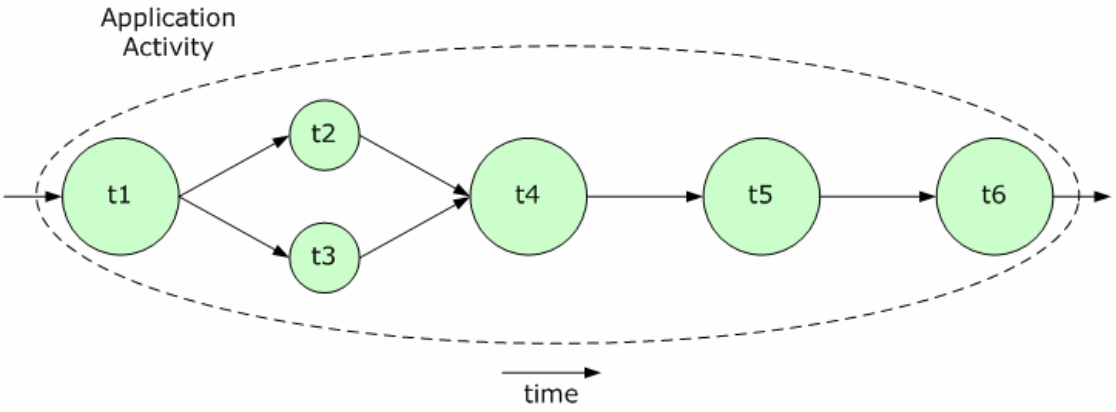


Figure 1: An example of a logical long-running "transaction", without failure.

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

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

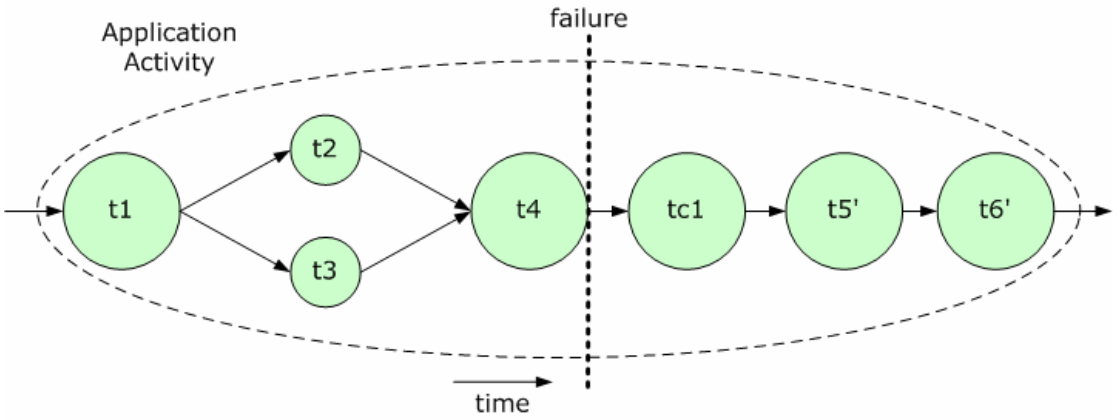


Figure 2: An example of a logical long-running "transaction", with failure.

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

2. Home entertainment system

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

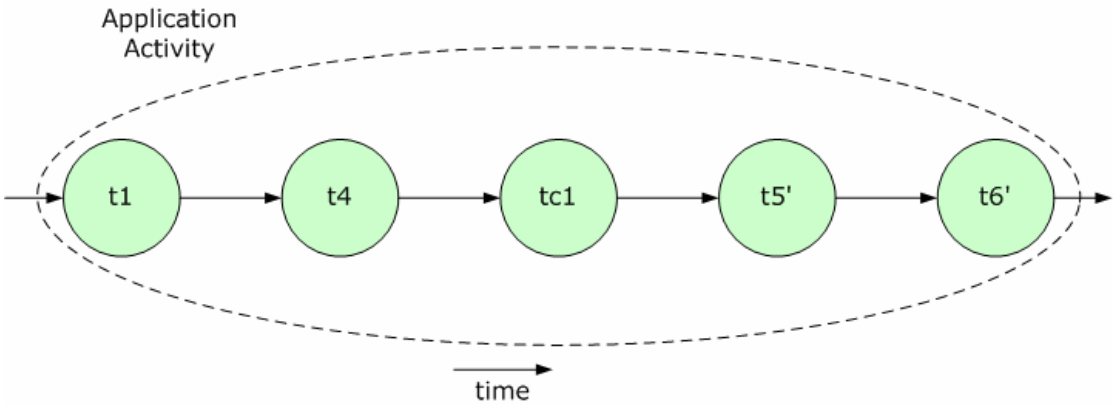


Figure 3: Building a home entertainment system via the Web.

When we visit the TV site we wish to start a transactional activity, t1, that will allow us to search and provisionally reserve (obtain transactional locks on) a number of different televisions (set A) that match our requirements. Before t1 terminates, we select a subset of the televisions, B, we are interested in, and provide a reference to our online bank account which the television site may contact to check that we have sufficient funds. t1

then ends, any locks obtained that are not members of B are released as normally for a transaction, (allowing other users to acquire them immediately if necessary), and all other locks are obtained and passed to t2.

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

3. Web services coordination

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

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

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

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

This notion of having a different extended transaction model for each use-case makes applications aware of the issues involved and helps to categorize objects and activities into which type of model they support. So, for example, one object may be used successfully within a two-phase and one-phase model, whereas another may only be used within two-phase. It is important to reduce complexity in developing "transactional" internet application by not over overloading a given model (e.g., ACID transactions).

4. Timed transactions

Many business transactions have specific "real-time" deadlines within which they must operate (e.g., purchasing of shares). After the deadline has elapsed, if the business transaction has not completed in a normal manner, there will typically be application

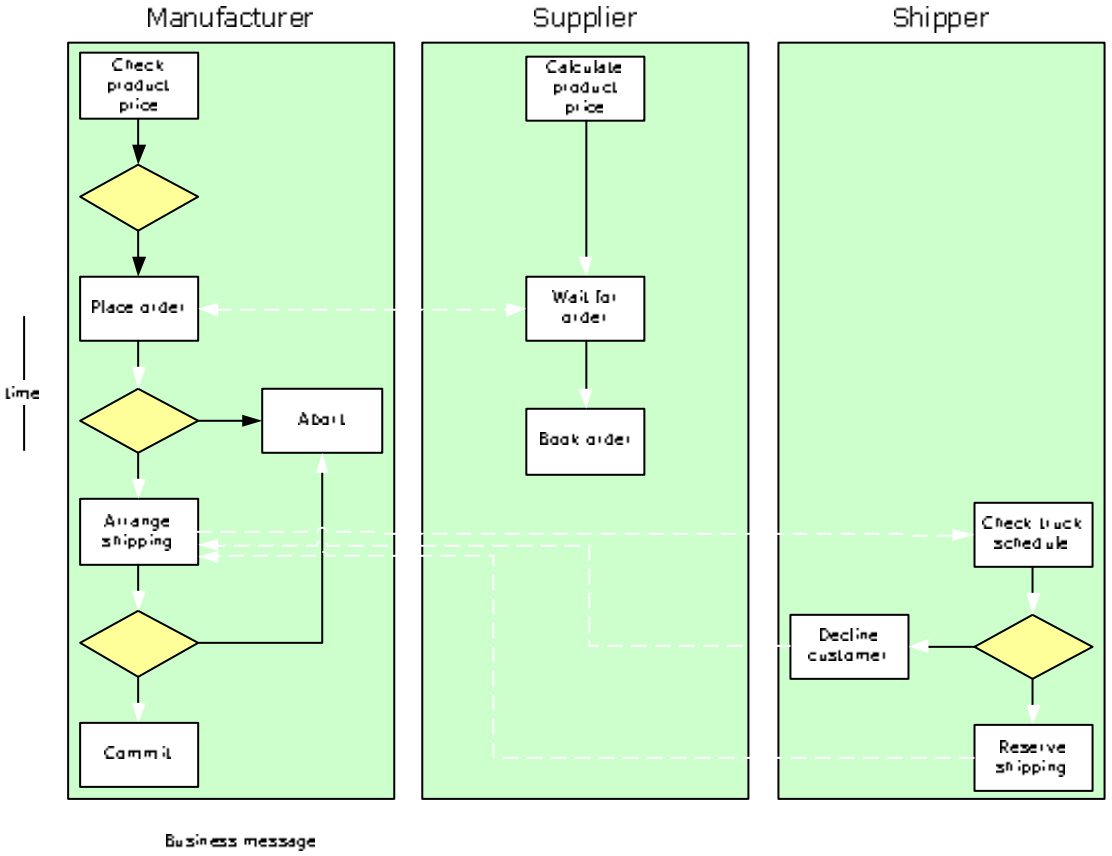
specific ways in which it must terminate (e.g., purchase the shares at the current price if it is less than a certain value). However, while this business transaction is running, its constituents may still require some or all of the ACID properties associated with transactions.

5. Arranging a meeting

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

6. Manufacturer-Supplier-Shipper

Let's consider an example business transaction scenario depicted in Figure 4.



A manufacturing company (Manufacturer) needs to order parts from one of its partners (Supplier). In order to make its production schedule, the Manufacturer has to make sure that the parts are shipped from the Supplier in a given timeframe by a logistics provider (Shipper), otherwise the Manufacturer would not be interested in these parts. All the parties in this example have automated computer systems that can communicate with each other via XML messages. Below we describe the interactions of this business transaction:

1. Manufacturer's production scheduling system sends an "order" message to Supplier.
2. Supplier's order processing system sends back an "order confirmation" with the details of the order.
3. Manufacturer orders delivery from Shipper for the ordered parts. Delivery needs to occur in two days.
4. Shipper evaluates the request and based on its truck schedule it sends back a confirmation or a "can't do" message.
5. Manufacturer either confirms the order and the shipping or it cancels the order, since the shipper was unable to fulfil the request

From the Manufacturer's point of view all the business messages described above belong to a single business transaction. The underlying systems need to make sure that this business transaction is

- *Atomic*: The parts either get ordered or the order gets cancelled.
- *Consistent*: If the parts get ordered, the shipping gets set up. If the shipping company cannot promise shipping with the required terms, the order is cancelled.
- *Durable*: All parties persist the outcome of the transaction.

Typically, the Supplier will have multiple business transactions with multiple Manufacturers executing concurrently. The time between the order is placed and it is confirmed can be long. Therefore it is not feasible for the Supplier to lock its order database and wait until the confirmation comes. It will rather book the order when it is placed and in case the order gets cancelled, it will invoke a compensating action to remove the order from the database. This means that concurrent business transactions are not executing in isolation: they are exposed to partial updates made by other concurrently executing transactions.

These requirements are not special to this example; in fact they must be met for any transaction that is critical to the business of the participating companies. Therefore an application independent facility should exist that can manage the mission critical multi-company business transactions to ensure the properties above.

7. Financial Trading

Many financial trading strategies are composed of several underlying financial products.

For example, an investor wishes to buy stock in company A in the belief that the stock price will rise, but at the same time wishes to 'insure' the investment against an unexpected drop in the stock price.

This strategy can be achieved by buying the stock in company A and at the same time paying a premium to purchase an option to sell the stock at an agreed price (a 'put' option). If the stock price rises the return for the investor is the price gain less the

premium paid for the option. If the price falls unexpectedly, the investor exercises the option to sell the stock and limits the loss.

To construct this trade the investor requests quotes on the current price of A's stock and also on the 'put' option on stock A. Quotes on the stock and the option may be available directly from several exchanges, or indirectly through intermediaries or brokers. Prices may vary from exchange to exchange and from broker to broker for many reasons, including differing transaction costs, broker spreads and option valuation models.

The 'value' of such a trading strategy requires the successful purchase of each component or 'leg' of the trade at a certain price and at the same time.

Using a cohesion protocol the investor can automate the quote gathering, quote choosing and execution of the trade.

Within a single cohesion, the investor gathers quotes for the stock and the option from a variety of exchanges and brokers. The quotes are only valid for a specified time after which they 'time-out'. The stock quotes are sorted according to some defined business logic and then the cheapest executed. Similarly, the option quotes are sorted according to some defined business logic and then the cheapest executed. The other quotes are ignored and simply time-out.