# Business Transactions, Compensation and the Try-Cancel/Confirm (TCC) Approach for Web Services

**Author:** Guy Pardon, Atomikos (http://www.atomikos.com)

## *Introduction*

As web services mature and more and more implementations start to appear, the discussions around related technologies and protocols such as security, transactions and reliability are also becoming more relevant. This article focuses on transactions for web services, and particularly on the concept of long-running transactions and the implications for web service providers. In particular, we will look at the ways in which transactions and compensation fit the business model of a web service provider, and the implications with respect to the architectural support that is needed.

## *Definitions*

Before we get started, let's make sure we agree on some vital terminology.

**Transaction**: for the purpose of this article, a transaction is a set of related interactions (or operations) that may need to be cancelled AFTER they were executed. These operations can be at different geographical locations, like different web services on the Internet.

**Long-running transaction (or business transaction)**: for the purpose of this article, a long-running transaction is one where the total duration exceeds the duration of an individual interaction by several orders of magnitude. Consequently, a long-running transaction is a transaction where the cancellation of an interaction may happen a relatively long time after the interaction was executed.

**Example**: Figure 1: Booking a flight at two web services shows a trip reservation: scheduling a trip from Brussels to Toronto can consist of the booking of a flight from Brussels to Washington, and then a second booking of a connecting flight from Washington to Toronto, the final destination. Suppose the second flight is with a different airline, in a different reservation system.

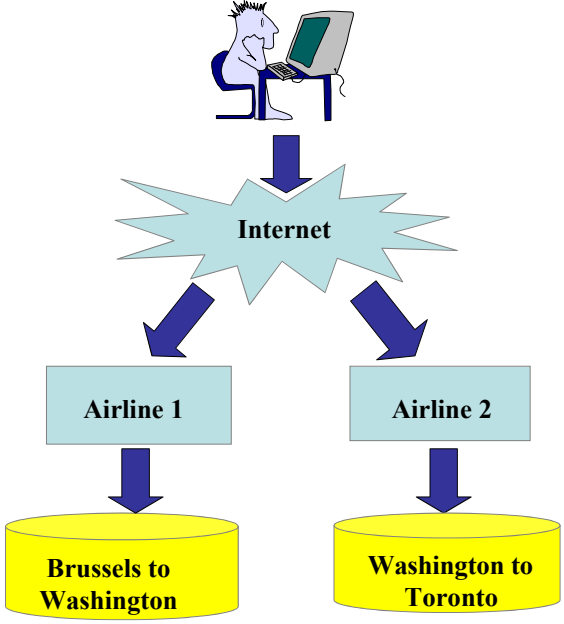



**Figure 1: Booking a flight at two web services**

If the second part fails then I am stuck with a ticket from Brussels to Washington (which can certainly be interesting, but it defeats the purpose of my trip planning). So in that case, it can be a valid option to cancel the trip to Washington. However, before I decide to do so, I may have been looking for alternatives like another connecting flight two hours later or so. This can potentially lead to a very long-running overall transaction.

Note that the crucial point here is that the reservation has already been made at the time I decide to cancel. That is, the airline reservation system has accepted and verified my first reservation, and is keeping my seat for at least some time. If there is no cancel event then either I will get booked with a ticket I don't need, or the airline is going to lose money on an unneeded reservation. Either way somebody loses money. Consequently, there are strong drivers for the existence of a cancel event.

Speaking purely technically, the cancellation of different interactions across different locations is nothing new: it has been done for decades by something called a transaction manager or transaction service. CORBA's OTS (Object Transaction Service) is an example of such a technology, J2EE's Java Transaction API and Java Transaction Service (JTA/JTS) are another example.

However, these technologies are all based on the concept of **ACID transactions**. The way ACID technology works is as follows: all data a transaction accesses become locked from other concurrent transactions until the transaction either commits (saves changes) or rolls back (cancels changes). This is illustrated in Figure 2: Lock time for ACID transactions.

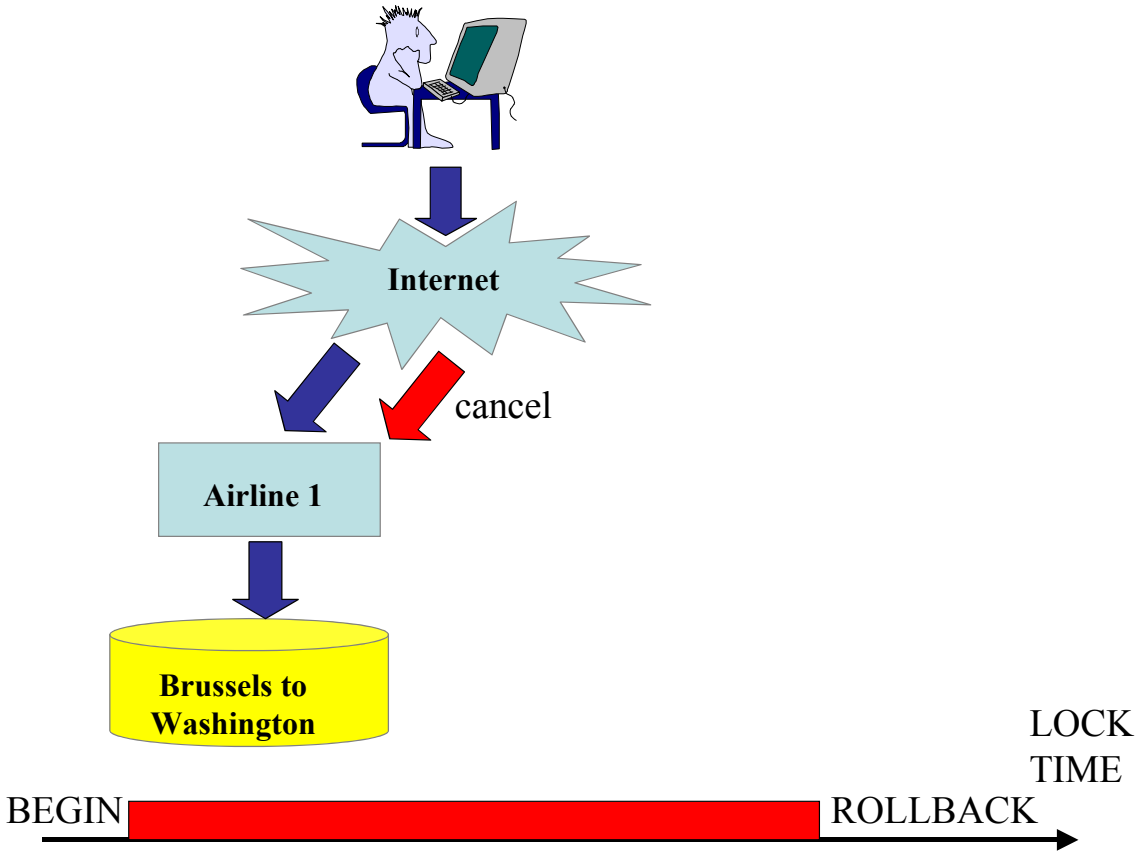



**Figure 2: Lock time for ACID transactions**

For long-running transactions this is far from ideal, because cancellation would only be possible if all the data were kept locked. Keeping locks on behalf of web service interactions may not be desirable for different reasons such as:

- Web services may require a long delay before cancellation is done, especially if there are human decision times involved. This would imply long lock times, and equally long data unavailability.
- Web services may involve parties that don't know or don't trust each other, making them vulnerable to denial of service (DOS) attacks at the database level.

So if ACID and rollback are not going to help us then what is?
A possible answer, as you may have guessed, is compensation.

Instead of one implementing a long-running transaction as one giant distributed ACID transaction, a compensation-based approach treats each web service invocation as one (short) local ACID transaction, committed as soon as it has executed. This drastically reduces lock duration, at the expense of losing rollback ability. This means that cancellation has to be done differently: by executing a separate local ACID transaction that logically cancels the work after it was

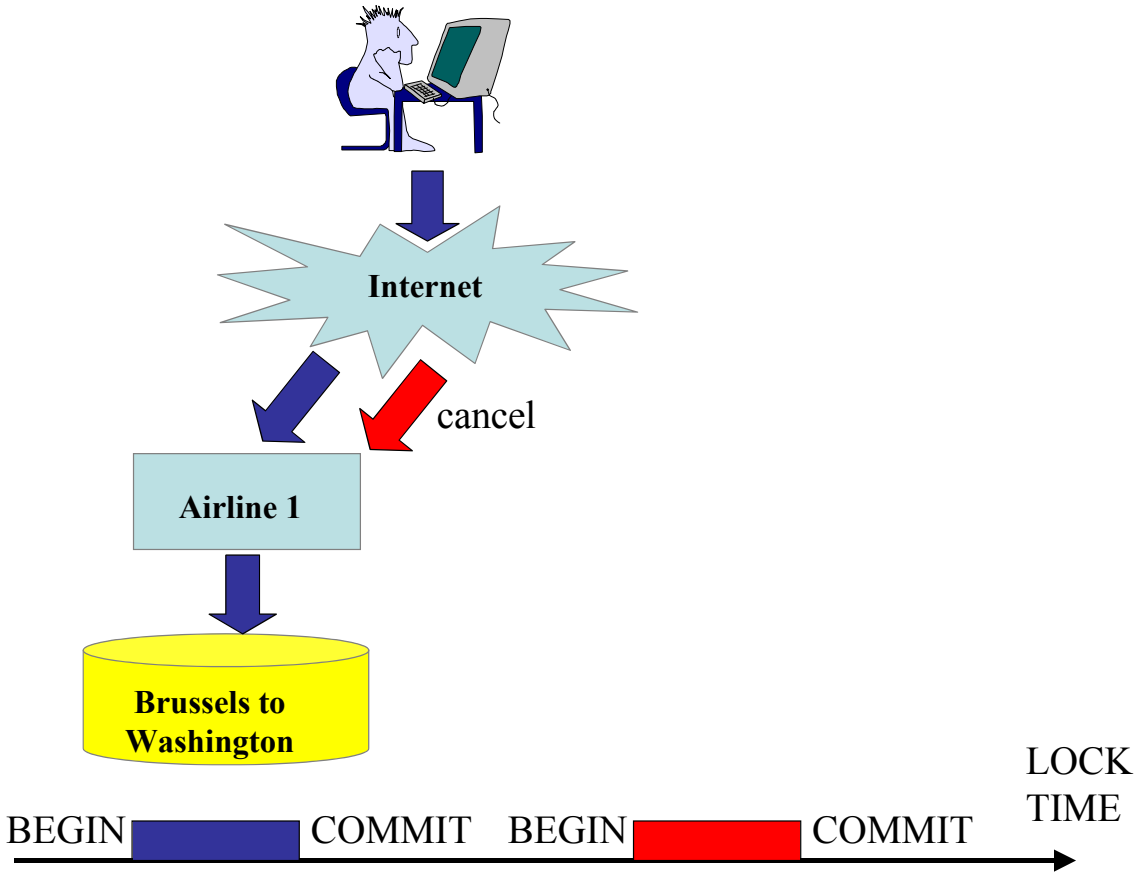
committed.

**Compensation**: for the purpose of this article, compensation is defined as a separate ACID transaction (local to a web service provider) that logically cancels the effects of a previous ACID interaction.

A compensation-based approach keeps database lock times to a minimum, thereby avoiding all the drawbacks mentioned before.
Let's see how that works.

Consider again the example of the airline reservation. Making the reservation of the flight to Washington would be an ACID transaction in its own right, local to the airline system and committed immediately. When there is a need to cancel, a second ACID transaction would logically undo the effect of the reservation. The resulting lock times are much shorter.

**Figure 3: Lock time with compensation**




How exactly does this work? It depends on the application, and the application programmers are responsible for the specification of what they need to do. For instance, canceling an airline reservation could be done in different ways: we could delete the reservation record (leaving no traces at all) or we could explicitly mark it as canceled. Either way is possible, and what you want is highly

dependent on the business model of the web service provider.

In general, we can distinguish two types of compensation:
**Perfect compensation**: the reverse logic cleans up all effects of the original. This is the case where we delete the reservation entirely.
**Imperfect compensation**: this happens if the reverse logic leaves detectable traces of the original. For example, if we explicitly update the reservation to status 'canceled'.

## *Compensation as seen from the business model*

Now that we have seen the notion of compensation, let's explore how it maps to the business. When we look at the business model of a service provider, two major distinct categories can be identified. I will now discuss each one in turn.

### Case1: compensation is just another business transaction

In this case, the compensation is nothing special as far as the provider is concerned: the compensating task is something that could just as well be a regular business interaction. In particular, the web service has no need to offer an additional service implementation to do the compensation and remains very simple.

An example of this kind of business is stock brokerage: if you buy stock, then you can compensate that by selling it again afterwards. Of course, everybody who has invested in the last years knows that this model of compensation is not perfect at all: the chances are very low that the stock price is still the same for buying versus selling. This means that the customer who requests the compensation either loses money or wins.

Nevertheless, this kind of compensation is interesting because the service provider doesn't have to worry about compensating logic. It is the simplest kind of compensation-based system in which the service provider is stateless.

### Case2: compensation is a second stage of the same business transaction

In this case, the compensation is an explicit second stage of the same business transaction involving custom logic and coding as well as business transaction state to be kept. The example with the airline reservation is of this category (contrast this with the stock-broker example we just saw). We can call this approach the **TRY-CANCEL/CONFIRM** (TCC) approach. TRY represents the normal business logic, such as the reservation of a seat. CANCEL represents the possible cancellation via a compensation mechanism. On the other hand, if there is no need to cancel then CONFIRM signals this fact. CONFIRM is also an ACID transaction to update the reservation state in the database.

Why the CONFIRM? To understand this we need to look at the states of a reservation during its lifetime. When a reservation is first made, its business state is PENDING. To the provider (the airline) this means that the reservation is not yet final, in other words: it can still be canceled upon customer request. This state is important for the airline's reporting and booking services: it should not (yet) take the reservation for keeps until it knows it is final.

If the airline prefers, it can also build in a timeout after which it will autonomously CANCEL the reservation. This could happen if I behave badly and just place reservations without ever confirming them.

However, as soon as a CONFIRM signal comes in, the web service knows that the reservation is permanent and can be billed to the customer.

The TCC approach is ideal for business models that are already two-phased, such as existing business models that are somehow based on the notion of a **reservation**. A service provider needs to expose extra interfaces to deal with CANCEL and CONFIRM events.

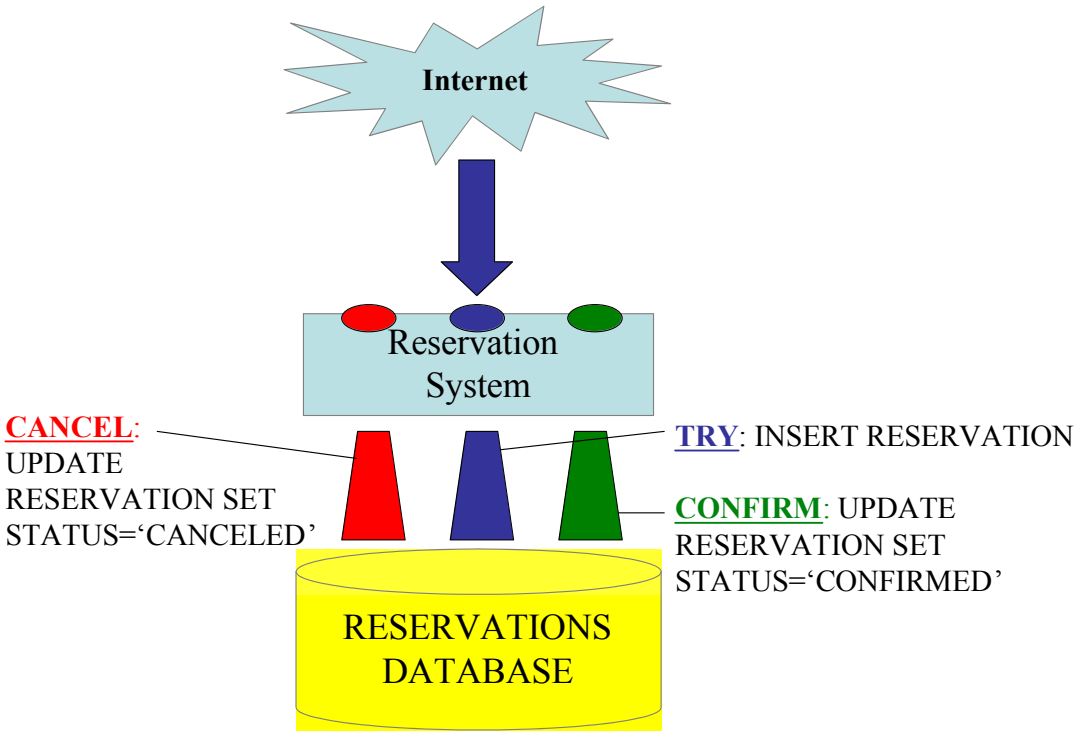



**Figure 4: The TCC web service model**

## *Architectural support for long-running transactions and compensation*

Now we can discuss some architectural implications for compensation-based

systems. The implications are different for each type of business model we discussed.

### Case 1: If compensation is a regular business operation

If the compensation can be considered part of the business as usual (as in the stock broker example) then there is nothing that the stock-broker has to worry about. Any stock purchase is compensated by a later sale, and the stock-broker probably earns his or her commission on both. There is absolutely no business risk involved, and therefore this is the most flexible case for compensation.

The architectural support for this kind of compensation is perfectly feasible via regular workflow technology such as the Business Process Execution Language standard (BPEL). A remote workflow engine can be used to model and execute the overall task execution, and the purchase of stock is just merely one step in the overall process. Should the process (or long-running transaction) need to be canceled, then the workflow engine sends a business request to sell the stock it previously acquired. Nothing else is needed here. Also, the stock-broker doesn't care how long it takes before something is compensated for, simply because to him or her it is merely another separate business transaction: there is no need for broker-specific state that links the compensation to the original purchase.

### Case 2: The TCC model

For TCC processes, the reality is more complicated. In particular, there is little chance that the business owner of a TCC service will ever allow a remote workflow to take full responsibility of its compensation, because there is an implicit reservation of resources for the duration of the TCC process. The state management for a TCC transaction is something that naturally belongs at the service provider.

For instance, in the flight reservation a seat is booked as long as the reservation is still PENDING. During this time, the reservation is not available to other potential customers. Consequently, here it very much matters how long the process takes, and it is unlikely that the business owners will allow a remote workflow engine to keep the reservation pending all the time.
This means that there will be a timeout after which the reservation is canceled autonomously by the reservation service. A logical consequence is that the reservation service at least needs to know how to cancel. In other words: the cancellation logic is co-located with the service. Triggering the cancellation is most likely to happen via an event that carries the transaction's token (ID), since the logic is already on the service anyway. The same holds for the CONFIRM logic.

Just like the business model, the resulting architecture follows a two-phased protocol: TRY is the first phase, whereas the second phase consist of either a

CANCEL request or a CONFIRM request. These arguments support the need for a two-phased transaction manager for TCC systems. A service provider could use such a plugin to automate the state management of its TCC transactions.

A workflow engine could also use such a plugin, to automate the process of sending CANCEL or CONFIRM requests to all relevant sites. For instance, in a BPEL engine a transaction management plugin could detect all remote calls and therefore know all participants in a distributed task. When a cancellation is requested, the transaction management plugin contacts each participating web service and asks it to CANCEL its part of the work. If we keep in mind that every workflow step can have its compensation, then this would reduce workflow-modeling complexity by about 50%: instead of explicitly modeling each compensation as a step in the workflow logic, the transaction management plugin keeps track of what sites need to compensate and takes care of the cancellation when needed. Consequently, all compensation steps can be left out of the workflow logic (which therefore becomes much simpler).

## *Conclusion*

As I have tried to show in this article, there are essentially two major types of compensation-based service models: the stock-broker type (a stateless service) and the stateful reservation type (try-cancel/confirm or TCC for short). For the former, a workflow architecture on Internet scale is sufficient. For the latter, a two-phase protocol is a more natural fit, with the additional potential of reducing workflow-modeling complexity by at least 50%.

## *Some Final Remarks*

There are a number of questions I have chosen not to deal with in this article, mainly for reasons of space. Among them:

- What happens if compensation fails?
- How can we deal with communication failures and retried requests, possibly leading to duplicate invocations of the same web service?

## *About The Author*

Guy Pardon is a founder of Atomikos (http://www.atomikos.com), a company specialized in transaction processing software. Guy is co-inventor of several patents related to transactions and compensation. In an earlier life, Guy has done extensive research on transactions and compensation in a federated (web service) architecture and has invented and built the first web service transaction system even before SOAP existed. At Atomikos, Guy is responsible for product development. You can contact Atomikos for more information or for proof-of-concept projects with web service transactions.

## *References*

- Composite Systems: Distributed Nested Transactions. PhD Thesis, Guy Pardon, December 2000.
- The OTS specification (http://www.omg.org/technology/documents/formal/transaction_service.htm) specifies the interfaces towards a CORBA transaction manager.
- BPEL: Business Process Execution Language is a formal language for expressing workflows over web services, enabling subsequent interpretation and execution by a workflow engine. For more information, check out http://www.oasis-open.org .
- JTA: The Java Transaction API specifies the interfaces towards a transaction manager from within a Java program. More information on http://java.sun.com/products/jta (specifications only). For a working JTA product, check out http://www.atomikos.com .