HPTS '95

Challenges in Component-Based Transaction Systems

Mohsen Al-Ghosein
Product Unit Manager
Transaction Processing
Microsoft Corp.
mohsena@microsoft.com

Pat Helland
Architect
Transaction Processing
Microsoft Corp.
phelland@microsoft.com

Introduction

Building enterprise applications out of binary components mandates that the component model be integrated with the transaction processing model.

Such integration presents a number of problems that are presented in this paper. A particular problem related to transaction commit authority and the lack of trust between components is presented in more detail. A discussion of the approaches proposed or implemented by many component transaction systems presents why such approaches are inappropriate in the practical case. An alternate approach is proposed to address the problem.

What is a Component ?

A component is:

- A binary object which encapsulates code and data.
- Written in any language that supports the binary contract between components
- A set of interoperability rules between components from various vendors
- A scheme to address the evolution and versioning problems associated with mixing and matching components
- A packaging model which allows components to be bundled, distributed, and deployed

Problems in Component-based Transaction Systems

Having defined a component system, it is generally not enough to assume that components will be simply interoperable. One needs to have an application architecture and the associated infrastructure to fully define the domain of interoperability between components. For example, Microsoft's Visual Basic application development environment defines such an architecture for visual elements (custom controls).

In the attempt to define an application architecture and the associated infrastructure, a number of key challenges arise:

1. The difficulty of defining a transaction model which addresses the needs of both database and document oriented computing. The isolation levels, concurrency models, and abort semantics, are very different between a document system (like Microsoft's Excel) and a relational database (like Microsoft's SQL Server). In most cases, a client which consumes the services of two components under one transaction many need to understand some of the internal semantics of the storage system used by them so that it knows what to expect in the case of failure.
2. Encapsulating a component's data behind its code presents a number of challenges when it comes to being able to perform query operations on the data.
3. Components must not have to trust other components.

It is the third problem above that will be the focus of this discussion. There are additional problems not presented above.

We believe that in a componentized transaction system, each component must have a means of not trusting the other components of this system[1]. Historically, the applications running on a TP system have been designed and implemented within a single organization. This has meant that a framework for protecting oneself hasn't been important. Now, as we enter a componentized world, we will see more and more components that are off-the-shelf products from different vendors. These cannot afford to assume proper behavior from other components.

Imagine a server component which happens to be part of an order entry system. The architecture calls for building an order component which represents a single order with many line items. The order component's interface has a method to define the order header and the number of line items to insert. This is followed by iterations of a method which inserts individual line items.

The order component needs to be designed so that it can work on its own independent transaction or on a larger transaction scoped by its caller (client). Furthermore, the order component needs to be designed so that even the most malicious client cannot damage the consistency of the order component.

In a particular scenario, the evil client starts a transaction, calls the order component's header method informing it that 20 line items will follow. It then makes a number of calls to the line item method and commits the transaction. If the client promises to send 20 line items and only passes 10, the component's consistency rule is violated. The problem arises each time the component returns control back to the client when it is not yet consistent. The client is always free to commit the transaction even if the component is not done with its work.

Historically, there have been two ways of avoiding this problem:

- Make each component context free
  This means that each component only gets a single call. The component cannot return to the client in an inconsistent state. By definition, it's not gonna get called again.
- Ignore the problem
  Since the client is trusted (historically), it'll do the right thing.

Components imply that neither of these solutions is general enough:

- Context is inadequate for most component interfaces
- Components come from diverse sources and can't be trusted

[1] Note that it is OK for two components to trust each other (and perhaps derive benefits from this trust); components need the option of not trusting.

Overcoming the Problem

Four approaches have been proposed for solving this particular problem

1. Disallow the client from being able to scope server transaction
   In other words, make the entire transaction run at the server

2. Make the server save everything up until the last call
   This means that the server doesn't change the database except in the last call

3. Use nested transactions
   The component can commit or abort a nested transaction within the larger transaction

4. Make the server pretend it's an RM and play Two Phase Commit
   In this fashion, the component can vote no if it's pissed-off.

Each of the approaches outlined above has serious drawbacks:

1. Disallow the client from being able to scope server transaction
   This simply ducks the problem since servers can also be clients. It essentially restricts the entire application to be exactly two tiered

2. Make the server save everything up until the last call
   This scheme requires the server to cache potentially large amounts of data, makes servers harder to write, may have different isolation semantics, and complicates the server's interface. In general, it's a pain-in-the-butt to write a server this way.

3. Use nested transactions
   Until the world of RMs changes, this is hardly practical. With rare exceptions, RMs do not support nested transactions due to implementation difficulty and potential overhead related to the added complexity in the locking code.

4. Make the server pretend it's an RM and play Two Phase Commit
   The fourth approach is perhaps the most problematic. The justification follows:
   - The server has to enlist its database on the transaction so that it doesn't have to keep persistent data.
   - If the database enlists on the transaction, then the application server's vote needs to be collected before any of the databases (the server will need to push its data out to the databases)
   - This creates ordering demands on the transaction manager performing two phase commit coordination. And is very difficult to achieve if parallelism is allowed.
   - If the server wants to receive commit notification, then it must have a durable identity (be recoverable), otherwise, it needs to vote abort or read only at prepare time and go away. If it votes read only, then it only gets one shot at validating the transaction, if other components are called afterwards, they may indeed violate the consistency of the server that just went away.

     Being an RM is a difficult proposition.

Another Way to Solve The Problem

We argue that the transaction coordinator must provide a means by which each component can block the commit of the transaction until later. While this is allowed in LU6.2 (via the SYNCPT application verb), it is noticeably missing from most Transactional RPC implementations.