Atomikos™ ExtremeTransactions™ Guide

Atomikos ExtremeTransactions Guide

Table of Contents

List of Figures

List of Examples

Chapter 1. Introduction

This guide introduces you to the revolutionary transaction technology incorporated into ExtremeTransactions™, the commercial version of our TransactionsEssentials™ core.

Distributed transactions have always been a debatable topic where the numerous opponents typically used to complain about the poor characteristics of traditional *ACID* standards in this area (where *ACID* means Atomic, Consistent, Isolated and Durable - see the Glossary in appendix).

This guide will show you how times have changed: *thanks to Atomikos™, it is now possible to build distributed (and even asynchronous) systems with transactional guarantees without having to resort to the limitations of traditional ACID technologies* (although you can still do that if you want).

1.1. Who Should Read This Guide

You should read this guide if you are interested in any of the following:

- Learning about transactional services and GRIDs and what they can do for you.
- Enabling your web services for ExtremeTransactions™.
- Making your existing *JTA/XA* transactions participate in distributed service transactions.
- Programming state-of-the-art, compensation-based services (departing from *ACID*).
- Supporting distributed transactions without the classical XA support (unlike *ACID*).

1.2. Prerequisite Reading

> You can read to this guide to grasp the concepts, but in order to really implement services with ExtremeTransactions™ you should be familiar with the concepts explained in the guides *AtomikosTransactionsEssentialsGuide* and *AtomikosAPISpecification*. If you haven't read those yet, please do so first.

1.3. System Requirements

The following platform is required for ExtremeTransactions™:

- Java 1.4 or higher.
- At least 128 MB of RAM.

1.4. Installation Instructions

Please see the installation instructions in the getting started pages.

1.5. The Impact of SOA on Transactions

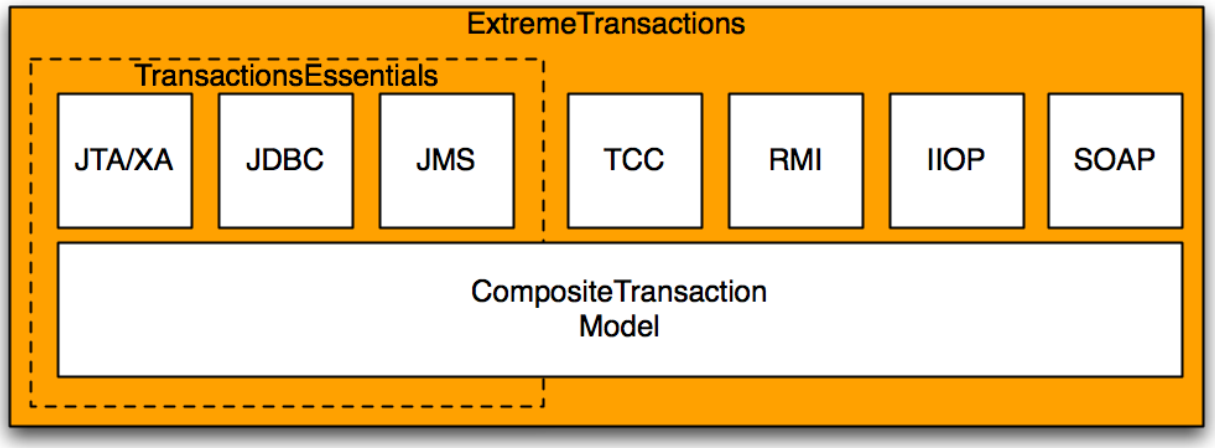
A service-oriented architecture requires fundamental changes from the traditional architectural viewpoint: traditional monolithic applications are (by definition) built in standalone mode, meaning that distributed transactions are rarely needed. By contrast, services are meant to be only a part of a composite application.

The result: many service-based applications of the new generation will be distributed by default. This drastically overturns the traditional architectural requirements in that distributed transactions are now more of a necessity than ever before. ExtremeTransactions™ answers these new needs.

1.6. Architecture

The architecture of ExtremeTransactions™ is shown below, along with its relationship to TransactionsEssentials™.

Figure 1.1. ExtremeTransactions Architecture



The main features offered by ExtremeTransactions™ (as shown on the diagram) are the following:

- *JTA/XA* support: you can start and commit/rollback transactions according to Sun's JTA API.
- *JDBC* support: connection pooling and *JTA/XA*-aware datasource implementations.
- *JMS* support: message-driven receivers and senders - even outside the application server.
- TCC™ support: our revolutionary compensation-based transaction model for loose coupling. With TCC™, you can focus on the happy path of your workflow logic, and let us take care of the rest.
- Extending transactions across the network: the scope of rollback and commit can be extended to other processes, called by *RMI*, *IIOP*, *SOAP* or almost any other protocol.

1.6.1. Features Inherited from TransactionsEssentials™

ExtremeTransactions™ is the commercial extension of TransactionsEssentials™ so it contains all the functionality offered by the latter:

- *JTA/XA* support for *ACID* transactions
- *JDBC* support
- *JMS* support

1.6.2. TCC™

One of the main innovations found in ExtremeTransactions™ is support for the TCC™ (Try-Confirm/Cancel) model. As explained later in this guide, TCC™ is a paradigm for the construction of transactional services while

preserving the capabilities for loose coupling and asynchronous interactions. This way, ExtremeTransactions™ effectively eliminates the main argument against a transactional SOA: there is no more extended locking of data (required for traditional two-phase commit), and the use of local transactions suffices in order to get distributed consistency.

1.6.3. Extending Transactions Across the Network

A transaction started in one application (or service, depending on the context) can optionally be extended ("propagated") across the network. As a result, rollback or commit of the original transaction will include all the work done in all of the processes to where the transaction was propagated. This mechanism works both for classical *JTA/XA* transactions as well as for networked TCC™ services.

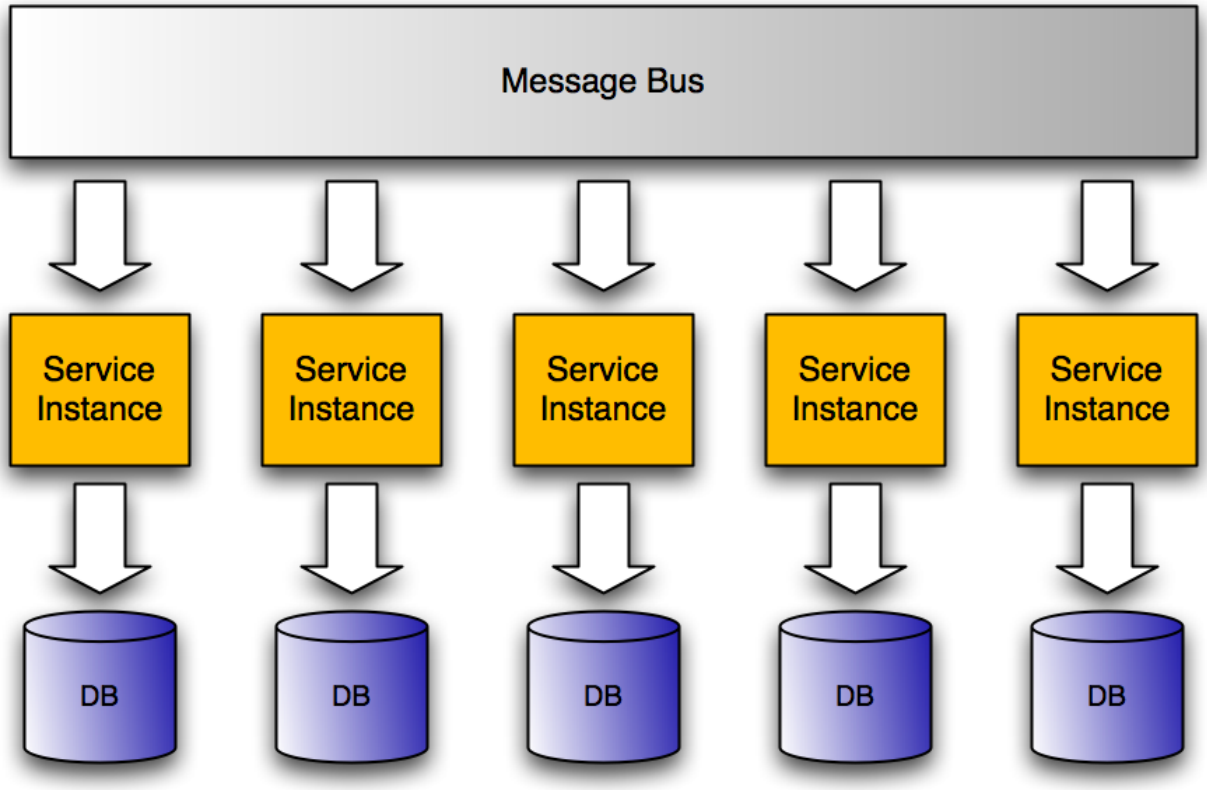
This extension mechanism allows for the construction of transactional composite applications; the TCC™ paradigm avoids tight coupling and eliminates traditional locking problems on the data.

Chapter 2. Writing Transactional Services with ExtremeTransactions™

2.1. Intra-VM Transactions: XTP/GRID Architecture

An intra-VM transactional service *GRID* is usually based on *JMS* for receiving messages from some *message bus* and then inserting some results into a database via *JDBC*. Gartner™ has termed this style of processing *XTP (extreme transaction processing)*. The usual architecture is shown below.

Figure 2.1. XTP GRID architecture with intra-VM transactions



ExtremeTransactions™ guarantees that the messages (requests) are either still on the message bus, or in the database, but nothing in between. In particular, ExtremeTransactions™ avoids message loss (missing requests) or duplicate delivery.

For examples of applications that follow this architecture, see the *JMS* examples in the download of ExtremeTransactions™.

It is important to realize that for this architecture, the typical scope of a transaction is the processing of individual (queued) requests. In particular, the sender of a request has no control over the outcome of its processing. If such control is desired then we recommend our TCC™ approach instead.

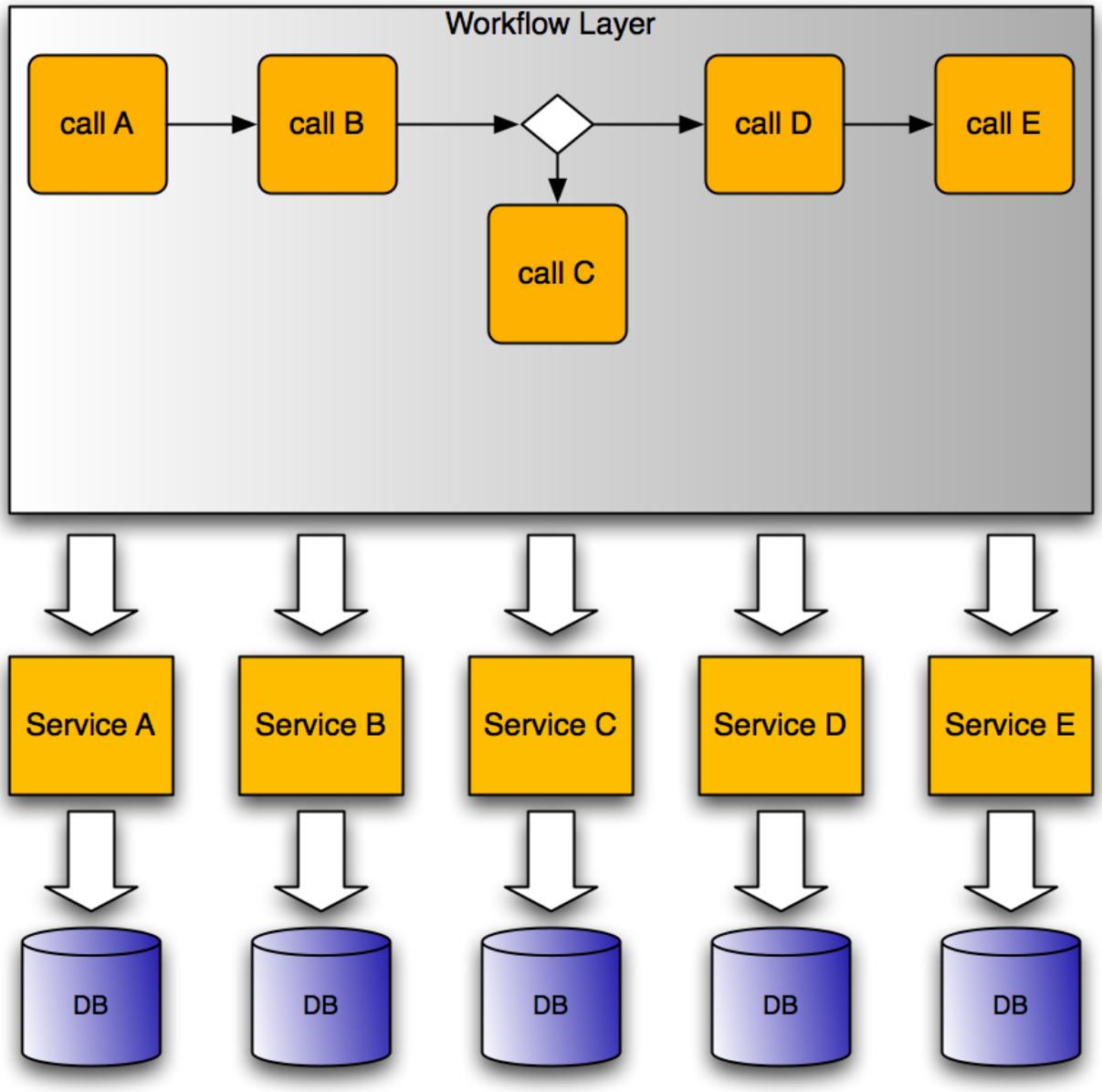
2.2. Inter-VM Transactions: Composite Applications and SOA

SOA and the *composite application* (an application composed of multiple related service calls to different services) can require something more complex than the previous architecture. This section outlines the problems encountered, and two solutions offered by ExtremeTransactions™.

2.2.1. The SOA Problem Illustrated

SOA applications are different from the previous case: the transaction scope effectively spans multiple services and clients. For a *composite application*, this means that a transaction can extend over the entire workflow. An example of such a workflow is shown below.

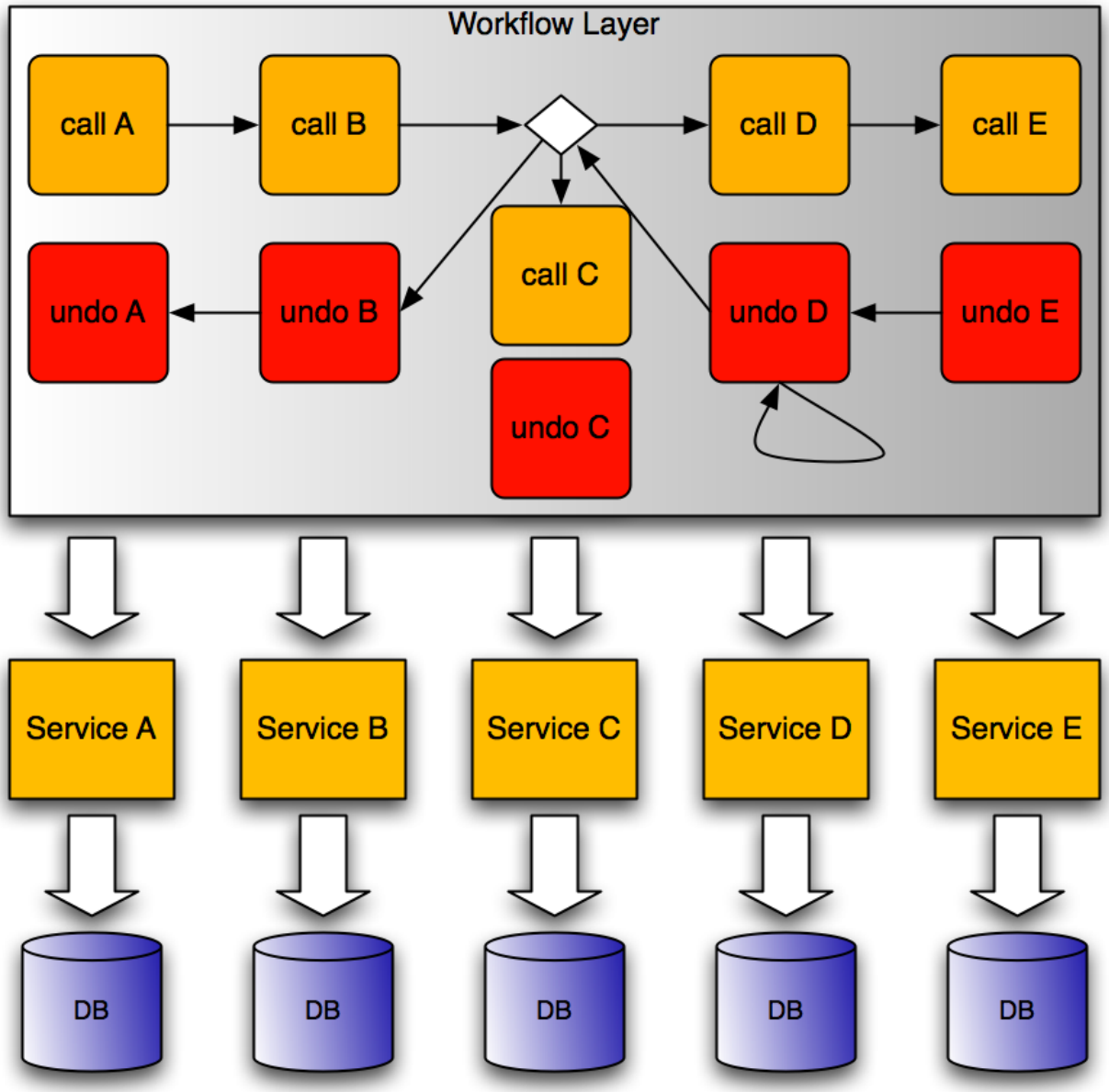
Figure 2.2. Example of a composite application SOA workflow



So far we have only shown the *happy path*: as long as no failures or crashes happen, this will work. However, in the realistic case of failures or crashes, the workflow of a composite application becomes a lot more complex: not only do we need to add undo logic in the workflow model, in addition we also need to model the inter-dependencies among undo operations and the right order. Finally, even more complexity comes into play when one considers the possibility of failures in the undo logic itself and when to retry the undo.

The result is often a composite application that becomes more complex than it should be. Moreover, the reliability goes down a long way due to the increased complexity, decreased testability and worse maintainability. This does not scale. The figure below shows this approach.

Figure 2.3. Example of a composite application with error logic



2.2.2. Solution 1: ExtremeTransactions™ JTA Support

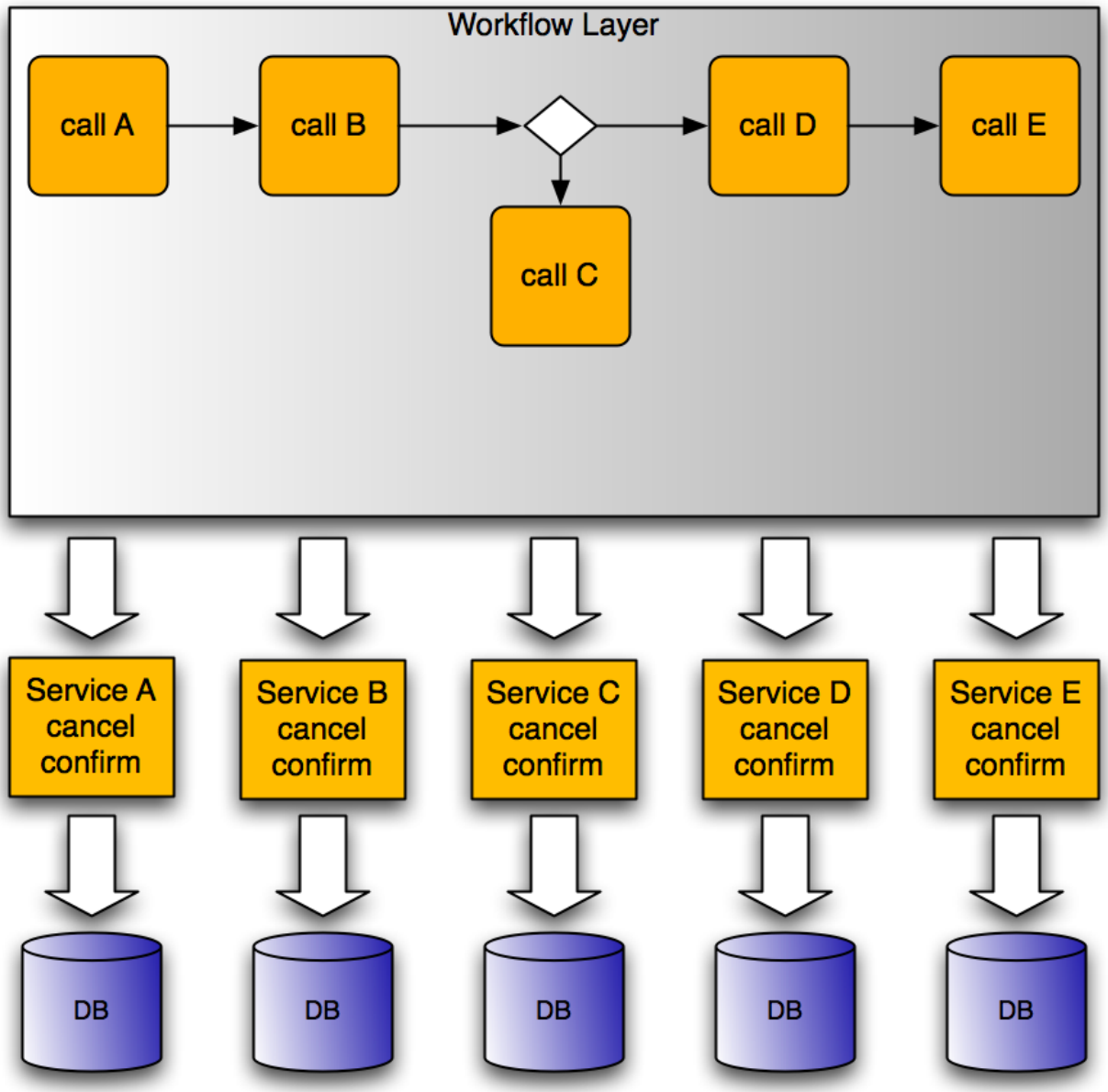
One way to solve the problem is by making the whole workflow an extended (distributed) *JTA* transaction. While this works from a technical viewpoint, it does not scale outside the enterprise: database locks are maintained for the entire duration of an extended transaction, and this exposes the services to *denial-of-service* attacks and other availability hazards. Consequently, this solution is only fit for intra-enterprise cases where there is a centralized point of control.

> Examples of this approach are shown in the installation folder of ExtremeTransactions™, under *examples/j2se/rmi*.

2.2.3. Solution 2: ExtremeTransactions™ TCC™ Support

With TCC, the workflow logic of a *composite application* can be reduced to its *happy path*. All other logic is moved to the service implementation: the services are now offering both *cancel* and *confirm* logic as well. This logic is moved out of the workflow (along with all interdependencies), thereby removing all complexity from the workflow itself. This is shown below...

Figure 2.4. TCC: focus on the happy path



At the expense of some additional (and reusable) logic *in each service*, the workflow is simplified enormously: there are no more undo calls to model/program, and no dependencies to take care of. The developers don't have to take into account all possible failure paths, nor do they have to track where things go wrong and what to do next. All this is handled by ExtremeTransactions™. In addition, the failure of undo operations is no longer a worry of the application developer: all this is handled by ExtremeTransactions™.

Examples of this approach are shown in the installation folder of ExtremeTransactions™, under *examples/j2se/tcc*.

2.3. The Atomikos™ Try-Confirm-Cancel (TCC™) API: Distributed Service Transactions Without XA

This API is a revolutionary approach to programming distributed transactional services (which can be exposed either as web services or as classical *RMI*/*IIOP* services). It combines the best of two worlds:

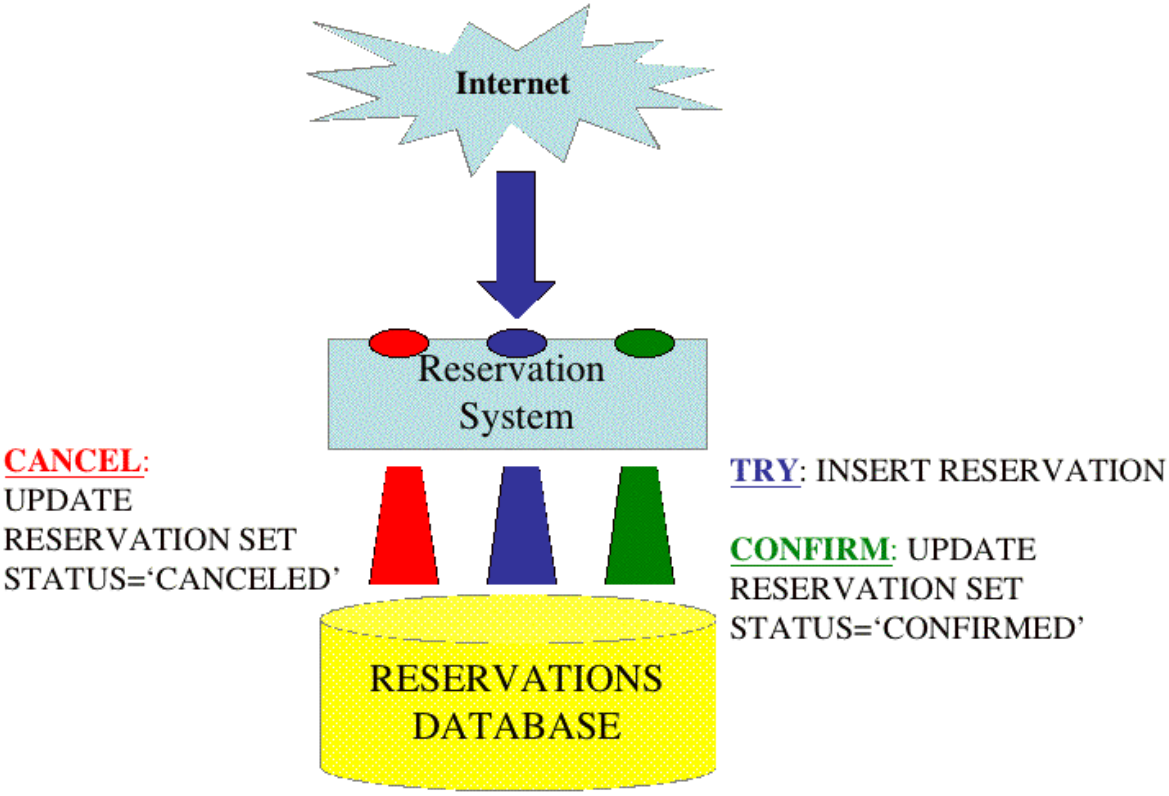
- The loosely-coupled style of messaging platforms, by supporting asynchronous and long-duration communication patterns.
- The reliability guarantees of transactions, by offering the guarantee that a distributed (and possibly asynchronous) task is either canceled or confirmed in its entirety.

Invented by Atomikos, this approach is new in the way that distributed transactions are structured: instead of requiring one long ACID transaction that lasts until commit or rollback, TCC™ splits up a web service transaction into three distinct phases, each optionally involving separate and short-lived ACID transactions:

- The *TRY* phase: from the viewpoint of a service provider, this is where the normal transactional service request is processed in a *tentative manner* (i.e., subject to later confirmation or cancelation) and in one short *local* ACID transaction (which can even be a non-JTA transaction such as in JDBC™). None of the classical distributed and long-lived locks are required to do this. At the end of this phase, the business logic reflects a tentative result that will become permanent only after the next phase (either *CONFIRM* or *CANCEL*).
- The *CONFIRM* phase: if the overall web service transaction commits, then the TCC™ service implementation receives a confirmation notification (this corresponds to the *commit* in the two-phase commit protocol). The interesting part is that this confirmation may trigger business-level processing to update the tentative business state to *confirmed* (again, this update can be a purely local transaction such as in JDBC™). The TCC™ paradigm only requires *local updates* during this phase: any remote confirmation is done by the protocols in the background.
- The *CANCEL* phase: if the overall web service transaction does rollback then the TCC™ service implementation receives a cancelation notification (this corresponds to the *rollback* in the two-phase commit protocol). Like for confirmation, this notification may trigger business-level processing to update the tentative business state to *canceled*. Again, only local updates are required; any remote cancelation is done by the protocols in the background.

Example 2.1. TCC™ example: an airline reservation web service

Figure 2.5. TCC™ example: airline reservation service



As an example, consider an airline reservation web service. This service offers online seat reservations that can be booked in a transactional way. The *TRY* phase inserts the reservation into the database. Upon commit, the *CONFIRM* phase explicitly updates the reservation to being valid. Otherwise, in the event of *CANCEL*, the reservation is marked to be canceled; because the business logic is aware that only confirmed reservations should be taken into account, this achieves transactional consistency in a loosely-coupled way.

2.3.1. Programming TCC™ Applications

The TCC™ paradigm is ideal for reservation-style (web) services, where business resources (seats, tickets, ...) have to be reserved for a short or large period of time until the reservation is confirmed or canceled. You can view TCC™ as *two-phase commit at the business level*: the tentative (prepare), canceled (rollback) and confirmed (commit) states are visible (and known) by the business logic. Tentative transactions are indeed accessible in the database(s), and should be ignored by the business if not relevant. The system offers the guarantee that tentative transactions are temporary, so resources are always released eventually. Indeed, a TCC™ transaction is either confirmed (if it succeeds) or canceled (if it fails or times out). Either way, the corresponding business logic is triggered (CONFIRM or CANCEL) to release any business-level resources associated with the tentative state.

The TCC™ API is defined in the package `com.atomikos.TCC` and implemented by the package `com.atomikos.icatch.TCC`. See the javadoc and the samples contained in the release for more details on the TCC™ model.

2.3.1.1. Recommended Pattern for TCC™ Services

The TCC™ paradigm corresponds to two-phase commit at the application level. This means that the likelyhood of failure during *CANCEL* or *CONFIRM* should be minimized (since these correspond to failures during the second phase of two-phase commit). Therefore, we recommend that your TCC™ services be developed along the following lines:

- *TRY*: reserve the necessary resources to make CONFIRM succeed. For instance, if you are selling something then TRY would check and decrement stock availability already, so CONFIRM can't run into availability problems.
- *CONFIRM*: make the work of TRY permanent, i.e. convert the reservation into a purchase (in the case of selling).
- *CANCEL*: release the reserved resources again. In the selling example, this would mean returning the reserved items to the available stock.

2.3.1.2. Programming the TCC™ Service

TCC™ service implementations should implement the interface `com.atomikos.TCC.TCCService`. This interface exposes the following methods to implement:

- `confirm`: the logic associated with confirmation. Implement this method to confirm the reservations made in the `try-phase`.
- `cancel`: the logic associated with cancelation. Implement this method to cancel the reservations made in the `try-phase`.
- `recover`: callback for recovery; this method is called by the transaction service when it recovers TCC™ transactions after a crash or restart. Any required system-level resources needed for cancel or confirm can be re-acquired in this method.

In addition to these termination callbacks (required by the transaction service), it is up to the application itself to implement the logic for the try-phase. As far as the transaction service is concerned, the try-phase starts when `register` is called, and ends when one of `completed` or `failed` is invoked by the application. An example TCC™ service implementation is shown below (the complete example is included in the demos for the release).

Example 2.2. Example of a TCC™ payment service

```
package payment;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;

import com.atomikos.icatch.HeurCommitException;
import com.atomikos.icatch.HeurRollbackException;
import com.atomikos.TCC.TCCException;
import com.atomikos.TCC.TCCService;
import com.atomikos.TCC.TCCServiceManager;
```

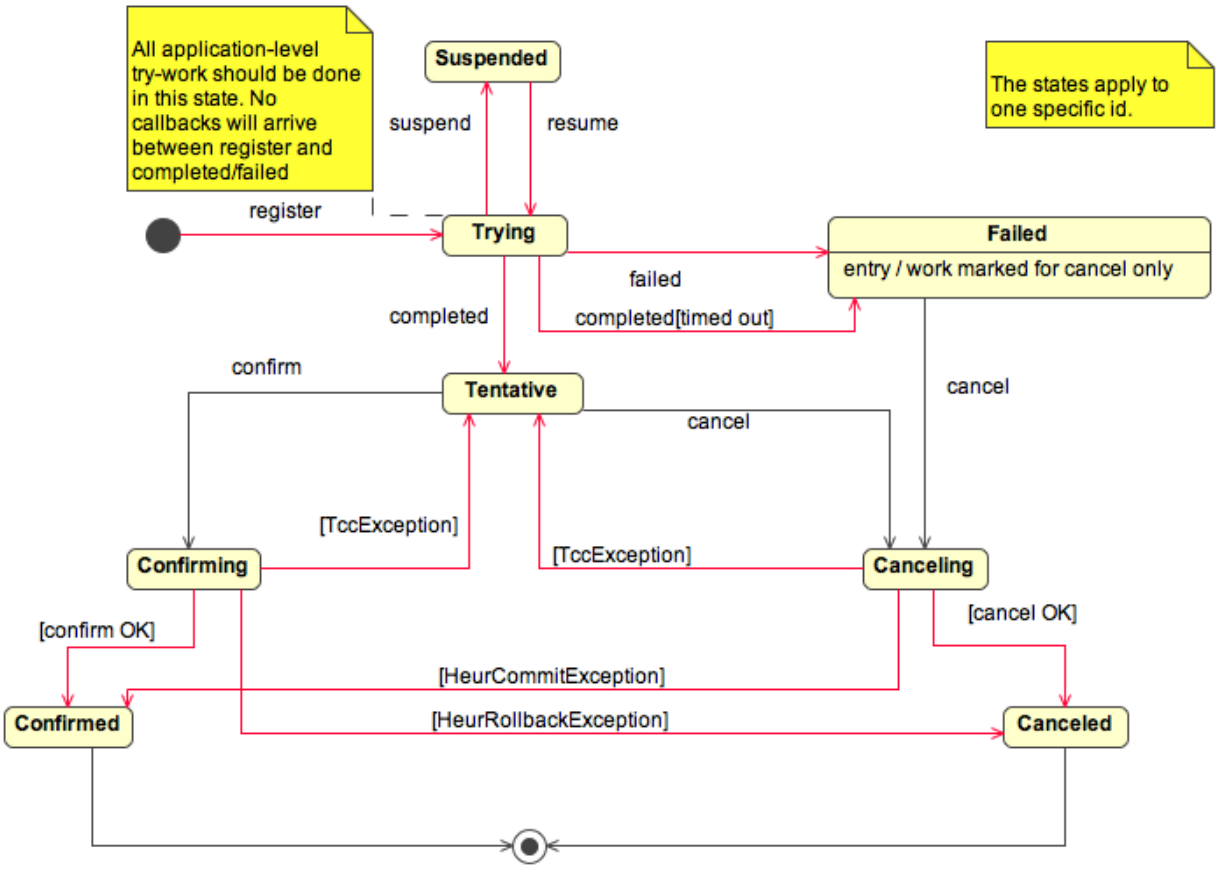
```
public class PaymentTCCService implements TCCService
{
 private TCCServiceManager TCCmgr;
 private long timeout;

 public PaymentTCCService ( TCCServiceManager TCCmgr ,
   long timeout )
 {
  this.TCCmgr = TCCmgr;
  this.timeout = timeout;

        //IMPORTANT: register with the system for recovery callbacks
        //to assist with application-level recovery of previous
        //service invocations

        TCCmgr.registerForRecovery ( this );
 }

 public void tryPayment ( String cardNo , int amount )
 throws Exception
 {
  //start (register) the work for cancel/confirm
  //and obtain the id to use as reference
  String id = TCCmgr.register ( this , timeout );


  try {
   //perform the work (i.e., the business logic)
   //the application should use the id to
   //identify the work for later confirm/cancel
   Connection conn = PaymentDbUtils.getConnection();
   Statement s = conn.createStatement();
   s.executeUpdate ( "insert into PAYMENTS values ( " +
   " '" + id + "' , '" + cardNo + "' ," + amount + "," +
   " 'PENDING' )" );

   conn.close();
   //mark the work as completed in the system
   //to trigger the cancel/confirm callbacks
   TCCmgr.completed ( id );

  } catch ( Exception e ) {
   e.printStackTrace();
   //on any exception: mark the work as failed
   //so cancel is the only possible callback
   TCCmgr.failed ( id );
  }
 }


 public void confirm ( String id )
 throws HeurRollbackException, TCCException
 {
```

```
  try {
   Connection conn = PaymentDbUtils.getConnection();
   Statement s = conn.createStatement();
   s.executeUpdate (
"update PAYMENTS set status = 'CONFIRMED' where key = '" + id +"'" );
   conn.close();
  } catch ( SQLException e ) {
   e.printStackTrace();
   //optional: throw TCCException to retry confirmation
   throw new TCCException();
  }

 }

 public void cancel ( String id )
 throws HeurCommitException, TCCException
 {
  try {
   Connection conn = PaymentDbUtils.getConnection();
   Statement s = conn.createStatement();
   //set status to canceled; alternatively, we could also
   //choose to delete the payment row instead; this depends
   //on the business-specific model of cancelation
   s.executeUpdate (
"update PAYMENTS set status = 'CANCELED' where key = '" + id +"'" );
   conn.close();
  } catch ( SQLException e ) {
   e.printStackTrace();
   //optional: throw TCCException to retry cancelation
   throw new TCCException();
  }

 }

 public boolean recover ( String id )
 {
  //check if the id is one of our work identifiers
        //return true if so


 }

}
```

The following state diagram illustrates the relevant TCCService states and their transitions.

> The invocation of `cancel` or `confirm` are guaranteed to happen only after the try-phase is over (i.e., after the application calls either `completed` or `failed`).

Figure 2.6. TCCService State Diagram



The figure above shows the relevant states for TCC™ instances, and the possible transitions. For clarity, the transitions initiated by the transaction system are shown in black arrows, whereas the service-generated transitions are shown in red. More precisely: the black arrows represent method calls that the TCCService instance will receive upon the transaction service's initiative. The relevant states are the following:

- *Trying*: the TCC™ instance is executing the tentative service implementation. This state is entered after the application calls `register` on the `TCCServiceManager`. The transaction service guarantees that there will be no interleaving calls of `cancel` or `confirm`. In this state, the application's thread is associated with the activity (this allows easy propagation via the JAX-RPC handlers).
- *Suspended*: this state is reached when the application's thread is no longer associated with the transaction (activity). This state is an intermediate state to change the thread association of the application. It is up to the application to reach this state, by calling `suspend` on the `TCCServiceManager`. More information on this technique will follow later.
- *TryFailed*: this state is reached after the *Trying* state fails (indicated by the application calling the `failed` method of the `TCCServiceManager`). In this state, the transaction service will later call `cancel`.
- *Tentative*: this state is reached only after the *Trying* phase finishes without errors (as indicating by a call to `completed` on the TCCServiceManager). This is the only state in which `confirm` can ever be called, but only if the global transaction (activity) commits. In case of global transaction rollback, `cancel` will be called.
- *Confirming*: the instance is in this state when `confirm` has been called and is still executing. If a `TCCException` happens then the TCC™ instance returns to the *Tentative* state, and `confirm` will be retried later.

- *Canceling*: the instance is in this state when `cancel` has been called and is still executing. If a `TCCException` happens then the TCC™ instance returns to the *Tentative* state, and `cancel` will be retried later.

- *Confirmed*: if the confirmation is successful (no exceptions) then this state is reached. This state is also reached when `cancel` finds the business logic to be heuristically confirmed by intermediate database administration interventions.

- *Canceled*: if the cancelation is successful (no exceptions) then this state is reached. This state is also reached when `confirm` finds the business logic to be heuristically canceled by intermediate database administration interventions.

> What about the `recover` method? Actually, this method is only called for recovered TCC™ instances, and can only be followed by either `cancel` or `confirm`. The only thing that `recover` should do is ensure that `cancel` and/or `confirm` can do their work, be re-acquiring any resources if needed. Most of the time, this method can be left empty. Note that TCC transactions are recoverable as soon as they enter the *TRY* phase. This guarantees that pending work is correctly recovered and terminated after a crash or restart.

2.3.1.3. Executing TCC™ Services

The (tentative, try-phase) execution of TCC™ services is started by calling `register` on the `com.atomikos.icatch.TCC.UserTCCServiceManager`, the default implementation of `com.atomikos.TCC.TCCServiceManager`. This method returns a `java.lang.String` identifier for the work; this identifier will be used as an argument for termination.

> The `TCCServiceManager` class should be initialized with the appropriate `TCCService` implementation (provided by the application) before it can be used. This is done via an invocation of the `registerForRecovery` method. For recovery, it is highly recommended that this initialization be done as soon as possible after startup of the application.

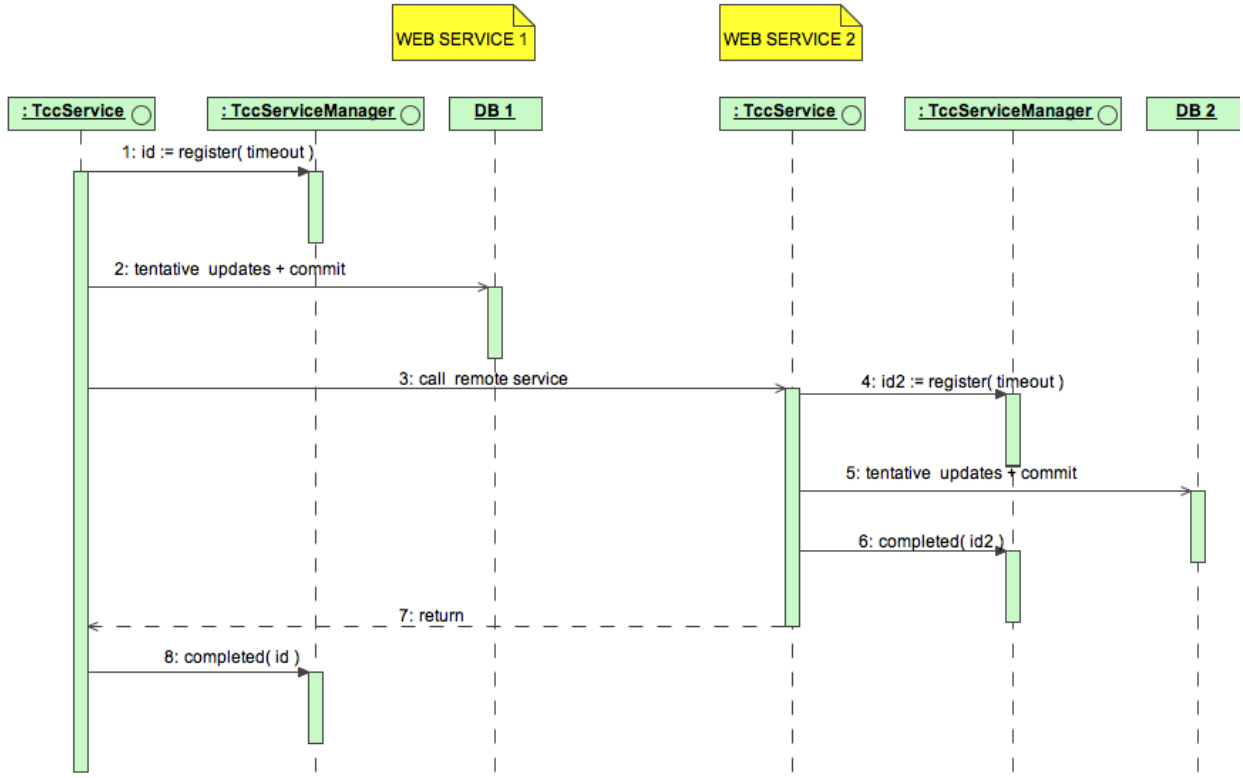
Once registered, it is up to the application to do any work that is subject to transactional termination. For convenience, the executing thread is associated with the activity (transaction). For remote calls, this allows the activity to be propagated with the built-in JAXRPC handlers. In case the application wants to switch threads for the activity, the `suspend` and `resume` methods can be used to manage the thread associations.

As soon as the *try-phase* is over, the application should call either `completed` (if successful) or `failed` (if erroneous) with the correlation identifier returned by `register`. For root activities, the entire, possibly distributed activity is then terminated in a consistent manner by the underlying transaction service. In other words, the system calls either `cancel` or `confirm` on each participating `TCCService` implementation in a consistent way.

Example 2.3. Distributed TCC™ Execution

The following illustration shows a sample distributed TCC™ execution between two transactional services.

Figure 2.7. Distributed TCC™ execution between two transactional services



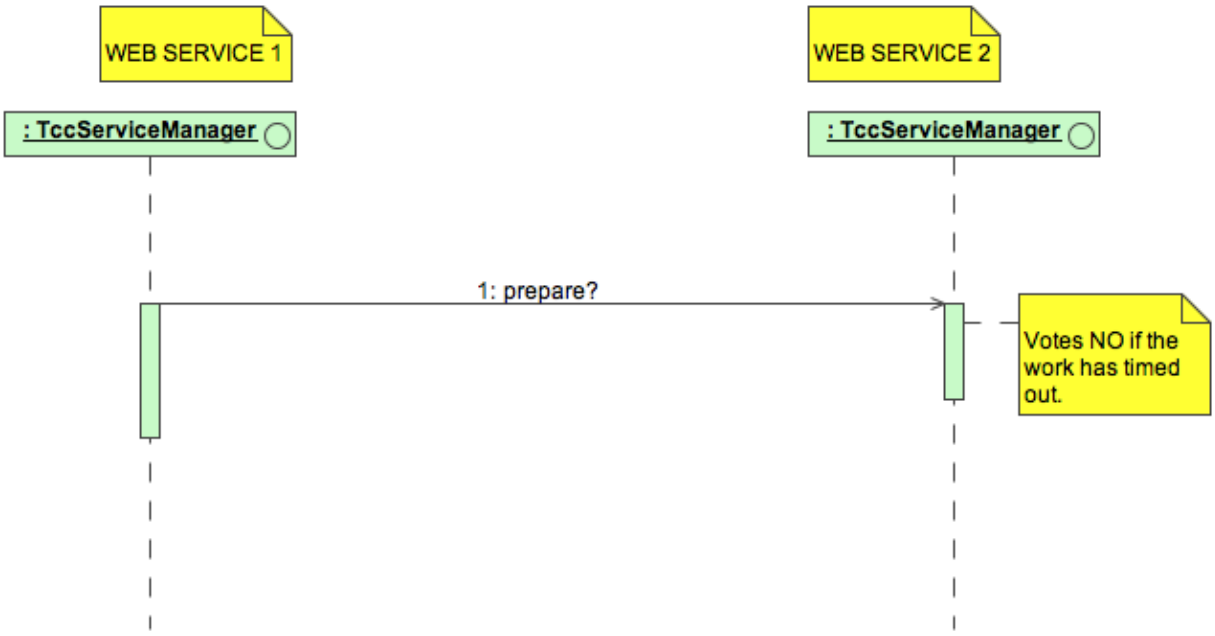
The following steps are displayed:

1. The application in service 1 registers with the transaction service. Because there is no pre-existing activity yet, a new (root) activity is created for this service (not shown). The identifier *id* is returned for this activity. Service 1 can now perform the *try-phase* of its work.
2. First, service 1 chooses to update its local database and commits. The effects become visible to other, concurrent applications but are still subject to two-phase commit termination of the TCC™ services. In order to be able to correlate the database updates with the activity, service 1 is likely to use the work identifier *id* as a (primary) key for its database updates.
3. As part of its *try-phase*, service 1 then chooses to call service 2. The activity is propagated by the handlers. If done asynchronously, then service 1 may choose to use the work identifier *id* as a correlation identifier for the communication.
4. The handlers at service 2 import the activity, and the call is forwarded to the application (this is how handlers work; the application doesn't have to worry about this). The application at service 2 wants to do work that is subject to TCC™ termination, so it first registers with the (local) transaction service. The application receives an identifier *id2* to refer to the (local part of the) work.
5. Service 2 can now update its local database to reflect the tentative state of its work. It is likely that *id2* be used as a database identifier to correlate with later cancelation or confirmation. The database updates are committed immediately, in a local database transaction.
6. Service 2 is satisfied with its work and signals to the transaction service that it is ready to confirm when asked to. This is done by calling `completed` with the local work identifier *id2*.
7. The result is returned to service 1 (if asynchronous, then the correlation identifier *id* of service 1 can be used).
8. Service 1 is now also satisfied with its work, and terminates its tentative phase by calling `completed` with its own work identifier *id*. Since service 1 is executing as the root activity, this will trigger completion by the transaction service (shown in the next examples).

Example 2.4. Distributed TCC™ Prepare

After the root activity is completed in the previous example, the transaction service will ensure consistent termination with cancel/confirm at all participant services. TCC™ activities can be distributed across nodes and can take a long time to complete. Consequently, parts of the work can time out while waiting for confirmation. To minimize problems with timeouts, the transaction service will do a prepare-phase (as in two-phase commit) behind the scenes. This is a purely technical step without application-level requirements.

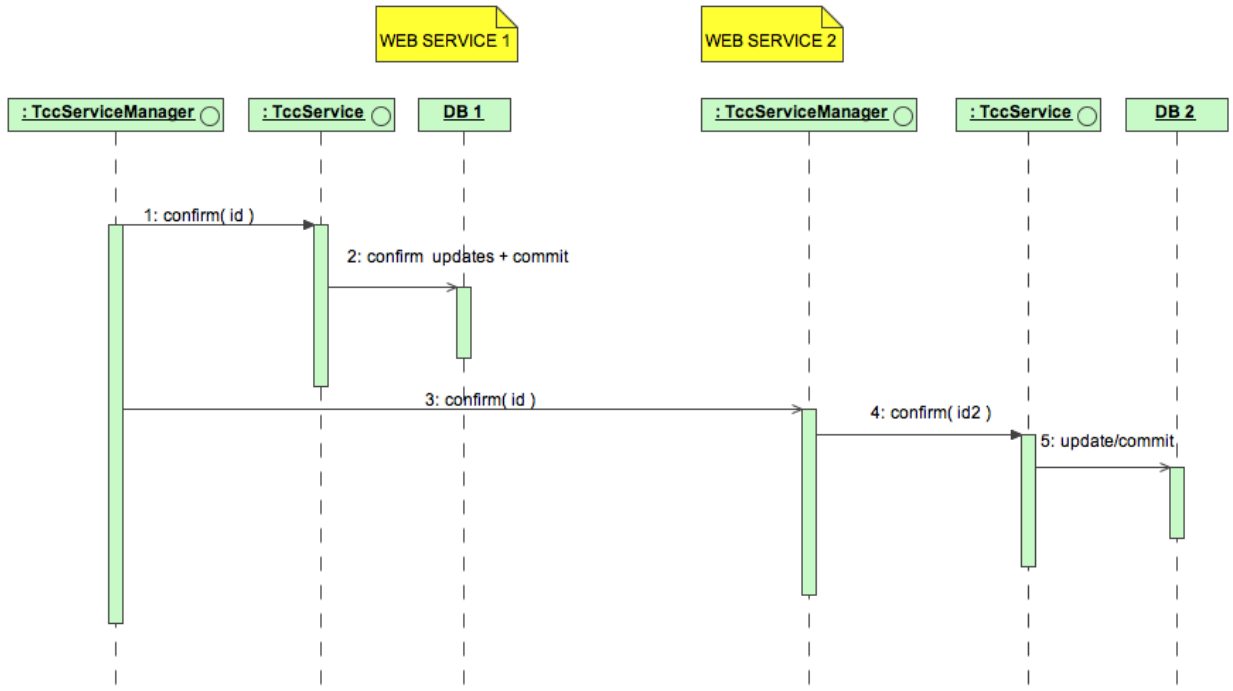
Figure 2.8. Distributed TCC™ prepare phase



Example 2.5. Distributed TCC™ Confirmation

Assuming that the prepare phase succeeded (no timeouts) then confirmation will happen as shown next. The system uses the registration identifiers of the work to call the application-level `confirm` implementations. Each service can use its respective identifier to confirm its changes in the database. This happens in a separate, local database transaction committed immediately.

Figure 2.9. Distributed TCC™ confirmation phase

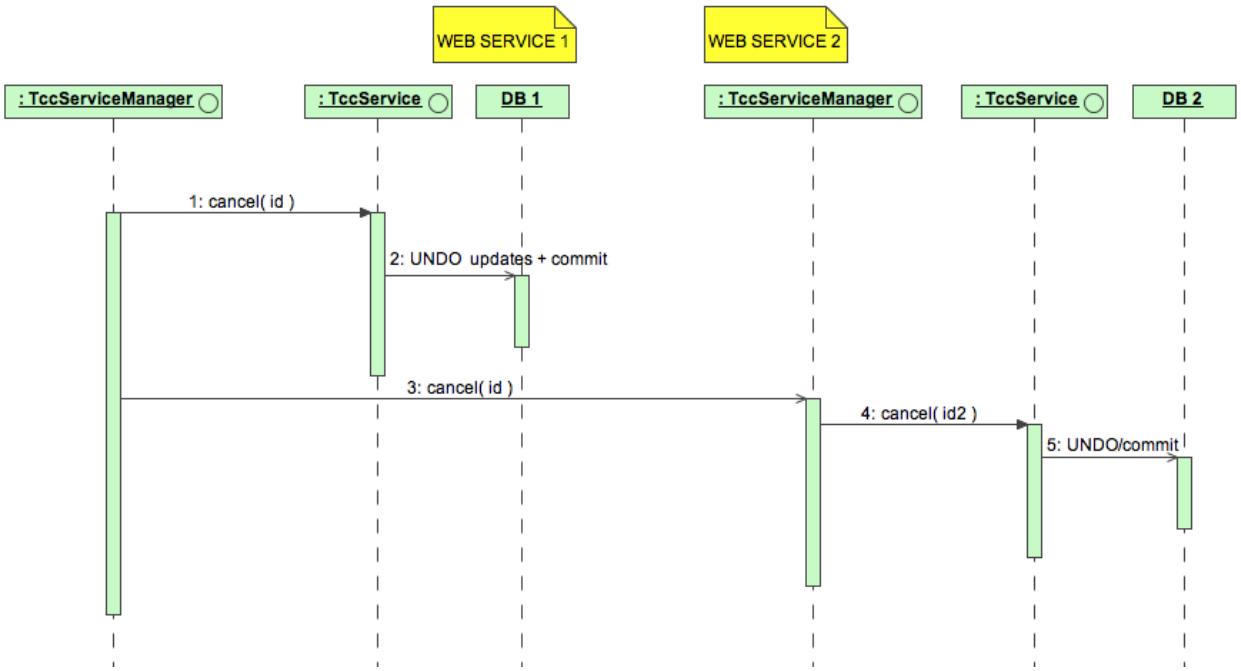


Note the local nature of confirmation: because the transaction service calls the remote parties, each TCCService implementation only has to confirm its own work locally. Also note that the *local identifier* is used to confirm the work at each site.

Example 2.6. Distributed TCC™ Cancelation

In case of one or more timeouts during prepare, cancelation will be done everywhere. This is similar to confirmation except that now `cancel` is called at each participating service.

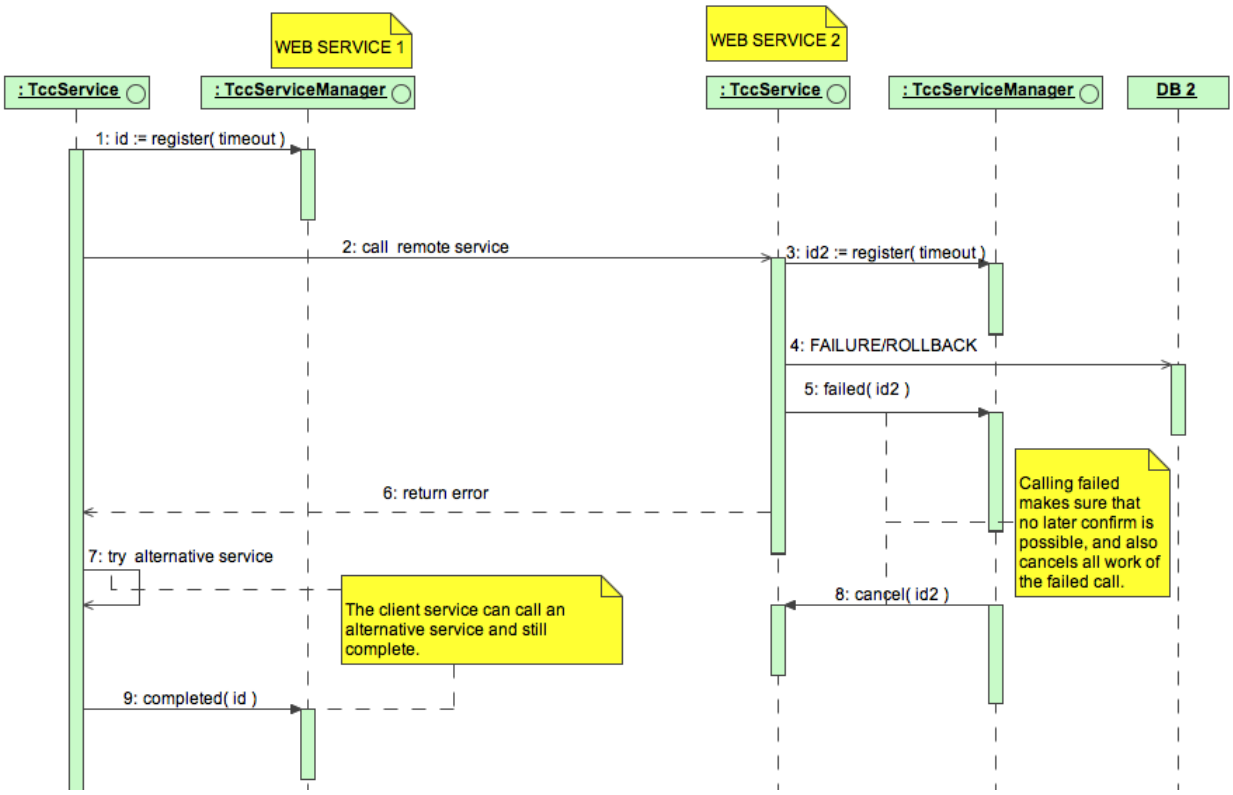
Figure 2.10. Distributed TCC™ cancelation phase



Example 2.7. Distributed TCC™ Failure

A *failure* is an error in the tentative logic (in the *try-phase*). A service should call *failed* (with its work identifier) whenever it encounters an error from which it can't recover. This will make sure that the transaction service forbids confirmation of (the relevant parts of) the work and calls cancel instead. This is shown next.

Figure 2.11. Distributed TCC™ failure



In this example, service 2 encounters a fatal error and calls `failed` to indicate that its part of the work (and any transactional calls that service 2 may have made) should be canceled.

Note that service 1 is still allowed to try an alternative service and complete. This means that failed remote calls don't have to lead to global failure of the entire activity.

2.3.1.4. TCC™ and Subtransactions

When `register` is called without an existing transaction (activity), the system wil create a *root* activity. On the other hand, if an activity already exists (for instance, an imported one) then a *subtransaction (or subactivity)* will be created instead.

Calling `completed` has a different effect for root activities versus subactivities:

- For root activities, calling `completed` will trigger the distributed confirmation process and lead to the consistent invocation of either `confirm` or `cancel` for each participating service.
- For subactivities (e.g., imported activities), `completed` will merely mark the activity is `tentative`; it is up to the root to trigger the final confirmation or cancelation.

2.3.1.5. Failures During Confirmation or Cancelation

A well-known problem with compensation-based approaches is what to do when compensation fails (the same can be said of confirmation). As the TCC™ state diagram shows, there are two possible ways of failure:

- A `TCCException` is thrown. In this case the transaction service will retry the `cancel` (or `confirm`) a number of times. If this eventually works then there is no problem. Otherwise, the transaction service will eventually give up and make the transaction fail with a *heuristic hazard* error. This means that the transaction information stays in the logs and is available for manual intervention. One of the patent-pending features of ExtremeTransactions™ is the detailed application-level information available in this case.
- A heuristic exception is thrown. This signals that an intermediate administrative intervention has already terminated the TCC™ process in an incompatible way. This is a fatal error condition, because at least part of the global transaction did not terminate the way it was supposed to. This again leads to a heuristic error for the overall transaction, and the information will be available in the logs for manual resolution.

2.3.2. Compatible Protocols

TCC™ service implementations are compatible with the following protocols:

- The Atomikos™ web service transaction protocol.
- The Atomikos™ binary *RMI-IIOP* protocol.

Any transaction imported via one of these protocols will automatically be available for you TCC™ service (i.e., confirm or cancel of your TCC™ service will be linked with the two-phase commit of the overall transaction).

Chapter 3. Configuring ExtremeTransactions™ Behaviour

The configuration of ExtremeTransactions™ is very similar to TransactionsEssentials™, with some extra parameters and values to set. This chapter discusses each parameter in turn. These parameters relate to the *transactions.properties* file in your classpath.

3.1. Enabling ExtremeTransactions™

The parameter `com.atomikos.icatch.service` must be set to `com.atomikos.icatch.trmi.UserTransactionServiceFactory` in order to make sure that ExtremeTransactions™ is enabled.

3.2. RMI-JRMP Configuration

If you plan to use *RMI-JRMP* as the protocol for calling your services, then you should set the following parameters.

- `com.atomikos.icatch.rmi_export_class`: set this to value `UnicastRemoteObject`.
- `java.naming.factory.initial`: set this to the JNDI initial context factory of your *JNDI* service. ExtremeTransactions™ needs a *JNDI* registry to store references to remote transaction objects it creates and exposes during its operation. The recommended value is `com.sun.jndi.rmi.registry.RegistryContextFactory` (the *JNDI* included in the Java installation).
- `java.naming.provider.url`: set this to the corresponding *JNDI* provider address. If you set the recommended factory above, then the URL will most likely be `rmi://localhost:1099` - although the port 1099 may be different for your system (depending on your configuration).

> The *RMI registry* needs to be started separately, *before* you launch your service. Otherwise, startup will fail. To start it, just type `rmiregistry` in a command shell.

3.3. RMI-IIOP Configuration

If you plan to use *RMI-IIOP* as the protocol for calling your services, then you should set the following parameters.

- `com.atomikos.icatch.rmi_export_class`: set this to value `PortableRemoteObject`.
- `java.naming.factory.initial`: set this to the JNDI initial context factory of your *JNDI* service. ExtremeTransactions™ needs a *JNDI* registry to store references to remote transaction objects it creates and exposes during its operation. For *IIOP*, the recommended value is `com.sun.jndi.cosnaming.CNCtxFactory` (the CORBA *JNDI* included in the Java installation).
- `java.naming.provider.url`: set this to the corresponding *JNDI* provider address. If you set the recommended factory for *IIOP* above, then the URL will most likely be `iiop://localhost:1050` - although the port 1050 may be different for your system (depending on your configuration).

The *CORBA naming service* (used as *JNDI* registry in this case) needs to be started separately, *before* you launch your service. Otherwise, startup will fail. To start it, just type `tnameserv -ORBInitialPort 1050` in a command shell.

3.4. Disabling RMI

If you don't need *RMI* services (because, for instance, you want to use *SOAP*) then you can avoid the overhead of having to start the registries by setting `com.atomikos.icatch.rmi_export_class` to `none`.

3.5. Setting Client Trust Preferences

If you trust client services to control the administrative aspect of your transactions then set `com.atomikos.icatch.trust_client_tm` to `true`. This will simplify administration of problematic in-flight transactions at the expense of giving up some control to the client (transaction) service.

Another aspect of client trust is whether you want remote (non-service) clients to be able to start and commit transactions. This can be indicated by setting `com.atomikos.icatch.client_demarcation` to `true`.

3.6. Enabling Web Service Transactions

Web service transactions can be enabled by doing the following:

- Set the initialization parameter `com.atomikos.icatch.soap_commit_protocols` to `atomikos`. This will trigger startup and export of the SOAP two-phase commit endpoints in your web service VM.
- Set the initialization paramater `com.atomikos.icatch.soap_host_address` to the DNS hostname of the machine where your web service is running. This setting is optional in case the default IP address guessed by ExtremeTransactions™ does not work.
- Set the initialization parameter `com.atomikos.icatch.soap_port` to the port number where you want two-phase commit to take place. This is optional and only needed if the default guessed by ExtremeTransactions™ does not work.
- On the server (receiving) side: add an instance of `com.atomikos.icatch.jaxws.atomikos.ImportingTransactionHandler` to the handler chain of the web service.
- On the client side, add an instance of `com.atomikos.icatch.jaxws.atomikos.ExportingTransactionHandler` to the endpoint of the service you are going to call.

Web service transactions are supported according to the Atomikos (native) protocol for two-phase commit across web services (http://www.atomikos.com/schemas/2005/10/transactions/atomikos.wsdl). Transactions can span multiple web service invocations provided that all nodes are configured as outlined here.

Example 3.1. Configuring the ImportingTransactionHandler on the receiving side

Adding a handler to the server (receiver) side is done like this in `JAX-WS`:

```
package com.atomikos.demo.jaxws.server;

import java.util.List;

import javax.xml.ws.Endpoint;
import javax.xml.ws.handler.Handler;

import com.atomikos.demo.jaxws.impl.WebShop;
import com.atomikos.icatch.jta.UserTransactionManager;
import com.atomikos.icatch.jaxws.atomikos.ImportingTransactionHandler;

public class Server {

 public static void main(String[] args) throws Exception {
  UserTransactionManager utm = new UserTransactionManager();
  utm.init();

  Endpoint endpoint = Endpoint.create(new WebShop());
  List<Handler> chain = endpoint.getBinding().getHandlerChain();

  //incoming requests are intercepted by an
  //Atomikos handler to deal with transaction aspects
  ImportingTransactionHandler handler = new ImportingTransactionHandler();
  //depending on the importPreference,
  //a new tx may be created for the incoming call
  //just like in regular JEE
  handler.setImportPreference("Required");
  //new transactions time out when? (milliseconds)
  handler.setNewTransactionTimeout ( 10000 );
  //should newly created transactions be JTA-like or TCC-like?
  handler.setJtaCompatible(true);
  //when a heuristic timeout happens, commit or rollback?
  handler.setCommitOnHeuristicTimeout(true);

  chain.add(handler);

  endpoint.getBinding().setHandlerChain(chain);
  endpoint.publish("http://0.0.0.0:8888/shop");

  System.out.println("server started");

  while (true) {
   Thread.sleep(1000);
   if (1 == 2)
    break;
  }

  utm.close();
 }

}
```

This handler class takes care of extracting any transaction context present in the incoming calls, and registers for the final (global) transaction outcome as controlled by the client. For the complete code, please see the JAX-WS example included in the download.

Example 3.2. Configuring the ExportingTransactionHandler on the client side

Adding a handler to the client (sender) side is done like this in JAX-WS:

```
import java.util.List;
import javax.xml.ws.Binding;
import javax.xml.ws.BindingProvider;
import javax.xml.ws.handler.Handler;

import com.atomikos.demo.jaxws.impl.WebShop;
import com.atomikos.demo.jaxws.impl.WebShopService;
import com.atomikos.icatch.jaxws.atomikos.ExportingTransactionHandler;


public class Client {

    public static void main ( String[] args ) throws Exception {

        WebShop shop = new WebShopService().getWebShopPort();
        Binding binding = ((BindingProvider)shop).getBinding();
        List<Handler> handlerList = binding.getHandlerChain();
        handlerList.add ( new ExportingTransactionHandler() );
        binding.setHandlerChain ( handlerList );

        // now, any calls made to the shop will be done in
        // the existing transaction context, if any...

    }

}
```

For the complete code, please see the JAX-WS example included in the download.

Chapter 4. Mistaken Alternatives for ExtremeTransactions™

You may wonder if there is really a need for ExtremeTransactions™ in your case. It really depends on what you want, but *if you have a service whose effects may time out or need to be canceled on request then you really need this technology*. This chapter discusses some perceived alternatives and shows why they are insufficient.

4.1. Exposing Cancel Operations as a Business Service

A technique that is often suggested is exposing the cancel (or confirm) operations as separate business services, to be invoked by a client who wants to cancel (confirm) a previous service invocation. This technique is inappropriate for the following reasons:

- It requires the client to explicitly construct the right cancelation call. This may be simple for trivial cases, but if you have complex workflows then this quickly becomes error-prone and it doesn't scale. Our approach allows clients to concentrate on the positive logic; all the rest is transparently offered by the transaction service. This reduces client complexity by at least 66 percent, without even counting all the 'transaction management' code that you don't have to write and maintain yourself.
- It implies that the client must persistently track each service's state: the client must be able to cancel pending services even after restarts or crashes. Otherwise, the number of 'pending' services will soon rise exponentially, and so will all related costs. In our approach, all of this is handled by the transaction service.
- It implies that you trust the client(s) to keep your own business logic consistent. This is inappropriate in a loosely-coupled and federated services world.
- Time-out is difficult to handle, because you don't have the necessary context (and even if you do then it is hard to deal with interleaving of timeout and incoming confirmations, leading to many anomalous transactions with a high administration cost).
- Issueing and receiving cancel notifications at the application-level may be problematic: for some communication means (such as JMS™), the order of messages is not guaranteed. This means that *in the do-it-yourself approach cancel messages may arrive before the request they are meant to cancel*. This means that cancel events are effectively lost. ExtremeTransactions™ avoid this problem.

The only case where this technique really works is when you don't care about time-out and consistency. This applies only to businesses where cancelation is 'just another business transaction' (like buying and later selling stock). For businesses based on the 'reservation' concept, this technique is entirely inadequate.

4.2. Using Reliable Messaging

Reliable messaging by itself is a great technology, but not adequate for reservation-based business models. The notion of cancelation is not really compatible with the usual 'fire-and-forget' mechanism offered by these messaging platforms. Moreover, reliable messaging doesn't solve the problem of timed-out (tentative, unconfirmed) business transactions. We see reliable messaging as complementary to ExtremeTransactions™ technology.

Appendix A. Glossary

- *ACID*: Atomic, Consistent, Isolated and Durable - the classical properties that are enforced on transactions.
- *CORBA*: Common Object Request Broker Architecture - a standard architecture for building distributed systems. Outdated with the advent of Java, SOA and web services (*SOAP*).
- *GRID*: an architecture for massive scaling, where multiple equivalent instances of a service or application are deployed on a number of independent hardware servers.
- *IIOP*: Internet Inter-Orb Protocol - the communication protocol for RPC in CORBA. Also used in JEE.
- *JAX-RPC*: an outdated API for developing web services in Java.
- *JAX-WS*: the current API for developing web services in Java.
- *JDBC*: Java Database Connectivity - the standardized API for accessing relational databases from within Java programs.
- *JDK*: Java Development Kit - the set of development tools required to build Java applications.
- *J(2)EE*: Java, Enterprise Edition - the Java libraries and extensions (to the JDK) required to support enterprise applications.
- *JMS*: Java Message Service - the standardized API for accessing messaging back-ends from within Java programs.
- *JTA*: Java Transaction API - the standardized API for demarcating ACID transactions from within Java programs.
- *JRMP*: Java Remote Method Protocol - the original protocol used for RMI in Java (later on, IIOP was added as well).
- *RMI*: Remote Method Invocation - a soft of "RPC for distributed objects", one of the core technologies in J(2)EE.
- *RPC*: Remote Procedure Call - a way of calling other processes transparently (by hiding the network from the caller).
- *SOA*: Service-Oriented Architecture - a way of developing software as a set of reusable services (available on the network).
- *SOAP*: Simple Object Access Protocol - a standardized way of encoding RPC-like calls as XML (used for web services).
- *TCC*: Try-Cancel/Confirm - a new way of providing transactional services, provided by Atomikos.
- *VM*: Virtual Machine - the runtime environment of a Java application.
- *web services*: services accessible on the network via SOAP.
- *XA*: Extended Architecture - a standardized API for co-ordinating ACID transactions across backend systems.
- *XTP*: Extreme Transaction Processing - an architecture for processing high volumes of mission-critical transactions on commodity infrastructure. As explained by numerous Gartner™ reports, J(2)EE application servers are not suited for this; something new is needed like ExtremeTransactions™.

Appendix B. More Information

- *TCC: Try-Confirm/Cancel Transactions for Web Services*

  Available online at http://www.atomikos.org/forums/viewtopic.php?t=97

- *The WS-Transaction Standardization Committee*

  See http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ws-tx (information about WS-Coordination, WS-AtomicTransaction and WS-BusinessActivity)

- *JAX-RPC*: http://java.sun.com/webservices/jaxrpc

- *JAX-WS*: https://jax-ws.dev.java.net

- *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

- *XA Specification*: Published by the Open Group (http://www.opengroup.org), available online via the website.