OBJECTSPACE VOYAGER TRANSACTION SERVICE

Voyager Transaction Service Technical Overview

Transaction management is an essential tool for building reliable systems. Transactions provide the ability to execute a sequence of operations in an atomic way, where the entire sequence succeeds or fails. This strategy maintains overall system integrity, moving the system from one valid state to another. In a distributed architecture, the state of the system spans multiple processors and resources, making the task of maintaining an overall system state more difficult. This paper describes ObjectSpace's Voyager Transaction Service (VTS), an easy-to-use solution to this problem.

OBJECTSPACE

© 1998 ObjectSpace, Inc. • All rights reserved • January 1998

Contents

Overview **1**
User API 1
Transaction Management 1
Resource Management 1
Features 2

Transaction Use **3**
Implicit Use Mechanism 3
Explicit Use Mechanism 4
Other VTS Uses 5

Database Integration **6**
JDBC Resource Example 6

VTS Versus JTS **10**

Overview

The Voyager Transaction Service (VTS) is built on the ObjectSpace Voyager™ Core Technology (Voyager). VTS provides an easy-to-use API for defining transaction boundaries. It also provides an implementation for managing transactions and maintaining their atomicity, consistency, isolation, and durability (ACID) properties.

VTS is compatible with industry standard transaction services like CORBA's OTS and X/Open DTP and can interoperate with their implementations. VTS consists of three areas:

- User API
- Transaction management
- Resource management

User API

The VTS user API exposes transaction semantics to domain applications, thus providing maximum design flexibility for domain applications while adding minimal coding overhead. This is accomplished through a static *Transaction* class with all the necessary behavior to start, suspend, resume, and conclude a transaction.

Transaction Management

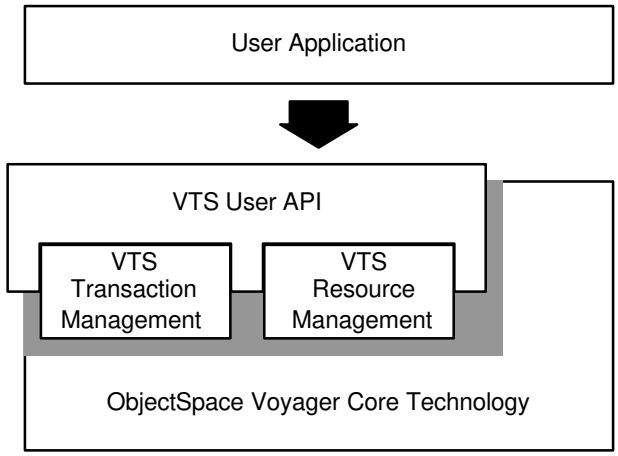
The VTS transaction management area comprises three layers:

- A transaction management framework that facilitates integration with third-party transaction managers.
- An in-process transaction manager that provides maximum performance.
- A stand-alone transaction manager to manage transactions from a dedicated transaction management process.

These layers provide Voyager applications with a wide range of deployment options that help optimize performance and improve scalability.

Resource Management

The VTS resource management area provides a framework for defining, creating, and registering resources with Voyager transactions. These resources act as a bridge between VTS and persistent storage systems (for example, Oracle, Sybase, and Objectivity). Each resource represents a given database's native transaction and can perform rollbacks and two-phase commits. The design of VTS resource management is compatible with the X/Open XA model.



Although VTS does not provide any special hooks for monitoring or controlling in-transaction remote method invocations, this functionality can be accomplished by using the Voyager event mechanism.

Features

The production release of VTS includes the following features:

- An easy-to-use API.
- Compatibility with industry standard transaction services, like CORBA's OTS and X/Open DTP.
- Support for both flat and nested transactions.
- A framework to integrate with third-party transaction managers.
- Two transaction manager implementations:
  - An in-process manager to maximize performance.
  - A stand-alone manager to support a wider range of system configurations.
- Support for multiple transaction managers per transaction.
- A framework to integrate third-party database transactions as resources.
- A resource manager implementation to `VoyagerDb`.
- An improved `VoyagerDb` to support nested transactions.
- Support for multiple resources per transaction.
- A VTS-JTS integration package to use JTS (JavaSoft's Java™ Transaction Services) implementations.

Other features being considered for this release or future releases are:

- Various transaction manager implementations that integrate VTS with Microsoft's MTS and BEA's Tuxedo.
- Various resource manager implementations that integrate VTS with OODBs and RDBMS.
- A resource manager implementation that integrates VTS with X/Open XA resource.

Transaction Use

VTS intentionally exposes the transaction semantics to domain applications to provide maximum flexibility in the design of these applications. This approach is also based on the belief that defining transaction boundaries is part of an application domain design, and not a deployment/configuration activity. However, the actual transaction management is hidden from the application and can be carried out by many or no implementations.

VTS provides an easy-to-use API to define transaction boundaries. The VTS API supports two use mechanisms: one for implicit use and one for explicit use.

Implicit Use Mechanism

The implicit use mechanism is most common. Upon executing Transaction.begin(), VTS starts a new transaction and associates it with the currently running Java Thread. If the thread is already associated with a transaction, VTS starts a subtransaction. If a thread needs to share the transaction of another thread, the Transaction.begin(Thread) operation can be used.

The following example demonstrates the use of the VTS implicit API. The program deposits $500 in Bob's bank account at CitiBank and updates Bob's checkbook for CitiBank to reflect the deposit. These two operations happen in the same transaction to ensure that they both either succeed or fail, keeping Bob's bank account and checkbook in sync.

```
try
   {
   //begin a new transaction
   Transaction.begin();

   //get bob's account at citibank:8000
   BankAccount myAccount = (BankAccount)
      VObject.forObjectAt( "citibank:8000/bob" );

   //get bob's citibank's checkbook
   CheckBook myCheckbook = (Checkbook)
      VObject.forObjectAt( "localhost/bob_citibank_checkbook" );

   //make the deposit
   myAccount.deposit( 500.0 );
   myCheckbook.deposit( 500.0 );

   //commit the transaction
   Transaction.commit();
   }
catch( Exception ex )
```

```
  {
  //rollback the transaction
  Transaction.rollback();

  //more exception handling code
  }
```

Explicit Use Mechanism

The explicit use mechanism of VTS gives the user application more control over what to associate the transaction with. With this control comes the added responsibility of maintaining the transaction and propagating it to other remote or local objects. This use of VTS is less common.

The following example repeats the previous one, but uses the explicit use mechanism for defining transactions.

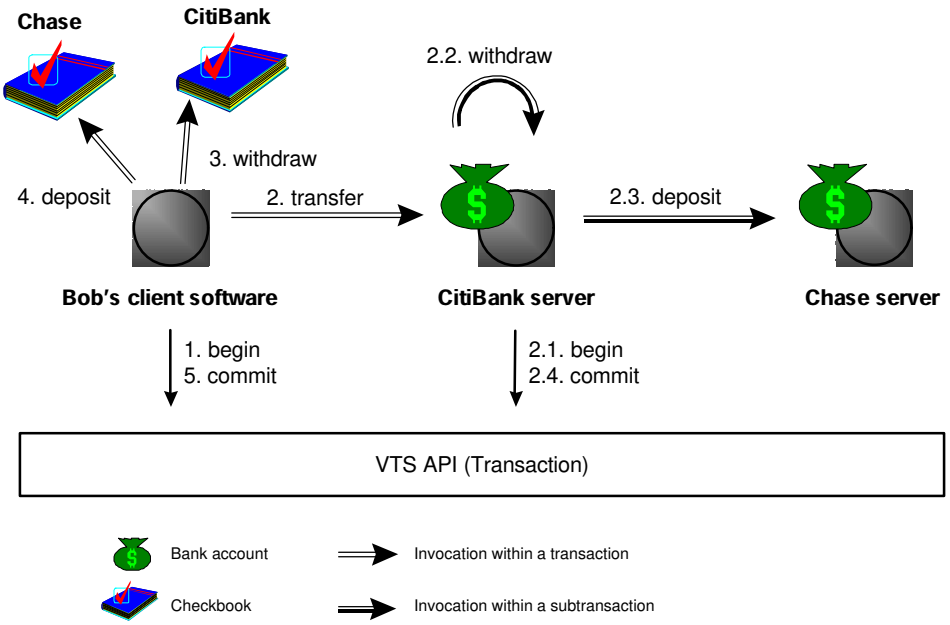
```
try
  {
  //create a new transaction
  TransactionControl transaction = Transaction.newTransaction();

  //begin a new transaction
  transaction.begin();

  //get bob's account at citibank:8000
  BankAccount myAccount = (BankAccount)
    VObject.forObjectAt( "citibank:8000/bob" );

  //get bob's citibank's checkbook
  CheckBook myCheckbook = (Checkbook)
    VObject.forObjectAt( "localhost/bob_citibank_checkbook" );

  //make the deposit
  myAccount.deposit( 500.0, transaction );
  myCheckbook.deposit( 500.0, transaction );

  //commit the transaction
  transaction.commit();
  }
catch( Exception ex )
  {
  //rollback the transaction
  transaction.rollback();

  //more exception handling code
  }
```

Note: As part of VTS development, the Voyager team is evaluating adding the concept of thread context to Voyager and exposing it to user applications. The thread context would provide an alternative to parameter passing for propagating a TransactionControl, both locally and remotely.

Other VTS Uses

VTS supports multiple patterns for exposing transaction semantics. For example, an application might expose the transaction API in a client, in a server, in both (nested transactions), or in neither.

To illustrate this concept, the previous example could be modified to transfer money instead of depositing it. Assuming that both the deposit and withdrawal operations are atomic by nature (that is, they do not need a VTS transaction), the following diagram depicts the modified example.



Bob's client software behaves the same way as in the previous example, wrapping both the account and checkbook operations with a transaction begin and end. The CitiBank server also wraps the withdrawal and deposit operations with a transaction begin and end. Because the execution thread is already in a transaction, the transaction begin operation creates a subtransaction. Although the Chase server does not communicate with the VTS API, its effect on the overall system state is still governed by VTS. In a single transaction, this example demonstrates exposing transaction semantics to the client, to the server, and to neither.

Database Integration

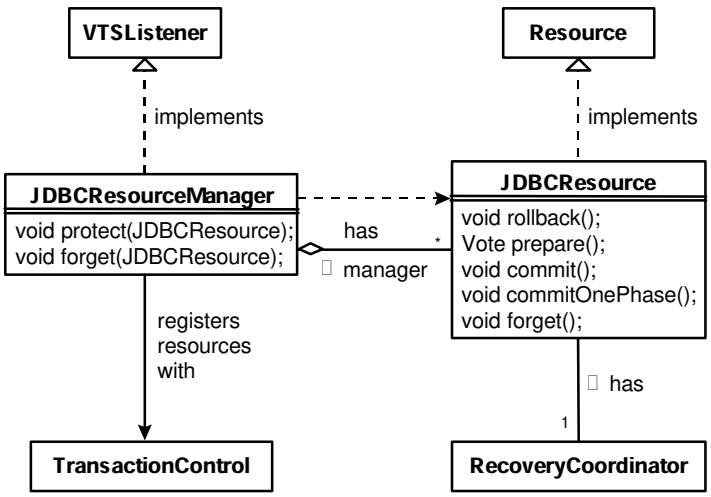
VTS provides a resource management framework for database integration. This framework consists of a resource interface and a set of behaviors on TransactionControl to register a resource. Implementing this framework is usually done by database vendors or system integrators. Voyager-based applications are not impacted by the implementation of this framework.

To integrate with a specific database, an application must include these tasks:

- Provide an implementation of the resource interface that can communicate with the database native transaction.
- Decide when to create and register a new resource.
- Manage active resources and restore them in case of failure.

JDBC Resource Example

The following diagram outlines a resource management implementation that works with databases that support the JDBC API and two-phase commits.



The resource implementation below instantiates a JDBC connection. It also forwards the transaction conclusion decision to the connection.

```
import com.objectspace.vts.*;
import java.sql.*;

class JDBCResource implements Resource
   {
   Connection connection;
   JDBCResourceManager manager;

   //constructors
   public JDBCResource( JDBResourceManager manager )
      throws SQLException
      {
      connection = DriverManager.getConnection( manager.databaseURL );
      connection.setAutoCommit( false );

      this.manager = manager;
      }

   //Resource behavior

 void rollback()
      {
      try
         {
         connection.rollback();
         }
      catch( SQLException ex )
         {
         //add exception handling code
         }

      forget();
      }

   Vote prepare()
      {
      try
         {
         //this is not a JDBC operation and it is
         //only used for demonstration purposes
         connection.prepareToCommit();

         manager.protect( this );
         return Vote.Commit;
         }
      catch( SQLException ex )
         {
         return Vote.Rollback;
         }
      }
```

```
void commit()
   {
   try
      {
      connection.commit();
      }
   catch( SQLException ex )
      {
      //add exception handling code
      }

   forget();
   }

void commitOnePhase() throws CommitFailedException
   {
   if( prepare() == Vote.Commit )
      commit();
   else
      throw new CommitFailedException();
   }

void forget()
   {
   connection.close();
   manager.forget( this );
   }
}
```

In the following example, a JDBC resource manager is implemented to detect when to instantiate and register a new JDBCResource. It is also responsible for protecting, in a persistent way, the resource in the period between prepare and commit.

```
import com.objectspace.vts.*;

class JDBCResourceManager implements VTSListener
   {
   String databaseURL;

   //constructors
   public JDBCResourceManager( String databaseURL )
      {
      this.databaseURL = databaseURL;
      recover(); //if necessary, recover from a previous crash
      Transaction.addVTSListener( this );
      }

   //VTSListener behavior
   void tranactionCoordinatorCreated( TransactionControl control, boolean newTransaction )
      {
      try
         {
         Resource resource = new JDBCResource( this );
         RecoverCoordinator recovery = control.register( resource );
         resource.recoveryCoordinator = recovery;
         }
      catch( Exception ex )
         {
         //add exception handling code
         }
      }

   //JDBCResourceManager behavior to protect resources
   void protect( JDBCResource resource )
      {
      //log resource information
      }

   void forget( JDBCResource resource )
      {
      //forget everything about this resource
      }

   void recover()
      {
      //go through persistent log and reestablish protected resources
      }
   }
```

VTS is compatible with the X/Open XA model for communicating with and managing resources.

VTS Versus JTS

VTS includes both a specification for defining transaction managers and resource managers, as well as a complete implementation to help design and deploy transaction-based systems. JTS from JavaSoft, on the other hand, is only a specification of a transaction service. It is a direct CORBA OTS conversion from IDL to Java. The production release of VTS, however, will include a separate package with VTS-JTS wrappers, so JTS transaction manager implementations and resource implementations can automatically run with VTS.

For additional technical information on ObjectSpace products and programs or for information on how to order and evaluate ObjectSpace technology, contact us today!

14850 Quorum Drive, Suite 500
Dallas TX 75240

972.726.4100
1.800.OBJECT.1
Fax: 972.715.9099

E-mail: sales@objectspace.com
Web: www.objectspace.com

Dallas • Austin • Chicago
San Francisco • Washington DC

Java is a trademark of Sun Microsystems.
ObjectSpace Voyager is a trademark of ObjectSpace, Inc.
All other trademarks are the property of their respective companies.