

Choosing Transaction Models for Enterprise Applications

Jim Tyhurst, Ph.D.
Tyhurst Technology Group LLC

This paper is being submitted as a position paper for the OOPSLA 2001 Workshop, "The Three-Tier Architecture Pattern Language". Details regarding the workshop are available online at <http://oopsla.acm.org> and <http://www.cs.wustl.edu/~pjain/ThreeTierPatterns>.

Contents

1. Overview
2. Forces that influence design decisions regarding transaction models
3. Organizing changes within a transaction
4. Infrastructure for recovering from transaction failures
5. Resolving conflicts between transactions
6. Updating views at transaction boundaries

1. Overview

Using a three-tier architecture for interactive enterprise applications requires architects and designers to think about transactions differently than with client/server applications. In order to achieve greater scalability, a thin client does not maintain a constant connection to the server. The client view is refreshed periodically, but for the most part, the end user is working on data outside of a transaction. At certain points, the user's actions cause the client to submit a transaction request to the server.

This paper summarizes a pattern language to help enterprise application architects to choose among alternative transaction models that might be used in situations where thin clients connect intermittently to an application server. We begin by describing some of the forces that influence design decisions with respect to transaction models. Then we summarize a number of design patterns for specifying the structure of interactions between the presentation layer, business logic layer, and data persistence layer in a three-tier architecture. A more detailed description of individual patterns is available at '<http://www.tyhurst.com/resources/patterns/tx>'.

2. Forces that influence design decisions regarding transaction models

The characteristics of a particular application will help determine the choice of transaction model. In this section, we identify a few factors to be considered by the application architect.

Failure to commit.

When trying to commit a transaction, the commit operation may fail due to conflicts with transactions from other clients. Typically, once a transaction has failed, it will never succeed in the future. Therefore, the client must abort (or *roll*

back) the failed transaction and start a new transaction with the desired changes. Either the application must provide the ability to replay the changes automatically or the user will have to re-enter the data manually.

Write-Write conflicts.

When you try to commit your transaction, the commit may fail as a result of conflicts with other transactions. A write-write conflict occurs when an object in your write set is also in the write set of a transaction that has been committed since your transaction began.

Probability of a commit failure.

Your motivation to design and implement a solution for a failure to commit depends on the probability of a failure occurring in your particular application. If the likelihood of a write-write conflict is very small, then it may be acceptable to simply display an error message to the user, abort the transaction, and force the user to begin the business transaction again. On the other hand, when losing the user's changes is an unacceptable solution, especially when it may occur relatively frequently, you will want to provide a solution to recover from a failure to commit.

Length of transaction.

There are several problems associated with long transactions:

- a. If locks are held, resources are tied up for the length of the transaction, which limits the ability of multiple users to access the data.
- b. If locks are not used, the risk of a commit failure increases with the length of the transaction, because there is a greater time period in which other users might commit a transaction that would cause a conflict.
- c. The architectures of most application servers encourage short transactions, because performance problems can occur when there is a large backlog of open transactions competing for resources.

Data currency.

The views displayed to a particular user must be updated as the domain model changes on the server (see patterns Push Model Changes and Pull Model Changes). However, these changes may only be integrated into the user's view at transaction boundaries.

Objects cannot be modified outside of a transaction.

Changes made to persistent objects while outside of a transaction cannot be committed. In order to change persistent objects, the application must begin a new transaction, make changes to the objects, and then commit the transaction.

Multiple units of work.

When mapping business transactions to server transactions, the application designer needs to determine each unit of work that must be recorded as a consistent whole. Dependencies between these units of work will influence the choice of transaction models.

3. Organizing changes within a transaction

Application servers typically provide some default transaction model. However, it may be useful to develop an extra layer of services for an application, in order to support

another model. These patterns will help the application architect to determine when it is worth the extra effort to extend a server's default transaction model.

Flat Transaction

Problem: How can a client application organize a set of domain model changes to be submitted in a transaction that is easy to implement?

Solution: Group all of the changes in a single transaction that has no internal structure.

Conversational Transaction

Problem: How can a client application organize a set of domain model changes to be submitted in a transaction where the user needs a relatively long time to specify the changes, but the server prefers short transactions?

Solution: Perform one transaction to read data, process the data offline, and then submit the changes in a new transaction.

Queued Transaction

Problem: How can a client application organize a set of domain model changes to be submitted in a transaction where there are many clients adding new independent data?

Solution: Specify the actions to be performed in a transaction and submit that specification into the queue of a transaction processor.

Nested Transaction

Problem: How can a client application organize a set of domain model changes to be submitted in a transaction where the changes are organized in hierarchical increments?

Solution: Provide a nested structure of transactions, so that each embedded transaction can be rolled back without affecting higher level transactions.

Distributed Transaction

Problem: How can a client application organize a set of domain model changes to be submitted in a transaction where the domain model is distributed across more than one server?

Solution: Provide a two-phase commit process, where each server verifies that it can commit its portion of the transaction before any server performs its portion of the commit.

4. Infrastructure for recovering from transaction failures

This section presents a variety of patterns for handling transaction failures, so that the transaction can be replayed automatically, rather than reporting a failure to the end user and expecting the user to resubmit the transaction.

Replay Changes

Problem: How can a client application preserve a user's changes after aborting a transaction?

Solution: The client keeps the changes independent from the view of the repository, so that when a view is refreshed, the changes may be replayed.

Transaction Specification

Problem: How can changes to domain model objects be saved independently from the domain model?

Solution: Save the sequence of messages that can be sent to cause the desired changes to occur.

Aspect Change Specification

Problem: How can a client application record a change to a domain model object independently from that object?

Solution: Specify the message that must be sent in order to accomplish the desired change. A message is completely specified by its receiver, method signature, and arguments.

Dirty Object Pool

Problem: How can a client application record a change to a domain model object independently from that object?

Solution: Keep a copy of the object in its changed state.

Locked Object

Problem: How can you reduce the risk of a commit failure when modifying an object?

Solution: Obtain a write lock on an object before modifying it.

Check Out Objects

Problem: How can you maintain a lock on an object for a long period of time?

Solution: Develop a lock table that holds persistent locks for objects that have been checked out. The application checks out a copy of the object, makes changes, and then checks in the changed copy.

5. Resolving conflicts between transactions

When faced with a commit failure, there are several ways in which to resolve the conflict that led to the failure.

First Commit Wins

Problem: How can you serialize a set of changes from overlapping business transactions that affect the same object when your primary concern is for users to see changes immediately as they occur?

Solution: Force the user to start a new business transaction starting from the latest server state whenever the user's changes cause a commit failure in a server transaction.

Merge Conflicting Updates

Problem: How can you serialize a set of changes from overlapping business transactions that affect the same object when your primary concern is that users should not have to reenter any data?

Solution: Selectively replay the user's changes in a new server transaction, prompting the user whenever there are conflicts at the individual aspect level of the object.

Last Commit Wins

Problem: How can you serialize a set of changes from overlapping business transactions that affect the same object when your primary concern is that users should not have to reenter any data and the probability of conflict is relatively small?

Solution: Replay all of the changes in a new transaction, overwriting previously committed changes if necessary.

6. Updating views at transaction boundaries

One aspect of an enterprise application architecture is to specify the way in which the presentation layer is updated by changes to the underlying business objects. In this section, we provide two alternative patterns for this problem.

Push Domain Model Changes

Problem: When one user commits a transaction that updates the domain model on the server, how can a client application display these changes to a different user?

Solution: The client registers as an observer of specific model components and the server notifies observers when changes occur.

Pull Domain Model Changes

Problem: When one user commits a transaction that updates the domain model on the server, how can a client application display these changes to a different user?

Solution: The client aborts its transaction periodically, in order to refresh its view of the repository.

Acknowledgments

Although I am responsible for developing this pattern language of transaction models, much of the insight and terminology was obtained from others. This is generally true of patterns, since one benefit of a pattern is that it captures widely followed practices. However, I want to especially acknowledge my reliance on course materials, publications, or personal conversations from (in alphabetical order): Michael Dillon, Trygve Reenskaug, Colleen Roe, Adam Springer, and Blaine Wishart.

About the author

Jim Tyhurst is an independent consultant, who often plays the role of an applications architect. He holds a Ph.D. in Linguistics from UCLA, where he developed mathematical and computational models of natural language semantics. He has been using object-oriented technology since 1992, preceded by 12 years of application development using artificial intelligence techniques, relational databases, and structured languages. Jim has designed and developed extensible class hierarchies for business objects in Java and Smalltalk, as well as object-oriented frameworks for application infrastructure. He has been working in the area of Enterprise Application Integration (EAI) since 1995.

Contact information

Jim Tyhurst, Ph.D.
jim@tyhurst.com

Tyhurst Technology Group LLC
14335 S. Hawthorne Ct.
Oregon City, OR 97045
(503) 632-7416