Distributed Object Transactions

By Scott Ambler, July 01, 2000

Applications used to run on centralized mainframes that also hosted data. Things aren't so simple anymore.

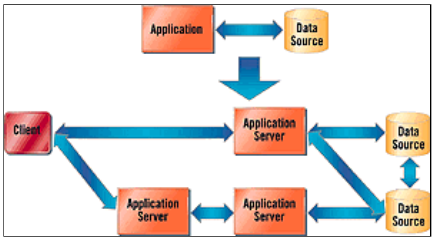
You have just won the lottery. Now you're out on the street, entrusting your big check to the local bank machine. You press the deposit button, select your savings account, input the amount that you have won, stick the check in an envelope as requested, feed the envelope into the machine, and it suddenly shuts down. The bank machine has taken your check but has not updated the balance in your account. Will you get your money or not? If the system is built to be transactional, you'll be able to buy that big house and car; if not, you still have to report for work on Monday.

A transaction is a unit of work that either completely succeeds or completely fails (the terminology of transactions is summarized in the sidebar). In the lottery check example, the transaction is to accept the deposit, update the balance in the appropriate bank account and post a financial record documenting the deposit. Either all three of these steps should occur or none of them–there is no middle ground. This example actually describes a distributed transaction, one whose steps occur in several different software nodes. The action of taking the deposit occurs at the bank machine itself, the updating of the account would occur on an application server somewhere within the bank's system, and the recording of the transaction would occur within the bank's database. The update of the account balance is an interesting step because it occurs at two nodes: within a bank account object running on an application and within the corporate database used to persist the bank account object.

The need for transaction control has existed since at least the 1960s, when information technology was first deployed for business applications. At that time, the system architecture was very simple: An organization ran software applications on a centralized mainframe computer that also hosted the data source(s) for the applications. The application itself typically acted as the transaction coordinator and focused on ensuring that data was consistently read from and written to a single data source. As complexity increased, it was common for an application to access several data sources simultaneously, heralding the introduction of sophisticated transaction coordinators called Transaction-Processing (TP) monitors (for example, IBM's CICS in the mid-to-late 1970s). Over time, the types of transactions that applications required also became more complex, requiring distributed architectures as well as object-oriented and complex database technologies as implementation platforms.

Figure 1 depicts the evolution from a centralized mainframe application to the distributed, n-tier architecture employed by leading-edge applications today.

Figure 1. From Centralized Mainframes to Distributed Architectures



Centralized mainframe applications have evolved to today's distributed, n-tier architecture.

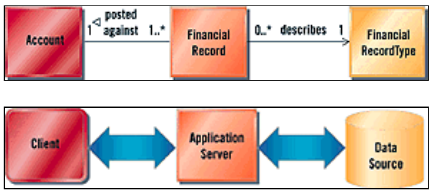
Complex Architecture

Transaction control is very straightforward when you have a single program manipulating the data of a single data source; it can often be managed by your data source itself. In fact, sophisticated transaction control is a common feature in most database technology

(Oracle, Sybase, DB2 and Versant). For simple applications, the approach often proves sufficient. Unfortunately, few things are simple these days. Modern applications are written using distributed object technology such as Enterprise JavaBeans, Common Object Request Broker Architecture (CORBA)-based technology, and, ostensibly, Microsoft COM+-based technology. These technologies typically have clients, such as personal computers or Internet browsers, interacting with application servers using object-oriented or object-based technology that in turn interact with databases on the back end. While the implementation architecture may have changed, the need to support ACID (atomic, consistent, isolated and durable) transactions has not. Your distributed object applications therefore need distributed object transaction control.

Consider an example: Figure 2 depicts a simple business schema, using the Unified Modeling Language class diagram notation, that is to be implemented within a three-tier client server environment. An object-oriented version of the business schema implemented in Java will be deployed to the application server, which in turn will be mapped one-to-one to an identical relational schema in the database. I have kept the example simple, so I can focus on the issues involved with distributed object transactions; I could have used another language such as C++ or Visual Basic, implemented this on an n-tier architecture involving many application servers and/or many data sources. The environment may become more complex, but the fundamental issues remain the same.

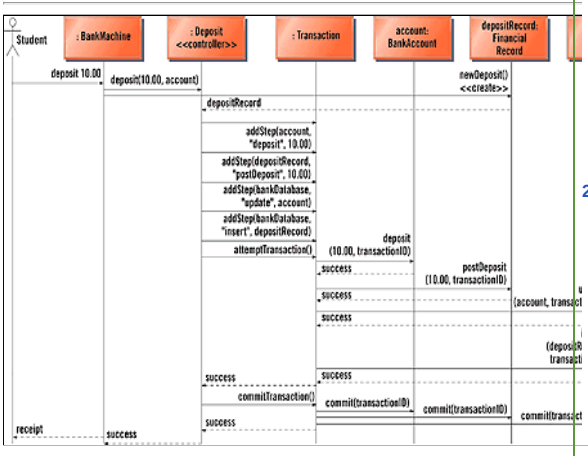
Figure 2. A Simple Business Scheme



An object-oriented version of this scheme (top) will be deployed to the application server (bottom).

Figure 3 depicts a UML sequence diagram for successfully depositing $10 into a bank account. Sequence diagrams are used to show the dynamic interactions of objects fulfilling a usage scenario, a topic that I cover in detail in the newly released *The Object Primer*, 2nd Edition (Cambridge University Press, 2000). The user interacts with the banking system via a bank machine (the client of Figure 3) and requests to deposit $10. The bank machine interacts with an instance of the Deposit class that controls the overall process. It interacts with the `FinancialRecord` class to create an instance of it representing the record of the actual deposit. Next, it collaborates with the Transaction object to start a transaction and proceeds to build the transaction one step at a time. Each transaction step is either an update to a business object, such as the account itself or the record of the deposit, or change to the database where information about the business objects is stored. A better approach may have been to have the business objects add the appropriate steps to update the database as needed, but I wanted to keep Figure 3 as simple as possible. The transaction object is then told to attempt itself, so it loops through the steps, attempting them one at a time. The appropriate operation is invoked on each object, which by definition must be a transaction resource that supports attempts, commits and rollbacks. Because each step was successful, the transaction is requested to commit itself, and it loops through each transaction resource one at a time telling it to commit the transaction. Had a step failed, the transaction would have been requested to roll itself back, and in turn it would have requested each transaction resource to roll back its part(s) of the transaction.

Figure 3. A UML Sequence Diagram for a Successful Deposit Trans



Sequence diagrams show the dynamic interactions of objects fulfilling a usage scenario; in this case, depositing $10 into a bank account.

The Good Old Database Days

There are several interesting points to be made here. First, transactions must be managed throughout your entire system–they are neither simply an application server issue nor a database issue. I have run across several organizations that are struggling in adopting new technology because they are still mired in "old-school" thinking from their centralized-database days about the transaction management features of their relational database.

Second, distributed object transaction control is very hard. My example was simple; in actuality, you would need to implement this in an environment where there are many application servers accessing several databases and thousands of concurrent users.

Third, locking is a key enabler of transaction control. As each transaction resource attempts an operation, it must obtain a lock on itself–either for read or for write–to ensure the ACID characteristics of the transaction. When the deposit is being made to the account object, a write lock would be obtained, whereas when the deposit record object is being posted, a write lock would be obtained on it and a read lock on the appropriate instance of `FinancialRecordType`. The read lock is needed to ensure referential integrity within your system; you wouldn't want to create a financial record of "deposit type" and then have "deposit type" deleted before your financial record was written to the database.

Fourth, referential integrity, like transactions, is a system-wide issue and not simply a database issue. Referential integrity–the act of ensuring that associations between objects are always valid–is a business logic issue. It is a fundamental mistake to assume that your database is sufficient to maintain referential integrity within a distributed object application, a mistake commonly made by "old-school" thinkers. It is just as important that the association between a customer object and an account object on your application server is valid as is the association between the customer row and the account row in your relational database.

Finally, transaction control is a common requirement for most of today's business applications and is therefore a topic with which all developers should be familiar.

When do you need to apply your newly gained knowledge about distributed object transactions? Whenever you are building systems using common technologies such as the Java Transaction Service (JTS), the Common Object Request Broker Architecture (CORBA), Object Transaction Service (OTS) or Microsoft Transaction Server (MTS). More information can be found about each of these technologies at java.sun.com/products/jts/, www.omg.org and www.microsoft.com, respectively.

Distributed object transaction management is hard, but it is critical to your success as a software developer. For further reading about transaction management I recommend the books *An Introduction to Database Systems* (Chris J. Date, Addison-Wesley, 1995), *The Essential Distributed Objects Survival Guide* (Robert Orfali, Dan Harkey and Jeri Edwards, John Wiley & Sons, 1996) and *Distributed Object Architectures with CORBA* (Henry Balen, Cambridge University Press, 2000). I also recommend the Web site www.opengroup.org for information about Extended Architecture (XA), an industry standard for distributed transaction processing.

The secret to success is to recognize that your transaction must work end-to-end, from your client through your various application servers all the way down to your data source(s).

The Terminology of Distributed Object Transactions

Transaction. A transaction is a unit of work, perhaps a collection of updates on several objects, which either completely succeeds or completely fails. A nested transaction includes subtransactions that may succeed even though the parent transaction does not. A flat transaction has no subtransactions.

Transaction step. A portion of a transaction, either an operation on a transaction resource, a lock on a required resource or a subtransaction. Each step can be attempted, rolled back and committed (in the case of a read lock a roll-back and a commit are both simply unlocks).

Transaction resource. An object that may fully participate in a transaction, including an attempt at an operation as well as the rollback or commit of that operation. Also known as a transaction target.

Distributed transaction. A transaction that involves transaction resources running two or more distinct nodes, such as an application server and a relational database.

Distributed object transaction. A distributed transaction involving one or more transaction resources implemented using object-

oriented technology.

Transaction manager. An object or component that manages the life of a transaction, including its initial attempt and subsequent rollback or commit. Also known as a transaction coordinator.

Attempt. The steps of a transaction are tried but not completed. A step is successful if the appropriate locks can be obtained on a transaction resource or, in the case of a subtransaction, the subtransaction is successfully attempted and committed. An attempt is successful when all steps of a transaction are attempted successfully.

Rollback. A transaction is backed out when an attempt is not successful. The operations on the transaction resources are not committed and all locks are released. Any committed subtransaction will not be affected, although uncommitted ones also will be rolled back.

Commit. All steps of a successfully attempted transaction are finalized against their targets. All uncommitted subtransactions are committed, all operations against transaction resources are run, and all locks are released.

Two-phase commit. The process by which the steps of a transaction are first attempted; if they are all successful, they are committed; otherwise, they are rolled back.

Objectspace. The environment in which objects exist, including both their "live" operational aspects as real Java, C++ or other objects in memory as well as their "flattened" data aspects in data storage.

Locking. When a transaction must ensure that an object in which it is interested will not change in some unpredictable manner during the transaction then it acquires a lock on the object. A read lock is an indication that the transaction only needs to access to the object but will not do anything to update it. A write lock indicates the transaction will need to modify the object.

Optimistic locking. An object is marked when it initially accessed, perhaps via a version number or date-time stamp. When changes to the object are later attempted the object is write locked, the original marking is checked to see if it has changed, and if it hasn't the update is committed (if it has an error or exception is returned).

Pessimistic locking. An object is initially locked and kept locked for the length of time that it is needed. With any updates being made, the updates are committed and the object is eventually unlocked. This is a simple locking approach that does not scale well because it does not permit ready access to shared resources.

ACID characteristics. Atomicity, consistency, isolation and durability are the four fundamental characteristics that a transaction should exhibit.

• Atomicity requires that all of the operations of a transaction are performed successfully for the transaction to be considered complete. If all of a transaction's operations cannot be performed, then none of them may be performed.

• Consistency refers to object consistency. A transaction must transition the object from one consistent state to another, preserving the object's semantic and physical integrity.

• Isolation requires that each transaction appear to be the only transaction currently manipulating the object. Other transactions may run concurrently, however, a transaction should not see the intermediate object manipulations of other transactions until and unless they successfully complete and commit their work. Because of interdependencies among updates, a transaction might get an inconsistent view of the objectspace were it to see just a subset of another transaction's updates. Isolation protects a transaction from this sort of object inconsistency.

• Durability means that updates made by committed transactions persist regardless of failures that occur after the commit operation; it also ensures that objectspace may be recovered after a system or media failure.

Related Reading

- News
- Commentary

- **Application Intelligence For Advanced Dummies**
- **Parallels Supports Docker Apps**
- **Docker Clocks In On Azure**
- **Mac OS Installer Platform From installCore**
  More News»

- Slideshow
- Video

- **2013 Developer Salary Survey**
- **NoSQL Options Compared**
- **Developer Reading List**
- **2012 Jolt Awards: Mobile Tools**
  More Slideshows»

- Most Popular

- **State Machine Design in C++**
- **Lambdas and Streams in Java 8 Libraries**
- **So You Want To Write Your Own Language?**
- **A Simple and Efficient FFT Implementation in C++:**
  Part I
  More Popular»