Java theory and practice: Understanding JTS -- An introduction to transactions

Transactions are the building blocks of reliable applications

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix Corp

01 March 2002

The Java Transaction Service is a key element of the J2EE architecture. Together with the Java Transaction API, it enables us to build distributed applications that are robust to all sorts of system and network failures. Transactions are a basic building block of reliable applications -- it would be prohibitively difficult to write a reliable distributed application without transactional support. Fortunately, JTS does most of its work transparently to the programmer; the J2EE container makes transaction demarcation and resource enlistment nearly invisible. This first installment of a three-part series covers the basics of what transactions are and why they are critical to building reliable distributed applications.

View more content in this series

If you look at any introductory article or book on J2EE, you'll find only a small portion of the material devoted to the Java Transaction Service (JTS) or the Java Transaction API (JTA). This is not because JTS is an unimportant or optional portion of J2EE -- quite the opposite. JTS gets less press than EJB technology because the services it provides to the application are largely transparent -- many developers are not even aware of where transactions begin and end in their application. The obscurity of JTS is in some sense due to its own success: because it hides the details of transaction management so effectively, we don't hear or say very much about it. However, you probably want to understand what it's doing on your behalf behind the scenes.

It would not be an exaggeration to say that without transactions, writing reliable distributed applications would be almost impossible. Transactions allow us to modify the persistent state of an application in a controlled manner, so that our applications can be made robust to all sorts of system failures, including system crashes, network failures, power failures, and even natural disasters. Transactions are one of the basic building blocks needed to build fault-tolerant, highly reliable, and highly available applications.

The motivation for transactions

Imagine you are transferring money from one account to another. Each account balance is represented by a row in a database table. If you want to transfer funds from account A to account B, you will probably execute some SQL code that looks like this:

```
SELECT accountBalance INTO aBalance
    FROM Accounts WHERE accountId=aId;
IF (aBalance >= transferAmount) THEN
    UPDATE Accounts
        SET accountBalance = accountBalance - transferAmount
        WHERE accountId = aId;
    UPDATE Accounts
        SET accountBalance = accountBalance + transferAmount
        WHERE accountId = bId;
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (aId, -transferAmount);
    INSERT INTO AccountJournal (accountId, amount)
        VALUES (bId, transferAmount);
ELSE
    FAIL "Insufficient funds in account";
END IF
```

So far, this code looks fairly straightforward. If A has sufficient funds on hand, money is subtracted from one account and added to another account. But what happens in the case of a power failure or system crash? The rows representing account A and account B are not likely to be stored in the same disk block, which means that more than one disk IO will be required to complete the transfer. What if the system fails after the first one is written, but before the second? Then money might have left A's account, but not shown up in B's account (neither A nor B will like this), or maybe money will show up in B's account, but not be debited from A's account (the bank won't like this.) Or what if the accounts are properly updated, but the account journal is not? Then the activities on A and B's monthly bank statement won't be consistent with their account balances.

Not only is it impossible to write multiple data blocks to disk simultaneously, but writing every data block to disk when any part of it changes would be bad for system performance. Deferring disk writes to a more opportune time can greatly improve application throughput, but it needs to be done in a manner that doesn't compromise data integrity.

Even in the absence of system failures, there is another risk worth discussing in the above code -- concurrency. What if A has $100 in his account, but initiates two transfers of $100 to two different accounts at the exact same time? If our timing is unlucky, without a suitable locking mechanism both transfers could succeed, leaving A with a negative balance.

These scenarios are quite plausible, and it is reasonable to expect enterprise data systems to cope with them. We expect banks to correctly maintain account records in the face of fires, floods, power failures, disk failures, and system failures. Fault tolerance can be provided by redundancy -- redundant disks, computers, and even data centers -- but it is *transactions* that make it practical to build fault-tolerant software applications. Transactions provide a framework for enforcing data consistency and integrity in the face of system or component failures.

What is a transaction?

So what is a transaction, anyway? Before we define this term, first we will define the concept of *application state.* An application's state encompasses all of the in-memory and on-disk data items that affect the application's operation -- everything the application "knows." Application state may be stored in memory, in files, or in a database. In the event of a system failure -- for example, if the application, network, or computer system crashes -- we want to ensure that when the system is restarted, the application's state can be restored.

We can now define a *transaction* as a related collection of operations on the application state, which has the properties of *atomicity*, *consistency*, *isolation*, and *durability*. These properties are collectively referred to as *ACID* properties.

Atomicity means that either all of the transactions' operations are applied to the application state, or none of them are applied; the transaction is an indivisible unit of work.

Consistency means that the transaction represents a correct transformation of the application state -- that any integrity constraints implicit in the application are not violated by the transaction. In practice, the notion of consistency is application-specific. For example, in an accounting application, consistency would include the invariant that the sum of all asset accounts equal the sum of all liability accounts. We will return to this requirement when we discuss transaction demarcation in Part 3 of this series.

Isolation means that the effects of one transaction do not affect other transactions that are executing concurrently; from the perspective of a transaction, it appears that transactions execute sequentially rather than in parallel. In database systems, isolation is generally implemented using a locking mechanism. The isolation requirement is sometimes relaxed for certain transactions to yield better application performance.

Durability means that once a transaction successfully completes, changes to the application state will survive failures.

What do we mean by "survive failures?" What constitutes a survivable failure? This depends on the system, and a well-designed system will explicitly identify the faults from which it can recover. The transactional database running on my desktop workstation is robust to system crashes and power failures, but not to my office building burning down. A bank would likely not only have redundant disks, networks, and systems in its data center, but perhaps also have redundant data centers in separate cities connected by redundant communication links to allow for recovery from serious failures such as natural disasters. Data systems for the military might have even more stringent fault-tolerance requirements.

Anatomy of a transaction

A typical transaction has several participants -- the application, the transaction processing monitor (TPM), and one or more resource managers (RMs). The RMs store the application state and are most often databases, but could also be message queue servers (in a J2EE application, these would be JMS providers) or other transactional resources. The TPM coordinates the activities of the RMs to ensure the all-or-nothing nature of the transaction.

A transaction begins when the application asks the container or transaction monitor to start a new transaction. As the application accesses various RMs, they are *enlisted* in the transaction. The RM must associate any changes to the application state with the transaction requesting the changes.

A transaction ends when one of two things happens: the transaction is *committed* by the application, or the transaction is *rolled back* either by the application or because one of the RMs failed. If the transaction successfully commits, changes associated with that transaction will be written to persistent storage and made visible to new transactions. If it is rolled back, all changes made by that transaction will be discarded; it will be as if the transaction never happened at all.

The transaction log -- the key to durability

Transactional RMs achieve durability with acceptable performance by summarizing the results of multiple transactions in a single transaction log. The transaction log is stored as a sequential disk file (or sometimes in a raw partition) and will generally only be written to, not read from, except in the case of rollback or recovery. In our bank account example, the balances associated with accounts A and B would be updated in memory, and the new and old balances would be written to the transaction log. Writing an update record to the transaction log requires less total data to be written to disk (only the data that has changed needs to be written, instead of the whole disk block) and fewer disk seeks (because all the changes can be contained in sequential disk blocks in the log.) Further, changes associated with multiple concurrent transactions can be combined into a single write to the transaction log, meaning that we can process multiple transactions per disk write, instead of requiring several disk writes per transaction. Later, the RM will update the actual disk blocks corresponding to the changed data.

Recovery upon restart

If the system fails, the first thing it does upon restart is to reapply the effects of any committed transactions that are present in the log but whose data blocks have not yet been updated. In this way, the log guarantees durability across failures, and also enables us to reduce the number of disk IO operations we perform, or at least defer them to a time when they will have a lesser impact on system performance.

Two-phase commit

Many transactions involve only a single RM -- usually a database. In this case, the RM generally does most of the work to commit or roll back the transaction. (Nearly all transactional RMs have their own transaction manager built in, which can handle *local transactions* -- transactions involving only that RM.) However, if a transaction involves two or more RMs -- maybe two separate databases, or a database and a JMS queue, or two separate JMS providers -- we want to make sure that the all-or-nothing semantics apply not only within the RM, but across all the RMs in the transaction. In this case, the TPM will orchestrate a *two-phase commit*. In a two-phase commit, the TPM first sends a "Prepare" message to each RM, asking if it is ready and able to commit the transaction; if it receives an affirmative reply from all RMs, it marks the transaction as committed in its own transaction log, and then instructs all the RMs to commit the transaction. If an RM fails, upon restart it will ask the TPM about the status of any transactions that were pending at the time of the failure, and either commit them or roll them back.

A societal analogy for the two-phase commit is the wedding ceremony -- the clergyman or judge first asks each party "Do you take this man/woman to be your husband/wife?" If both parties say yes, they are both declared to be married; otherwise, both remain unmarried. In no case does one party end up married while the other one doesn't, regardless of who says "I do" first.

Transactions are an exception-handling mechanism

You may have observed that transactions offer many of the same features to application data that synchronized blocks do for in-memory data -- guarantees about atomicity, visibility of changes, and apparent ordering. But while synchronization is primarily a concurrency control mechanism, transactions are primarily an exception-handling mechanism. In a world where disks don't fail, systems and software don't crash, and power is 100 percent reliable, we wouldn't need transactions. Transactions perform the role in enterprise applications that contract law plays in society -- they specify how commitments are unwound if one party fails to live up to their part of the contract. When we write a contract, we generally hope that it is superfluous, and thankfully, most of the time it is.

An analogy to simpler Java programs would be that transactions offer some of the same advantages at the application level that `catch` and `finally` blocks do at the method level; they allow us to perform reliable error recovery without writing lots of error recovery code. Consider this method, which copies one file to another:

```
public boolean copyFile(String inFile, String outFile) {
  InputStream is = null;
  OutputStream os = null;
  byte[] buffer;
  boolean success = true;

  try {
    is = new FileInputStream(inFile);
    os = new FileOutputStream(outFile);
    buffer = new byte[is.available()];
    is.read(buffer);
    os.write(buffer);
  }
  catch {IOException e) {
    success = false;
  }
  catch (OutOfMemoryError e) {
    success = false;
  }
  finally {
    if (is != null)
      is.close();
    if (os != null)
      os.close();
  }

  return success;
}
```

Ignoring the fact that allocating a single buffer for the entire file is a bad idea, what could go wrong in this method? A lot of things. The input file might not exist, or this user might not have permission to read it. The user might not have permission to write the output file, or it might be locked by another user. There might not be enough disk space to complete the file write operation, or allocating the buffer could fail if not enough memory is available. Fortunately, all of these are handled by the `finally` clause, which releases all the resources used by `copyFile()`.

If you were writing this method in the bad old C days, for each operation (open input, open output, malloc, read, write) you would have to test the return status, and if the operation failed, undo all of the previous successful operations and return an appropriate status code. The code would be a lot bigger and therefore harder to read because of all the error-handling code. It is also very easy to make errors in the error-handling code (which also happens to be the hardest part to test) by either failing to free a resource, freeing a resource twice, or freeing a resource that hasn't yet been allocated. And with a more substantial method, which might involve more resources than just two files and a buffer, it gets even more complicated. It can become hard to find the actual program logic in all that error recovery code.

Now imagine you're performing a complicated operation that involves inserting or updating multiple rows in multiple databases, and one of the operations violates an integrity constraint and fails. If you were managing your own error recovery, you would have to keep track of which operations you've already performed, and how to undo each of them if a subsequent operation fails. It gets even more difficult if the unit of work is spread over multiple methods or components. Structuring your application with transactions lets you delegate all of this bookkeeping to the database -- just say ROLLBACK, and anything you've done since the start of the transaction is undone.

Conclusion

By structuring our application with transactions, we define a set of correct transformations of the application state and ensure that the application is always in a correct state, even after a system or component failure. Transactions enable us to delegate many elements of exception handling and recovery to the TPM and the RMs, simplifying our code and freeing us to think about application logic instead.

In Part 2 of this series, we'll explore what this means for J2EE applications -- how J2EE allows us to impart transactional semantics to J2EE components (EJB components, servlets, and JSP pages); how it makes resource enlistment completely transparent to applications, even for bean-managed transactions; and how a single transaction can transparently follow the flow of control from one EJB component to another, or from a servlet to an EJB component, even across multiple systems.

Even though J2EE provides object transaction services relatively transparently, application designers still have to think carefully about where to demarcate transactions, and how we will use transactional resources in our application -- incorrect transaction demarcation can cause the application to be left in an inconsistent state, and incorrect use of transactional resources can cause serious performance problems. We will take up these issues and offer some advice on how to structure your transactions in Part 3 of this series.

Resources

- *Transaction Processing: Concepts and Techniques*, by Jim Grey and Andreas Reuter, is the definitive work on the subject of transaction processing.
- *Principles of Transaction Processing*, by Philip Bernstein and Eric Newcomer, is a fine introduction to the subject; it covers a lot of history as well as concepts.
- The Java Transaction Service specification is quite readable and offers a high-level explanation of how an object transaction monitor fits into distributed applications.
- The lower-level details of transactional support in J2EE is detailed in the Java Transaction API (JTA) specification.
- The J2EE Specification describes how JTS and JTA fit into J2EE and how transactions interact with other J2EE technologies, such as Enterprise JavaBeans.
- In their article "Understanding quality of service for Web services" (*developerWorks*, January 2002), Anbazhagan Mani and Arun Nagarajan discuss how transactions should satisfy the ACID test.
- Find other Java-related articles and tutorials on the *developerWorks*Java technology zone.

About the author

Brian Goetz

Brian Goetz is a software consultant and has been a professional software developer for the past 15 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, CA. See Brian's published and upcoming articles in popular industry publications.