email

Pyrasun - The Spille Blog

Saturday April 03, 2004

search this site

All | General | Java | Books | Groovy | XA | NIO

XA Exposed, Part I

Update 11-Jan-2004: *Many thanks to the people here and on TheServerSide.com who offered many corrections and clarifications, both grammatical and technical in nature. The suggested changes have been made here as a result of those corrections, and some things make alot more sense now - thanks!*

If you liked this article and haven't seen the TSS thread, you may want to peruse it - several people offered some very interesting insights on this subject. Thanks also go to Dion Almaer for linking to this from TSS.

TSS Discussion on XA Exposed Part I.

Links

Site Links

Front Page
comments
About
Mike
Archive
Weblog
Login

Technorati

Blog Roll

Cameron
Cedric
Bile
Blog
Dion
Rickard
Carlos
Fishbowl

The following is Part I of my magnum-opus on how the innards of 2 Phase Commit works in a J2EE environment. The focus here is on the interaction between XA resources, like JMS providers, JCA adapters, and JDBC drivers, and the transaction manager, which is typically an application server like Websphere, WebLogic, JBoss, or Jonas. The article concerns itself almost exclusively on these sorts of innards, and specifically does *not* look at things from the application developer's point of view. You won't find details on setting up your deployment descriptors or looking up XA-compliant drivers here. No, this is the down 'n' dirty - this is all of the usually [illegible] and covered 2PC plumbing - exposed!

Part I gives a brief background on XA, reviews basics of the protocol, and then shows how the protocol works in action with multiple resources. More than anything else, this sets the stage for more advanced XA concepts.

Part II will go into much greater detail on failure recovery scenarios, go into some of the more subtle bits of the J2EE XA protocol, and provide warnings for XA problems which application developers should look out for.

Note: Much of the material in this entry was compiled early in 2003, and my editing time has been severely limited in putting this all together from my notes. Given the complex subtlties in the XA protocol, chances are I may have gotten one or two pieces wrong (but certainly no more than that :-). If anyone spots any errors I'd appreciate it if you point them out!!

Introduction

Back in the late 1980s, and continuing into the '90s, there was a continuing need and drive to integrate systems. Managers, business users, and even the lowly software developer began to realize that their company had a wide range of disparate systems, each often costing millions of dollars to develop and maintain - and that there was significant amount of overlap between many of these systems. Even when functionality varied widely between two isolated applications, often they would share common data.

To counter the waste of uncounted millions of dollars (not to mention the frustration of subtly out of sync data across the enterprise), efforts began to unify systems. The first step was data integration - how do we get data across to multiple systems?

To make a long story short, companies tried a number of ways to integrate data across these systems, and thus they hoped to stem the flow of those uncounted millions of dollars that was caused by these "data islands". As you might expect, these early enterprise integration efforts lived up to the "enterprise" moniker by causing the waste of uncounted millions of *new* dollars - and since they rarely worked well, those *other* millions going to the legacy islands and stovepipe applications continued to flow as well. Thus, those of us involved had enough work to keep us going until the next hot thing came along....

But some technological good came out of these early pioneers. One of these goodies was the *two phase commit protocol*, which is alternatively referred to as "2PC" or XA (I'm mostly going to say "XA" here, since it has the positive attribute of being short and easily recognizable). XA is designed to accomodate the concept of *global transactions*. Where regular Relational Database Management Systems (*RDBMS'*) featured transaction capabilities within themselves which exposed developers to the ACID principals of transaction processing, global transactions extended these ACID concepts to multiple databases (and ultimately, multiple "resources" in general). In a nutshell, global XA transactions let you update multiple resources in sync - and therefore commits and/or rollbacks will be done consistently for all resources involved in the transaction (e.g. all resources will either be committed, or rolled back in coordination with the global transaction).

So the key to XA is two-fold: updating multiple resources at once, and doing it in a global transaction context that adheres to ACID principals. These two aspects are highly intertwined and lay at the heart of XA. It's not just about updating multiple resources, but updating them in a way

that adheres to the ACID tenets. This gives application developers a powerful tool - but the tool has some high costs.

This little article will begin by dissecting 2PC transactions, and showing exactly what's happening under the covers. It will then precede to show how these hidden mechanisms affect application programs, their costs and tradeoffs, and what application developers can do to optimize these mechanims for their own environment and needs.

Before we dive right into the technical spaghetti of XA transaction flows and requirements, let me address something that may be on your mind: why should *I* care? XA is one of those weird, complex things that huge multi-national conglomerates do, one of those bizzaro ideas inherited from those dinosaurs of the main frame world. Right?

Well, until recently that was indeed the case. XA was pretty rare, and in fact it was rare enough that alot of very well known vendors got surprisingly large pieces of it wrong - and few people noticed. But with the advent of JMS, the returning focus on integrating across applications, and the growing influence of the Java Connector Architecture (JCA), XA is suddenly appearing on the scopes of many developers for the first time. Increasingly, people want to hit multiple resources simultaneously - their ERP system, their RDBMS, and maybe publish a few messages or throw something on a JMS queue while they're at it - and they want to do this in a transactionally sound manner. And the way to do this is with XA. And along the way, many developers are finding out about the pitfalls of XA along with the positives, and are *also* discovering an old adage coming alive once again: "Yep, XA sucks, but for us it sucks less than the alternatives".

2PC Refresher

First, a little refresher on the XA protocol. For a given XA resource, a JTA transaction manager will get a handle on the resource's XAResource object, and then call methods in following sequence. If it helps for you to follow along, take a look at the **Javadoc for XAResource** . For brevity's sake, this is an oversimplification of the protocol, but it conveys the flavor fairly well:

1. start(Xid) - enlist the resource as part of the transaction
2. end(Xid) - tell the resource that no more transactional work is coming its way
3. prepare(Xid) - prepare for commiting. The resource can respond "OK" or "oh no!". The latter indicates that the whole global transaction should be rolled back.
4. commit(Xid) - Really commit.

The Xid parameter identifies the transaction - it is a unique key to the transaction from the XA resource's perspective. The main important bits of the Xid are the "global transation ID" and the "branch qualifier". All XA resources for a given transaction will share the same global transaction ID; additionally, each resource will see its own "branch qualifier" (which is unique to the global transaction). So, for instance, if you've got 3 XA resources involved in a global transaction, you'll have 3 Xids, all with the same global transaction ID, but each with a different "branch qualifier". For those in the know, "branch qualifier" means "just the bit this XA resource should care about".

You don't strictly need to know this minutiae about Xids, but it's a useful datum to file away in your hind brain somewhere. If you're anally retentative and just have to know everything, check out the **Javadoc for Xid here.**

Gettin' back to the XAResource, The start() and end() calls tell the resource when it's starting to get involved in a transaction, and when it's involvement ends. For practical purposes, you can think of start() as saying "OK, all future work should be recorded as part of this transaction until I tell you otherwise". The end() call is the "otherwise" - signals to the resource to stop recording. All of the work between the calls should be held in a bucket somewhere keyed by the Xid.
The latter two calls - prepare() and commit() - are the real part of 2PC. Once all the work is done in a transaction, the transaction manager first calls all the resource managers with a prepare() call. This means "get ready to commit, but don't do it yet!". At this point, all resources get to vote on the outcome of the transaction. If any resource votes no, the transaction rolls back. If they all vote yes, the transaction will be commited. If the latter occurs, the transaction manager then calls commit() on all of the resources.

Differentiating between the start()/end() bits and the prepare()/commit() bits can be a little confusing to people who aren't knee-deep in XA. Don't worry about it - it confuses most everyone the first time through. Think of it this way: start() and end() is the transaction manager's way of letting the XAResource know when to flip on transactional recording and turn it off again. Before start(), you're not in a transaction. After end(), you're also not in a transaction - but you have to remember everything you did between start() and end(), tag it with the Xid, and hold onto that information in a bucket somewhere. At this point, post-end(), you've done a bunch of work but not unleashed it globally anywhere. The unleashing is where prepare() and commit() come into play. The prepare() part makes sure everyone thinks the transaction is kosher, the commit() part does the final bit of unleashing of all the transactional work out into the wild.

Dissecting the Lifecycle of a 2PC transaction

The JTA specification, along with its philosophical parent, the X/Open XA specification, go to great lengths to show the individual pieces of XA transaction processing. But, in the way of such detailed specifications, they neglect to show a comprehensive example that pulls everything together for the reader. This I will attempt here.

The previous section shows how things look from the perspective of a single resource. Now let's look at the whole shebang. Figure 1 below outlines the lifecycle of a typical global transaction involving 2 XA resources. In this example, a J2EE client is making a request to an application server (Websphere in this case, mainly because I know how Websphere does XA). This example request entails doing some work with a SQL Database, and publishing a message out to JMS. All of this activity takes place within the context of a global, XA transaction. The diagram shows the order in which the XA operations are invoked, on which resources, and emphasizes heavy weight operations vs. lightweight operations. For simplicity sake, all of the JTA transaction rigamarole is elided - we're focusing just on the extras that XA brings into the picture.

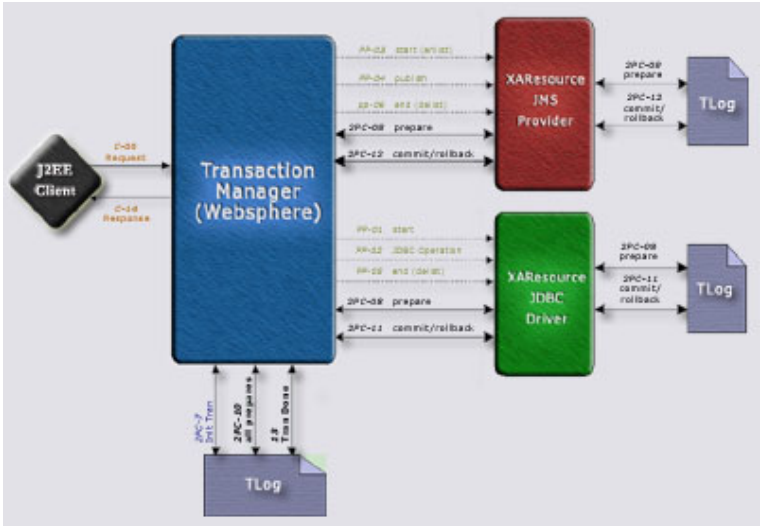

Figure 1 - lifecycle of a typical global 2PC transaction - click picture to enlarge

Reading the Diagram

- The diagram depicts the major operations involved in performing 2PC with two XA "resources". The details of the resource manipulation are elided to highlight strictly the operations involved in XA.
- The sample transaction shown is performing a SQL operation with a database, and publishing a message via JMS in a single global transaction. For simplicity, assume these operations are taking place within a stateless session bean, with the transaction controlled by the EJB container.
- Dotted arrows w/ light text indicate "lightweight" requests with little blocking. Note that "lightweight" is a relative term here - obviously, some not-so-innocent SQL statements can take a long time to execute!
- Heavy arrows w/ bold black text indicate heavyweight operations in the XA protocol that involve disk forcing somewhere in the chain
- The transaction request is initiated on the left by the J2EE client.
- The order of operations are defined by sequence codes associated with the arrows. The codes consist of a prefix followed by a number. The prefix defines where generally in the lifecycle the operation takes place, the number is the sequence number (going from 0 up to 14).

  Sequence Codes:
  - C - Client-side event.
  - PP - Pre-prepare. The transaction not durably recorded anywhere yet, failure implies rollback.
  - 2PC - 2 Phase Commit. The transaction has been durably recorded in at least one point. Failure implies Transaction Manager must recover transaction information and resolve outcome - may result in rollback or commit, depending on transaction state at time of failure.
- The diagrams show both the JMS XA Provider and JDBC XA driver as single, monolithic units with writes to their respective transaction logs exposed. In reality, the providers "seen" by Websphere are a) a JMS client side library which adheres to the JMS interfaces, and b) a client side JDBC XA Driver which adheres to the JDBC interfaces. These client side drivers typically connect to a remote JMS server/RDBMS server. The client/server nature of both have been hidden in the diagram for purposes of clarity and compactness; nevertheless, it's important to remember that they are typically remote resources

Definitions of items in the Diagram:

- J2EE Client - The stand-alone client initiating the transaction request. All actual transaction processing is handled within the Websphere.
- Websphere/Transaction Manager - Websphere acts as both the server interface point for the client, the J2EE EJB container, and the 2PC transaction manager for the transaction. Other app servers conceptually can serve the same role and generally do so via JTA.
- XAResource JMS Provider - The sample JMS provider is assumed to be a "foreign" JMS provider which Websphere sees as an XAResource.
- XAResource JDBC driver - An XA-aware JDBC driver which Websphere sees as an XAResource.
- TLog - a transaction log used to durably record XA operations and their outcomes. Each XAResource must have a transaction log (or its logical equivalent), in addition to Websphere's transaction log.

Sequence of Events

The following table describes the events in a typical transaction as shown in figure 1. We're focusing on normal transaction flow here where responses are generally in the positive, and the transaction commits.

Note: The ordering of JMS vs. JDBC is undefined. JMS may be "hit" first, or the database may be "hit" first. For our example we're showing the database being hit first.

Table 1 - Sequence of Events in Global Transaction

Event		Description
C-00		Client Request to App server

PP-01		JDBC XA start() - Websphere enlists JDBC connection in transaction. This typically occurs within container-level code when the EJB application code gets the JDBC connection.
PP-02		JDBC operations - select, insert, update, delete, etc. This occurs within the application EJB code as it performs normal JDBC operations.
PP-03		JMS XA start() - Websphere enlists the JMS provider session in the transaction. This typically happens within container-level code when the application EJB code gets its JMS session. Note: where JDBC providers are involved in transactions at the connection level, JMS providers are involved in transactions at the JMS session level. This is because the JMS specification creators inexplicably added a Session abstraction on top of Connections, and it is at the JMS Session level where the XAResource refence can be grabbed.
PP-04		JMS operations - pub/sub publish, queue send, etc. This occurs within the application EJB code as it performs normal JMS operations.
PP-05		JDBC XA end() - Websphere delists the JDBC connection from transaction. This typically occurs when the application calls close on the JDBC connection.
PP-06		JMS XA end() - Websphere delists the JMS session from transaction. This occurs when the application code calls close on the JMS session.
2PC-7		Websphere writes transaction & XA resource info to tranlog, disk forces. This operation is shown in light blue on the diagram because while it's a heavy weight disk forcing activity, Websphere only writes this information in its transaction log once for its resource on its first use.
2PC-8		JDBC XA prepare() - Websphere sends a prepare() request to the JDBC XA driver and gets its vote. The database server must write the prepare() to the tranlog if its vote is "commit" and must disk force it.
2PC-9		JMS XA prepare() - Websphere sends a prepare() request to JMS provider and gets its vote. The JMS server must write the prepare() to the tranlog if its vote is "commit", and must disk force it.
2PC-10		Websphere writes prepare() votes of XA resources, disk forces. This will only happen if at least one resource voted "commit". If the first resource voted rollback, then this action is never recorded.
2PC-11		Database XA commit() - Websphere sends a commit() request to the JDBC XA driver. The database server force-writes the commit() to its tranlog and must disk force
2PC-12		JMS XA commit() - Websphere sends a commit() request to to the JMS XAResource. The JMS server must write the commit() to the tranlog, and must disk force.
2PC-13		Websphere writes "Tran done" to indicate transaction is over.
C-14		Client receives response

As you can tell from all the pretty lines in the diagram, and from the myriad events shown in the table (all 15 of them!), there's alot of stuff going on here, and this example just shows 2 XAResources. Each additional XAResource adds another 6 or 7 events. But the seeming complexity here is a bit deceiving - alot of these events are vanilla work that need to happen anyway - the start() is just a transaction start for a resource, the actual work obviously has to happen anyway, and you've gotta commit somehow. To truly understand the impact of all of this, and to see where the "weight" of XA truly becomes felt, you need to focus on the important bits - those with the highest costs.

Pay attention to the Force, Luke

A critical item to understand about XA in the J2EE world is that your highest costs in terms of execution time are going to occur from "disk forcing". Most of the transactional work in XA is plain old work that you'll have to pay for one way or another - your RDBMS insert/update/selects, your JMS publishing, etc. Bits of the XA protocol around these pieces add no really noticable costs - in fact, in some implementations of XAResources, calls like XA start() and end() may be coded as fire-n-forget, so you're not even paying for a round trip network cost to the server.

The real cost in XA is disk forcing the transaction logs at critical points in the protocol. Let me 'splain what I mean here, Lucy.

In order to meet the failure/recovery requirements of XA, the Transaction Manager and all XAResource managers have to record transaction information "durably". In practical terms, this means they have to save the data to disk in some sort of transaction log. The XA protocol defines exactly when such transaction log disk-forces have to happen. This gives guarantees to the various components and allows them to make certain assumptions.

The points marked in figure 1 as 2PC-08, 2PC-09, 2PC-10, 2PC-11, and 2PC-12 are the critical junctures in the XA protocol where suitable transaction data *must* be durably recorded to disk. A bit more specifically, each XA Resource must durably record its transaction information when it votes "yes" to a prepare call, the Transaction Manager must durably record "committing..." if all XA Resources vote yes, and each XA resource must durably record all XA commit calls.

Here's how that works in practice. The start()/end() calls on each XA resource, the actual work done by each resource in a transaction (insert, delete, publish, etc), and all of the Transaction Manager work during this time period all happens only memory. Something like an a database insert may involve some disk activity, and in general disk hits may be taken as needed by an XAResource, but it's *not required*. For the most part, all of the work before the first prepare() call happens as fast as possible, and in memory if you can.

The prepare() calls are different. Each XAResource which votes "OK" on a prepare call *must* record that fact to disk. In practice, this means that the resource writes out important control information to its transaction log covering the prepare()'d transaction *and then disk forces*. The disk force bit means that XAResource flushes all of its I/O buffers all the way to the physical disk i.e. you're forcing your data out. Prior to prepare(), nothing has to be written to disk if you don't want to, but at prepare() you have to write enough info to disk so that your server could die immediately afterward, and that data can be recovered sufficiently to complete the transaction.

Likewise, if all XAResources vote "OK", at this point the Transaction manager must write a record out to its transaction log saying "Committing...", and that has to be disk forced as well. And, finally, each XAResource has to disk force the commit() calls.

Why all this disk forcing? The reason is for proper failure and recovery. Remember that the purpose of XA is to ensure that all resources are in sync. This 'in sync' mantra extends to failures as well. If you lose your network, if a machine melts down, if an errant production operator does a "kill -9" on your server process, XA does its best to ensure that once you recover your server process, processing continues normally - even for those transactions that were "in flight" when the process went down.

Note that XA doesn't provide absolute guarantees in this regard, but instead it tells you what assumptions you can make. It tells you when you know the state of a transaction definitively - and when it doesn't know. You'll have to wait until the next section to explain exactly why this is so, but take my word for it for now. At the moment, understand that the exact disk-force points are important, and that these points are your number one costs in using XA.

That's all for now...

Part II will pick up right where we left off with more details on disk forcing and protocol guarantees, and will proceed into much greater detail on failure recovery scenarios, some of the more subtle bits of the J2EE XA protocol, and will explain in detail some troublesome aspects of XA which application developers should be aware of and look out for in their own environments. If I have the time (and energy!), I may also do a part III which goes into greater detail on the protocol from the perspective of an XAResource developer's e.g. the realities of creating an XA-compliant resource that plugs into an app server with all the bells 'n' whistles.

April 03, 2004 02:05 PM EST **Permalink**

Comments [15]

XA Exposed, Part II

Update 16-Jan-2004 15:01:23 NYC time: Description of indeterminate error returns to client has been expanded, several minor grammatical mistakes have been fixed.

Update 14-Jan-2004 18:35:30 NYC time: Last resource gambit has been altered slightly, thanks to some offline correspondence with Greg Pavlik which made me rethink that bit a bit.

Recap

Part I of *XA Exposed* focused on the basics . A semi-fictional history of XA was offered, a focus on XA's goals was explained. A brief tangent explained why the average J2EE developer might wish to care about all of this.

We then all took a little tour of prime players in J2EE's implementation of XA: the XAResource, which is the gateway from the transaction manager into any "resource" which may be used in a 2PC manner, and the Xid, the grand identifier which lets the XAResources know which transaction the Transaction Manager is bleating about *now* (those TMs are really just insufferable).

These basics were tied together in a simple yet typical 2PC transaction: the flow of information from the TM out to the resources, and back again, involing a stateless session bean, an RDBMS, and a JMS provider. A few of you even managed to slog through all 14 events - bravo!

The article concluded with a real cliff hanger, talking about the dominating role that disk forcing plays in the 2PC transactions.

Of course, TheServerSide.com **also got into the act**, with most of the drama we've come to love and expect from TSS. Cameron Purdy being the squeaky clean geek with massively in-depth knowledge of distributed computing. Guglielmo Lichtner and Horia Muntean asking a number of astute questions and forcing me to divulge almost half of what was destined here for Part II (shame on you guys, robbing me of page views! You must be shills for TSS...). Mark Little keeping me honest. Even Andreas Mueller dropped in. Sadly, no JBoss guys deigned to join our little soiree (that's French, right Nathalie?), and Rolf never showed up, so ultimately it can only be considered a second-rate TSS thread. All kidding aside, the comments from the TSS thread have contributed a great deal of the material contained in Part II, and in a very real way the participants there helped to shape final form of this entry.

Now, with Part II, we're going to go into alot more depth on this subject. Part I really covered only the easy parts when nothing goes wrong, with just a few references to failure states and recovery. Here in Part II we're going to dive deep into the murky depths of rollbacks, heuristic decisions, optimizations like Presumed Abort, the 1PC optimization, and similar more advanced concepts. In programming, they say the last 10% takes 90% of the effort, and the same is true when trying to explain the XA protocol. When everything works, XA is pretty easy to explain. But when you have to deal with correctness, and durability, and you wanna do it all as fast as possible, then you're talking a whole 'nuther kettle

of fish. All the corner cases and exceptional conditions take alot more effort to deal with and explain then the straight line stuff. So gird your loins, cinch in your belt, and let's get it on....

P.S. You didn't actually believe any of that stuff from the trailer, now did you? Silly Wabbit...

References

The details of XA are complex enough that you shouldn't take my words as correct in isolation. Beyond any errors I might make, these articles really are only scratching the surface of global transactional processing. If you want to be a true Transactional Jedi Knight (ack, the JBoss Mind Control Device must have gotten through my tinfoil for a moment, sorry about that...). Anyway, if you really want to know this stuff inside and out, consulting a wide array of references is a must. Here's a smattering of web links I found useful:

- ***Commit Processing in Distributed On-Line and Real-Time Transaction Processing Systems", Ramesh Kumar Gupta***
  An excellent description of distributed transactional processing via Gupta's Masters Thesis. This thesis approaches transaction processing from an abstract point of view as opposed to a C or Java spec, such as X/Open and J2EE provide. Pages 22-40 in particular have a wealth of detail on 2PC, Presumed Abort, etc. When you go to the site, click on PDF under "View or download" on the upper right hand side of the page.
- ***Distributed TP: The XA Specification, The Open Group***
  The definitive XA specification. Warning to Java zealots: the spec is written entirely in terms of C. The link to the document is on the far right of the page.
- ***Java Transaction API (JTA)***
  The J2EE transaction standard. This standard is not only based on the X/Open *The XA Specification*, but in fact it leaves vast pieces of XA transaction processing more or less undefined. To understand half of what this spec is talking about, you must be fully conversant with *The XA specification*, and to make the trip even more amusing, you have to mentally transform C concepts from the Open Group's spec into Java on your own.
- ***The javax.transaction API javadoc***
  The Javadoc for the main JTA API.
- ***The javax.transaction.xa API javadoc***
  The Javadoc for the XA part of the JTA API (tiny, ain't it?)

For a full understanding of how everything hooks together, you'll also have to read through the relevant parts of the JMS specification, the JDBC specification, and the JCA specification. I know, I know, what a monstrous PITA - but if it were easy everybody would be doing it!

Defining Failure Categories

There three two broad categories of failures that can occur during an XA transaction:

1. Pre 2PC failures.
2. Failures during prepare() phase
3. Failures during commit() phase

It's important to consider these seperately, because each is handled in a radically different way within XA. This section is going to primarily enumerate the various failure scenarios so as to divvy them up nicely into readily identifiable categories. The following section will then detail how these failures are handled in the XA protocol.

Failure Category #1: Pre 2PC failures

Pre 2PC failures are problems that occur before we ever get into the 2PC cycle of disk forcing, prepare() and commit() calls. In the context of our **XA lifecycle original diagram from Part I**, these cover PP-01 to PP-06. Here we're using the imaginary code behind that diagram to illustrate this category of failure, so keep the example of a stateless session bean publishing to a JMS provider and doing some database calls via JDBC in your head.

In such a setup, many things can go wrong "pre 2PC", including:

1. An exception can be thrown up by the JMS provider - loss of connectivity to the JMS server, violation of Message class constraints, etc.
2. An exception can be thrown by the JDBC driver - bad SQL, constraint violation, loss of connectivity to the database, etc.
3. The application can cause a rollback on its own, or throw an exception to achieve the same ultimate end
4. The transaction can timeout.
5. The application server could come crashing down in flames

The key to remember here is that any of these failures generally mean a rollback should happen, and additionally that any such rollback is happening before we ever enter the prepare()...commit() cycle. Typically, these rollback conditions are happening due to exceptions or rollback calls from within the application code itself, or from within code the application is using, in this case the JMS provider and JDBC driver.

Failures during 2PC

Failures which occur during the 2PC cycle are another beast altogether. In an EJB environment, such failures generally happen after the application code has executed, and are thereby seen only by the container and, possibly, the caller of the EJB. When transactions are being manually controlled, these failures will occur when the application code calls txn.commit(). Since handling of such errors differ depending on where exactly you are in the 2PC lifecycle, I'm splitting them into what I'll call the prepare() cycle and the commit() cycle. The prepare() cycle

from the lifecycle diagram covers 2PC-08 and 2PC-09. The commit() cycle encompasses 2PC-10, through 2PC-13.

In the literature, when failures occur in this stage, the transaction is said to be *in-doubt*. From the time that the first resource has had prepare() called on it, up until the last resource has been called with commit() or rollback(), the transaction is in-doubt. You will find that many people abuse this term and call any transaction which was in-flight during a failure to be in-doubt, but now you can show your superior brainpower and knowledge and pedantically correct them on this :-)

It's important to remember that in-doubt transactions (in the pedantic sense) cannot be unilaterally rolled back by XAResources. Once a resource votes "yes" to a prepare() call, that transaction is effectively durably locked in until the TM tells the resource to rollback() or commit() (an exception to this role is Heueristic decisions - more on this later). The resource specifically cannot time out a prepare()'d transaction, even if the connection to the TM is lost.

A semi-complete list of failures that can happen on the XAResource side of the house can be found **in the XAException javadoc**. As you can see, there is a dizzing array of error codes that are supported. These can be broken down into a few clearly related areas:

- Heuristic decisions. A heuristic decision is said to have occured when someone alters an XAResource's transaction state without the knowledge or direction of the TM. Heuristic in this context most often means that an operator or admin somewhere went into the database/ JMS admin console/whatever and manually committed or rolledback a transaction. More bluntly, it typically implies that something got screwed up somewhere, and a human made a guess on how to resolve a transaction (indeed, Heurstics most often mean a guess). If this guess was wrong (e.g. the admin said "rollback" when in fact a commit was the right thing to do), your global transaction state can be put out of sync. Some of the possibilities here are covered in XAException from XA_HEURCOM to XA_HEURRB.
- Rollbacks. The underlying resource may have rolled back without the TM's direction. This may have happened because the network link to the resource was severed, a deadlock happened, the transaction timed out within the resource, or a few other reasons. These are covered from XA_RBBASE to XA_RBTIMEOUT.
- Someone screwed up. There are a number of XAException codes which indicate that either the TM or the XAResource screwed the pooch and did something illegal in the XA protocol. This includes XA_PROTO, XAER_INVAL, and XAER_DUPID.
- Who knows? If it doesn't t know what caused a problem, an XAResource may just use the XAER_RMERR or, more popularly, XAER_RMFAIL codes.

People who pride themselves in well-formed and logical Exception hierachies will note that XAException is a big mish-mash of stuff and is terribly designed. All of the XAResource methods throw XAException, and you have to pour through the javadocs of the methods to try to figure out what code may apply to what condition. This most often leads to big switch statements or if...else...if... blocks inside of TMs within their catch blocks to try to figure out what happened during an exception. The kindest thing I can say about this design is that it's a big steaming pile of dog crap. But for now that steaming pile is all we've got.

Failure Category #2: Failures during Prepare() Phase

Failures during the prepare() phase include:

1. An XAResource could vote "no" to a prepare() call via a number of states set in a thrown XAException.
2. The Transaction Manager could crash during the prepare() cycle. This could happen due to logical application error, a so-called "heuristic decision", transaction timeout within the XAResource, and failure of the Resource or loss of connectivity to it.
3. The Transaction Manager could crash after the prepare() cycle but before forcing a "Committing..." record.

Failure anywhere in this area of the lifecycle always implies rollback, as with the Failure Category #1 cases. The difference here, however, is that the implications are a bit more serious, since some XAResources may have durably recorded their transaction information already.

Failure Category #3: Failures during Commit() Phase

Failures during the commit() phase include:

1. An XAResource can throw an exception on a commit() call. This could happen due to logical application error, a so-called "heuristic decision", failure of the Resource, or loss of connectivity to it.
2. The Transaction Manager could crash right after forcing a "Committing..." record, but before calling commit() on anything.
3. The Transaction Manager could crash while calling commit() on the XAResources.

Failure anywhere during the commit() cycle are generally quite dire. Such a failure at this phase of the cycle implies that your global system state could be unsynchronized, with some resources committed and others not yet.

The Missing Failure Condition

The astute reader may have noticed that I haven't mentioned one type of failure: disk failure. The reason why is simple: if you lost the disk containing your TM's transaction log, or the disk for one of your XAResources, you are screwed. If you lose your TM's log, you've lost the only definitive source of your global system's state. If you lose the disk for one of your XAResource - say, the one for your RDBMS - well, the resource is basically hosed.

This goes beyond mere data loss, because it also means that the carefully synchronized resources that XA is supposed to try to guarantee are no longer synchronized. It is for this reason that large organizations invariably put things like transaction logs on RAID arrays with battery backup, hot swappable disks, and every other possible fault tolerance bell and whistle you can think of. There is a secondary benefit to this as well: if you're using a RAID array or SAN type network storage device, your system can survive catastrophic server loss by allowing multiple machines to mount the device. If a failure slags down a server to monomolecular particles, mount the drive on your handy backup machine, bring the processes for the failed machine up on the new one, and you're back in bizness. This sort of failover scenario, based on the catastrophic loss of a server, is a fascinating topic, but is beyond the scope of this article.

Failure Recovery and Presumed Abort

Now that we've neatly categorized the various errors and failures, it's time to see how they're handled under XA. Note that the JTA spec has very little to say on this subject - it is up to the reader to essentially know the X/Open XA spec by heart and to, presumably, divine the rest from cosmic inspiration. The spec leaves much to be desired when it comes to recovery operations and rollbacks - going so far as to show just a single state diagram for the normal committing case, and offering no diagrams for failure conditions at all. To infer legal call states for error conditions, one must turn to the XA Spec and decipher tables like Table 6-4 therein. Presumed abort is never mentioned, even in passing, in JTA - you have to hunt that down in section 2.3 of the X/Open XA spec, too.

Given this state of affairs, my own information on these topics comes from a combination of a very small bit of information from the JTA spec, quite a bit more from X/Open, various research papers on the net, and by observing several Application Servers in action.

Handling Category #1, Pre-2PC Failures

Failure category #1, Pre-2PC failures, are the easiest to explain. In any of these cases, no information has been durably recorded when the failure occured - and hence, any failure means a rollback must happen. Whether its your JMS provider throwing an exception, the database barfing on bad SQL, or catastrophic failure of your Application Server, the outcome is the same - rollback.

Just how the rollback happens is dependent on the circumstances. In the most common case, a failure will happen in some component and raise an exception, or else the application will direct the transaction to rollback directly. In either case, if this occurs the Transaction Manager calls rollback() on all live XAResources that have been enlisted on the transaction.

If connectivity is lost somehow to one of the XAResources, most commonly that XAResource will report something like XAER_RMFAIL. In that circumstance, we can't really get to the underlying resource (like the database) to indicate that a rollback should happen. As it turns out, this is OK, and one of two things will happen. Most commonly, the server-side of the XAResource will detect that the connection has been dropped or severed, and will rollback the transaction locally of its own accord.

In the less common case where the server does not detect connection loss, or cannot, or where really lazy programmers are employed who couldn't be bothered to detect such conditions, the transaction should time out locally if we don't hear from the Transaction Manager for a time.

Of course, the server behind the XAResource may also have dumped core all over the place, and this calls for a slightly different handling of the problem. In this situation, it's highly possible that when you bring that server back up that all pre-2PC information was just plain lost - which is OK, because we don't expect durably during this phase. In this case you have an implicit rollback within that XAResource. Alternatively, the underlying transaction log implementation and a bit of luck may have resulted in the information being held around somewhere. If this happens, then you'll have to rely on the transaction information being timed out internally after restart. To some this is not entirely kosher - anything pre-prepare should generally be thrown out in failure cases. But for some implementations, like a JMS provider, it may be harmless to hold onto to some pre-2PC transactions if you happen to have them laying around in your log and may simplify your coding - so long as you have a timeout mechanism!

While I'm on the topic of timeouts....because of the indeterminably of network programming, and the fact that network losses sometimes can go undetected for a long time, every resource which is designed for use with XA should have a configurable timeout mechanism. This is especially important on the RDBMS side, where resources can be locked up due to operations like updates and deletes. This is a lesson that's sadly left out of many distributed discussions, and is most often learned the hard way by someone new to the field. Whenever information is being transferred remotely, you must have a timeout to ensure that resources get cleaned up in the event of really bad failures.

The corollary to this is that the user of distributed systems should take pains to ensure that all components in their system have reasonable and consistent timeouts set. There are few things more annoying than the smart ass who sets the JMS timeout to 2 seconds, or the zealous RDBMS admin who sets their timeouts to 257 days.

Getting back to Pre-2PC failures, I stated that exceptions in some component, or alternatively directed rollbacks, are the most common case. The less common case is failure of the TM itself (e.g. your app server takes a header off the Chrysler building). As it turns out, this works in a similar manner to the XAResource getting disconnected from the TM. The various XAResources will detect the loss of connectivity and rollback, or will alternatively timeout and rollback. Of course the TM generally does do some extra work in this case, but I'll leave that for a bit later in this discussion.

Handling Category #2, Prepare()-Phase Failures

Any failure during the prepare() phase of a transaction always implies rollback. If any XAResource fails on its prepare() call, or the TM itself fails before writing its "Committing..." record, then the outcome has to be rollback. In the case where the TM is still up, the TM must call rollback() on any XAResources which have already voted "yes" (and have not responded with an error or XA_RDONLY). In the case where the TM itself fails, the outcome is still "rollback", but it's now up to the Transaction Manager recovery mechanism to ensure that rollback() is called on all resources.

Handling Category #3, Commit()-Phase Failures

Handling failures during the commit() phase are generally similar to handling those in the prepare() phase, except in this case the outcome must always be commit. Once the Transaction Manager successfully force writes its "Committing..." record, the outcome of the transaction has to be commit.

In the case where the Transaction Manager itself fails, once again the Transaction Manager recovery mechanism must come into play (this is described in the next section).

In the case where an XAResource reports an error, things get interesting. The overall global transaction has to commit, but one or more XAResources are reporting problems, and in the meanwhile the inititator of the transaction is waiting around for a result. For any true error

condition, like getting an XAER_RMFAIL error back from an XAResource's commit() call, the Transaction Manager has to balance the needs of the calling client with its own need to complete the transaction.

In practice, TM's split the difference here. Possibly after some number of retries, they'll typically report back to the client that the transaction outcome is incomplete in some way. In practice this means the transaction maybe partially committed. In the meanwhile, the Transaction Manager will put the partially completed transaction on a background thread and keep trying to complete it. This is of course not the only possible course of action - alerts may be raised on a console, options may be given to an operator to manually take charge of the transaction in some way, etc. By and large J2EE application servers do not seem to be this sophisticated, but other non-J2EE TP Monitors have some serious infrastructure built up to take care of exactly this sort of condition.

Either way that it's handled, the application programmer must be aware a transaction may ultimately committed - but not yet, and they may get an error code back indictating that partial commit has happened, and hopefully some day the rest will be completed. In addition, in certain architectures if the TM itself fails, the client may not know if the transaction failed in the prepare() phase or the commit() phase (or, indeed, the transaction may have been fully commited, but failed before reporting that back to the caller). The net sum of these various scenarios which a client may see is that there are three states you have to code for in an application:

1. Committed - everything went well and you know for a fact it committed.
2. Rollback - an error occured somewhere along the chain, and a rollback was issued and is complete.
3. Indeterminate - an error occured somewhere along the chain, but the client doesn't know where

The first two are self-explanatory. The third case is the tough one - the client may lose connectivity to the server, or the TM may have failed, or one or more of the RMs may have failed, all of which leaves the client in a state where it does not know the transaction outcome.

The Transaction Manager Recovery Lifecycle

Failure of the Transaction Manager itself, and recovery of that process, kicks in the full-blown XA recovery procedure. In the event that the TM blows chunks down Wall Street, on reviving itself the TM has to take a number of steps to recover any transactions which were in progress at the time of the crash. An outline of those steps are shown below in two seperate areas:

Straight Line Recovery:

1. Find transactions that the TM considers dangling and unresolved
2. Find and reconstitute any XAResources which were being used when chunk blowing occured.
3. Call the recover() method on each of these XAResources.
4. Throw out any Xids in the XAResources' recover lists which are not owned by this TM.
5. Correlate Xid's that the TM knows about with remaining Xid's that the XAResources reported.
6. For XAResource Xid's that match the global transaction ID which the TM found dangling with a "Committing..." record, call commit() on those XAResources for those Xids.
7. For XAResource Xid's that do not match any dangling "Committing..." records, call rollback() on those XAResources for those Xids.

Exceptional conditions:

1. For any rollback() calls from step 6 which reported a Heuristic Commit, you are in danger or doubt, so run in circles, scream and shout.
2. For any commit() calls from step 7 which reported a Heuristic Rollback, you are in danger or doubt, so run in circles, scream and shout.
3. For any resource you can't reconstitute in in step #2, or who fails on recover in step #3, or who reports anything like an XAER_RMFAILURE in step 6 or step 7, keep trying to contact them in some implementation defined manner.
4. For any heuristic outcome you see reported from an XAResource, call forget() for that XAResource/Xid pair so that the resource can stop holding onto a reference to that transaction

I'll step away from the exceptional conditions for a minute, because they're just too painful to contemplate right now. For the moment, let's take a page from the JTA spec and just focus on the positive side.

The heart of the TM recovery lifecycle lies in the assumptions that come from using the *Presumed Abort Protocol*. Presumed abort is an optimization to "regular" XA that allows fewer disk forces to occur and still achieve correctness. In all this is a good thing - anything which helps XA go faster must be considered a good thing!

Presumed Abort can be a bit annoying to decipher in a J2EE context, because most texts on the subject assume a much less constrained environment than JTA gives us. For example, Gupta defines Presumed Abort in this manner:

> As described in the previous section, the 2PC protocol requires transmission of several messages and force-writing of several log records. A variant of the 2PC protocol, called presumed abort (PA), attempts to reduce these overheads by requiring all cohorts to follow a "in the no information case, abort" rule.

The problem here, and where confusion can especially arise for J2EE oriented people, is that this description of Presumed Abort (as indeed is in the case with many academic treatises on the subject) make the assumption that Resources Managers may query the Transaction Manager. There's much discussion of "cohorts" and queries from them up to the master TM, and anyone who's only read the JTA spec will invariably become hopelessly confused. A smaller but persistent problem is use of the word "Abort", when everyone knows real programmers Rollback.

In practical terms, what Presumed Abort means to J2EE is nothing more than the following:

- Transaction Managers only need to durably record a "Committing..." record. They do not need to record a "Start Trans" record, or a "Committed" or "Rolled Back" record (though they may do so in a non-forcing manner for convenience).
- XAResources don't need to durably record anything until they vote yes to prepare() (but then, we knew that already :-)
- Since XAResources can't get to the TM, the TM comes to them: The TM calls recover() on everybody's XAResource, and matches that info with dangling "Committing..." records in its logs.

In J2EE terms, the above is the whole of Presumed Abort in all of its glory.

Of course, from a TM's perspectives things aren't quite so simple as I've described here. Problem #1 is figuring out what XAResources were out and about when the TM failed. Problem #2 is figuring out how to get real XAResource objects back. Problem #3 is doing something constructive if your XAResources decide not to cooperate.

Problem #1 in reality isn't too difficult. The Transaction Manager must note what XAResources are getting used, and optimally record them just once somewhere. This can be in the transaction log or elsewhere.

Problem #2 is slightly more difficult. You see, an XAResource isn't serializable. It's obtained by calling a method like getXAResource() from a JDBC connection or a JMS XA Session, and you just can't stash it somewhere. What this means is that the TM has to record all information which will allow it to reconstitute the XAResource in the first place. For JDBC, it means it has to record all the driver information so it can create a JDBC connection and then get the XAResource for it. Likewise, for JMS it has to get the right XA[Topic|Queue]ConnectionFactory, and then build a JMS connection and session from there, and finally get the XAResource from the session. In practice, only XAResource is obtained using one connection/session during this recovery process, and all our commits and aborts will funnel through it, regardless of how many transactions may be in-doubt. As a side note, this can be a bit of a pain in the ass for JMS, JDBC, and JCA provider writers - you have to consider that a rollback() or commit() may not come from the original connection/session from which the transaction was initiated.

Problem #3 is, quite frankly, a bitch. Let's say your TM gets pulled back together after its stunning crash, and starts trying to get its XAResources back - and one of them fails? Even worse, what if you get 'em all back, but one or more of them start throwing Heuristic exceptions back at you that don't match your expectations?

For this sort of problem, there are two widely divergent trains of thought. One attitude is that if you can't recover then you're hosed, and so the TM aborts. Some popular application servers actually do this - if an XAResource can't be manufactured on restart, or throws an unexpected error, the application server throws up its hands and exits with an error.

Another attitude is to take a page from the Little Engine That Could. If you fail, keep trying until it works. This could mean delaying bring the application server all the way up until you succeed, or possibly spinning off a background thread to do it in the background, and otherwise to come up normally. Which approach your application server takes is very important thing to find out about, because whether or not your application server can come back up quickly under certain failures can obviously be a rather critical datum to consider.

The second part of the problem, dealing with incompatible Heuristic decisions, has no clean answers. Consider a case like this - admittedly rare, but these things do sometimes happen in real life:

- The system's humming along, transactions are flowing, your boss is smiling, all is good with the world.
- The boss' son, Little Jimmy, sneaks into the server room and pulls the plug on your Application Server machine's UPS. He plugs in his GameCube w/ LCD in its place, and plays a bit of Zelda while your UPS slowly winds down to 0 ergs.
- The system operator in charge of seeing that the UPS light is flashing on his monitoring console, is instead frantically running around on the corporate floors trying to find Little Jimmy for the boss.
- The UPS reaches the end of its rope, gives one last Gasp "Oh, Stella, if I'd only told you how I truly felt!", and then promptly expires, taking the machine and your application server along for the ride to go see Thanatos.
- The horrified system operator, peeking into all the toilet stalls on the 17th floor (still looking for Little Jimmy), hears over the corporate-wide intercom "John SysOp, John SysOp, please come to the 4th floor operations room immediately. The server opsx1115 has gone down, a quarter of the users are offline, and the boss swears he heard a UPS in the machine room calling the name 'Stella'. The talking power supplies and downed servers require your immediate attention. Thank you and have a nice day."
- John SysOp runs to the 4th floor, finds that not only has all hell has broken loose, but Satan himself is sitting in his cubicle drinking his cafe latte, and almost as bad finds the door to the production machine room is jammed. Meanwhile, checking his consoles (after directing Satan to the men's room on the 17th floor), John sees that half the transactions in the system are blocked due to 10 "in-doubt transactions" in DB2.
- John consults his runbook, frantically skims the index, and finds the entry on page 1,231 which covers downed application servers, jammed doors, Satan sitting in his cubicble drinking lattes, and a slight drizzle coming from the West. First checking that it is indeed drizzling outside, John follows the directions: "Log onto DB2, list in_doubts, rollback all in-doub transactions. Call maintenance to unjam the door. Borrow and umbrella from Suzie on the fifth floor. AND ABOVE ALL, DON'T LET SATAN GO TO THE BATHROOM!
- Despite his lapse with the Satan thing, John considers he's done good. Transactions aren't blocked, maintenance has unjammed the door, he's found Jimmy, he's plugged the UPS back in, and the app server's coming back up
- Upon sauntering back to his console, John finds 7 messages on the alert console. "ERROR: Heueristic rollback is incompatible with the global state of this transaction - WHY DID YOU LET SATAN GO TO THE BATHROOM!?!?"

The moral of this story is, of course, that the Satan thing was a red herring - really, showing someone to the bathroom is the only thing to do in a polite society, even if he is the Devil in the Flesh. No, the real devil here was unilaterally rolling back all of the DB2 transactions. 3 of them, as it turned out, worked out OK. But the other 7 did not, and the end result was that DB2's state is now inconsistent with the rest of the system, all thanks to the Runbook inspired Heuristic decisions. Fortunately for John, Little Johnny hacked into the corporate net a few minutes later through his GameCube, and slagged down the entire network infrastructure to smouldering ruin. But do you think **you'll** be so lucky as to have a little Jimmy of your own to come in and save your ass?

Leaving the anecdote aside for a moment (it's all true, I swear, Richard Saunders told me about it just yesterday), heuristic decisions aren't necessarily evil, but they can cause alot of problems. Clearly, corporations need an "out" in the cases where a TM just cannot be brought up, and this is especially critical if there are in-doubt transactions holding locks in your database. But you should be aware of these cases and the consequences of heuristic decisions.

The Nooks and Crannies of XA

There are still many aspects of the XA protocol that haven't been mentioned in any significant detail yet, such as the 1PC optimization, details on enlistments and when XA end() happens, etc. I'm going to handle these in a rather grab-bag fashion here.

When does start() happen exactly? How 'bout end()?

The TM has pretty wide latitude in deciding when the call to end() happens on an XAResource. It can keep the XAresource "open" until txn. commit() happens if it wishes, but it can also call end() at other points as well, such as when close() is called on the connection/session. In fact, a TM can call start(), end(), start(), end() ... repeatedly on the same resource if it wants to.

Speaking of closing of connections and sessions - the application developer should understand that generally their application code is never accessing the "true" connection and session objects from their JMS provider, JDBC drivers and the like. Instead, the application servers that I know of return wrappers around these resources. The purpose of these wrappers is tie use of the resources into the Transaction Manager transparently. For example, upon opening a JMS session, the wrapper objects may enlist the underlying JMS Session with the Transaction Manager as a resource via a start() call. Likewise, the wrappers may call end() when the wrapped session is closed() (although, as mentioned, it also may not :-). For this reason it's important that your application code looks up the correct JNDI factories/Data Sources so that you are getting references to the correct wrapped objects. It's all too easy to get a reference to the wrong object - like the JMS provider's "real" connection factories, and thereby bypassing all of the application server transaction processing altogether.

The XA_RDONLY vote

In the course of transactional events, a global transaction may end up with a result where one or more XAResources haven't actually changed anything. For example, the application code may be organized so that under some conditions it is doing a JDBC insert, update, or delete, and yet under other conditions only a select may occur. In the latter case, the JDBC driver maybe enlisted in the transaction, and yet no transactional work has been accomplished. As it turns out, the XA protocol allows an optimization in this case. As the **javadoc for XAResource prepare() states** for the return code from prepare():

A value indicating the resource manager's vote on the outcome of the transaction. The possible values are: XA_RDONLY or XA_OK. If the resource manager wants to roll back the transaction, it should do so by raising an appropriate XAException in the prepare method.

What this means is that if no state change was affected within the XAResource during the course of a transaction, it is legal for that resource to return XA_RDONLY from prepare(). If an XAResource does this, the Transaction Manager should skip the resource during the commit() cycle - for the simple reason that there's no changes to be committed. As you may guess from this, this *can* mean that no disk forcing happens for this XAResource, which saves us 2 disk forces and unnecessary commit() calls. Even more importantly, this can be one of the triggers for the 1PC optimization...

The 1PC Optimization...

A nice little feature in the XA protocol can be found in the "1PC Optimization". For a variety of reasons, it may come to pass that a transaction is being processed in an XA fashion, but the Transaction Manager finds that when 2PC time comes rolling around, only 1 XAResource has actually done any useful work. Here are some reasons why this can happen:

- Only one transaction ever got enlisted. There are a number of combinations of configuration and/or application logic that can lead to this condition.
- Every resource but one returned XA_RDONLY from its prepare() call

The 1PC optimization itself is very simple: the Transaction Manager skips writing a "Committing..." record, it skips the prepare() call, and calls commit() on the lone XA Resource directly. Specifically, the signature for commit() is:

```
public void commit(Xid xid, boolean onePhase)
```

If the paramter *onePhase* is true, then the Transaction Manager is signalling to the XAResource that it is using the 1PC optimization, and that this commit is no longer part of a global transaction, but a local one.

The nice thing about the 1PC optimization is that a container and application can be coded to always use XA, and if only 1 resource gets used the application will only pay the cost of a local transaction (plus a very small overhead for things like generating and processing the Xid). You can see this from the disk forcing for 1PC vs. regular 2PC operations:

- 1 XAResource: 1 disk force (the 1PC optimization)
- 2 XAResources: 5 disk forces
- 3 XAResources: 7 disk forces
- .....

The reason only 1 disk force is required is because this transaction truly gets processed in a local-transaction manner - not only do we not need a prepare() phase (or maybe have a number of low-cost prepares which returned XA_RDONLY), but the TM need not durably record anything either. The last bit may be a bit of a mind bender if you're not doing transactional internals on a regular basis. You see, in the case where either only one XA Resource exists, or all resources but one has returned from prepare() with XA_RDONLY, no information has been durably recorded by any XAResource until the single *commit (Xid, true)* call is made, and in fact there is no distributed transaction state, only the state of the one lone XAResource. Since that's true, we can safely avoid the TM disk force of "Committing..." record. In the worst case where the TM fails just before calling *commit (Xid, true)*, no durable prepare() information is recorded anywhere, so the lone XAResource can either detect the failure of the TM and rollback, or timeout the transaction.

Doing XA w/out an XA driver: the Last-Resource Gambit

Some application servers have done a bit of hacking and support what I call the "Last Resource Gambit". Here's how it works. Let's say you've got a bunch of stuff which you want to perform transactionally via XA, and all your components are fine and XA compliant - exception one. It

may be some crusty old MOM system, some pre-JCA ERP system, or something that Joe the Sysop did on the side, but in any case you really want to do XA and there's just this one component standing in your way. The Last Resource Gambit lets you get away with it.

How it works is like this. The TM orders all of the resources so that the non-XA resource is always "last" on its list. Assuming we get through the app code successfully, it then enters the 2PC cycle and calls prepare() on all XA Resources. Obviously, the non-XA guy won't be part of this. If they all succeed, it then calls commit() on the non-XA guy. If this succeeds, the remaining XA resources are committed. If it fails, everything is rolled back. This little trick works because we only call commit() on the non-XA guy only if all the rest of the 2PC machinery votes yes, and if the non-XA guy effectively votes no by failing on the commit, we can still rollback the prepared XA resources.

The one gotcha, of course, is that this can only work with a single non-XA resource. If you have 2 or more, it just doesn't work.

More on Disk Forcing

Part I of the series gave an overview of why disk forcing plays such a large role in XA processing time, but admittedly it didn't go into an enormous amount of detail on why that's so. Here in Part II, I'll go into a bit more detail to explain the situation more clearly.
Let's start by looking at a plain old RDBMS transaction - let's call it a PORT - vs. an XA transaction with two resources.

In the PORT, we can get away with 1 disk force (or maybe a fancy checkpointing system if that's your thing).

In the XA example scenario we have:

- 2 disk forces at prepare(), that's 1 per XAResource
- 1 disk force by TM to signal "Committing..."
- 2 disk forces at commit(), that's 1 per XAResource

That's 5 disk forces in XA vs. 1 disk force for a single database transaction, and the fact of the matter is that disk forces tend to be expensive. It may turn out that you're doing big SQL work somewhere that takes 500 milliseconds or something, in which case the disk forcing will be lost as noise by the actual transactional work (who cares about 10 or 20 milliseconds for forcing when you've got a 500 milli query?).

But if your doing small queries, the disk forces can stand out as the _majority_ of your transaction time. I've seen many systems doing XA which do a quick select, an update or two, and a few JMS publishes. Overall time for the "work" here might be only 50 milliseconds. Where as the disk forcing might eat 100 milliseconds.

For some slightly anecdotal evidence, I'll shun modesty for a moment and discuss my own XA transaction log implementation, and talk about some of the numbers I've seen in my own testing.
In my own XA work, my transaction log implementation is pretty well optimized. It uses all NIO and double fixed sized logs, and a typical transaction only takes up a few hundred bytes of disk space. The logging code itself is all hand-tuned and hand written to get bytes out fast - no serialization or other automation. Forces are batched together in a tunable manner to amortize the disk force cost over as many concurrent transactions as possible. The tunable bit allows me to control how many transactions can get amortized at once and optional delays, which lets me throttle access to the disk so that I don't have one process dominate it too much.

With that kind of code, non-forcing I/O has shown to be inconsequential - it's well under a millisecond. But the forces, well, they carry alot of force. I've tried out my system on a variety of disks, from garden variety disks that come with workstations all the way up to massive EMC RAID arrays with mega (actually, giga :-) RAM caches. On the worst disks, I see forced I/O times around 30-50 milliseconds. On the massive and hideously expensive (but worth every penny!) EMC arrays, I see disk force times between 5 and 10 milliseconds. Variations happen on the EMC side because these mothers are big enough that they have to be shared among systems and I can never test in 100% isolation.

With really small message payloads and reasonable concurrent publishing (say, 20 publishers going full out) under the best conditions I can concoct, a single server can get about 250 transactions/second. CPU utilization in these tests stays under 50%, while disk I/O's show around 175 IO ops/sec. As you can see, the EMC I/O is fast enough that it's hard to pull much concurrency out of it - it's tough to find multiple transactions to share a disk commit, but I might be able to get better throughput if I look at many tests carefully.

I also have a magic hidden option, which I refuse to divulge to outsiders in detail, which lets me turn off disk forcing. This is used mainly to take the disk out of the equation when we need to for certain types of test, and also to debug some capacity sharing issues we've had with the EMC getting hit by other systems and cache thrashing a bit. Anyway, turn off disk forcing and throughput jumps up to around 800 trans/second - with most of this time being damnable serialization on both ends of the network sucking up almost all of the CPU. Get rid of serialization, and you can easily get at least 1,500 trans/second without trying too hard. Getting rid of serialization w/out disk forcing drops CPU utilization from my previous "under 50%" to practically non-existent, but throughput stays pegged around 250 tran/second.

Based on the above testing, it seems pretty clear to me that disk forcing tends to overwhelm most other considerations when you're talking about XA. Put your transaction log on a faster disk and you will force faster. The same hold true with systems with better I/O channels and the like. Making XA faster almost invariably means making those forces faster, or alternatively reducing serialized access by using multiple disks. For the latter, you can "virtualize" this by using a RAID array, but this considering how far a RAID array is from Java code, it may be too low-level for you to see all that much benefit. An alternative solution, which I first seriously considered based on an e-mail from Greg Pavlik, is to use multiple log files, possibly with each on a different physical disk. I haven't done any serious testing on this idea, but conceptually a system which uses multiple logs on multiple disks may be able to extract much better parallelism out of its transaction system.

Be Careful Out There

As with many "enterprise" technologies out there today, when XA works it's all pretty simple, really. Where 90% of the effort has gone is in all those niggling error conditions, and in optimizating things so work reasonably well under heavy loads. For XA, when everyone plays nice together you get some super guarantees in global system consistency, at a fairly fixed time cost for the disk forcing. But what application programmers, architects, and managers need to be aware of is that not everyone plays nice together, and failures in an XA environment can be tangled skein to unwind. It's fairly well known in the industry that some RDBMS' XA drivers are full of problems (*cough* Oracle *cough). What's worse, many major JMS implementations switch off disk forcing altogether by default. This makes them look really, really good on

benchmarks, and makes you look really, really bad if Little Johnny pulls the plug on your JMS server at an inopportune moment. And the lack of clarity in the JTA specs has lead to TMs that are mass of spaghetti code, trying to deal with many esoteric XA errors each of which seem to pop up in at least one of the XAResource implementations out there.

And even when all the vendors get it right, I think this article has shown that error recovery in XA can be an arduous task - and harrowing for the operations people to boot. But don't dismiss XA outright - for all the faults, the benefits are quite real. But it pays to know what you're really buying into, and hopefully this article has brought a few people a small step closer to what's really going on under the hood, and how it can affect them.

Well, that's about it for Part II. Sorry I didn't have time to go into the size of Presumed Abort's disk force, and left you dangling with the whole Nick and Jessica and TickleEmphasneeze. Perhaps, if the gods are smiling on me, I'll have the extra cycles to address those serious issues in *Part III: The XA Implementors Notebook*.

April 03, 2004 02:04 PM EST **Permalink**

Comments [4]

XA Exposed, Part III: The Implementor's Notebook

Welcome one and all to *XA Exposed III: The Implementor's Notebook*. This is part III in my series on describing the internals of the XA protocol in a J2EE environment. If you're a first timer here, I strongly recommend reading **Part I** and **Part II** first before diving into Part III. Feel free to ignore my advice, of course, but if you do so don't go asking any questions that were already covered in the earlier parts :-).

Part I covered the basics of the XA within the J2EE world - we met Mr. Xid, Ms. XAResource, and got them rolling and all hot and bothered in a sample XA lifecycle.

Part II left the basics behind, and dived right into various failure and recovery scenarios. Additionally, Part II rounded out the first article with various efflusia like the 1PC optimization and heuristic decisions. We also got to meet Joe Sysop, took a tour through a Saunders-inspired failure scenario, and came away from the experience knowing that dealing with heuristics can be tough, and to never let Satan go to the bathroom.

Now, in *XA Exposed Part III: The Implementor's Notebook*, the focus is shifted towards actual implementation issues. I'll be looking at some of the major concerns, from a reliable XA viewpoint, that exist in creating a JTA Transaction Manager. This mostly centers around how Xid's are created, and the all-important issue of writing a recoverable transaction log. I was hoping to cover XAResource implementation as well, but the material required just to cover the basics on the transaction manager side turned out to be overwhelming. As such, there may very well be an XA Exposed Part IV sometime in the future.

For those interested in my own personal motivations in tackling this subject, my goals behind this are two-fold. First, it may be of some use to somebody out there who's actually implementing this stuff. Who knows, maybe even various TM implementations will actually agree on major semantics :-) Secondly, I believe that knowing the guts behind implementations is quite important for application developers. Having an accurate mental model of what goes on behind the scenes can help a plain old developer to plan better, understand costs behind their decisions, and understand what the real issues are when you get into esoteric failure and recovery conditions.

As is always the case, I am only a flawed human, and as a result I do make mistakes from time to time (or so my wife tells me). If you see any such mistakes, or take issue with any points contained herein, please let me know.

References

As was the case with the past installations of *XA Exposed*, there are alot of niggling details to cover, and I couldn't possibly work all of this out on my own. So, of course, I turned to the Internet to give me a helping hand. The following links and resources were of great use to me in putting this article together, and you may find them useful as well to fill out the background behind this discussion.

- **The HOWL Project**
  During the time of XA Exposed Part II, Dain Sundstrom brought the HOWL Project to my attention. The primary purpose of HOWL is to create a high performance journal system for use by ObjectWeb's Transaction Manager JOTM. And JOTM, in turn, will be an optional plug-in component to Geronimo. I previously wrote an article **expressing my displeasure with HOWL**, but I'm happy to report that the HOWLers appear to have mostly righted their ship and now appear to be on course to creating the core of a nice system that can be used for transaction logging (among other things). HOWL is still in the early planning stages, but the mailing list contains alot of meat about transaction logging, and it should be interesting to watch the CVS tree as the project progresses. NOTE: I am semi-active on the developer's mailing list, so I suppose I cannot be considered entirely objective on this.
- **The Geronimo Source**
  Developers have a rare opportunity to see an application server's code (and of particular interest to me, a TM) literally as it is being built. Geronimo is of course still in the very early development stages, but it's fascinating to take occasional peeks at the source of the Next Big Opensource App Server. The Transaction manager stuff can be found under modules/transaction, although the reliable TM implementation will come from JOTM, as mentioned (nobody seems to do a transaction log demselves!).
- **JOTM** JOTM is the pluggable and stand alone Transaction Manager used by Jonas, and someday soon to be used by Geronimo as well. This source is pretty cool to look at as well, but it won't have transaction logging until HOWL is done.
- **JBoss** Surprise surprise, even JBoss has a transaction manager. As is the case with all of the open source solutions today, they do not have transaction logging, but otherwise the source is an interesting read.

The Transaction Manager

As I've mentioned in the past, the X/Open XA and JTA specifications, taken together as a whole, do a mediocre job of specifying just what a Transaction Manager's roles and responsibilities are. The gist is conveyed fairly well, but many important niggling details are left pretty much

hanging up in the air. Or, occasionally, some details may be decently described for XA, but go unmentioned in the JTA spec - and a manual translation from old C XA stuff to a multithreaded Java J2EE environment pretty much mangles any possible translation. The goal of the Transaction Manager section of this article is to try to pin down some of the major behaviors of a TM, and to outline major design goals and possible solutions. This is *not*, in any way-shape-or-form, an attempt to fully define a JTA-compliant transaction manager, and additionally non-XA bits will be pretty much completely ignored by me. This is only an outline of what I think are a couple of the major XA-oriented design points for a JTA transaction manager. If you want to see a full treatment of a TM in all its glory, check out the full source of JOTM, Geronimo, and/or JBoss in their respectve CVS repositories. And maybe consider buying a few cases of beer for the major authors involved therein :-)

Xid Management

One of the most basic things a Transaction Manager has to do is generate Xid's. You may recall from *XA Exposed Part I* that an Xid is the fundamental transaction identifier used in both the JTA and the XA specification. This identifier is made up of three parts:

1. The format ID. This identifies the format being used within the Global Transaction ID and Branch Qualifier byte arrays.
2. The Global Transaction ID. As per the XA specification, this ID must be *globally unique*
3. The Branch Qualifier. Each XAResource participating in a global transaction gets an Xid containing the same Global Transaction ID, and a unique branch qualifier that applies to just that resource.

When you're considering a strategy for generating Xid's, you have to keep several things in mind:

1. Uniqueness really is important. The Xid really has to be as unique as you can make it - across multiple TMs (possibly different vendor TMs), multiple TM processes, and the like.
2. A TM must be able to recognize it's own flavor of Xids. More accurately, it needs to distinguish its own Xids from Xids owned from another TM. This is critical for XA's peculiar brand of presumed abort to function properly. Since XA presumed abort means that the TM only disk forces post-prepare(), pre-commit() time, it may be seeing a number of Xid's which it does not recognize. Some of these may be the TM's own transactions, which were generated before prepare() was complete. And some of them may be owned by another server in a cluster of TM's, or owned by another manufacturer's TM altogether (e.g. some other group may be sharing your database and using a completely different TM to manage it).
3. To further qualify point #1, uniqueness means unique even across TM crashes and restarts. And, ideally, you want uniqueness on the cheap. It would really suck to use an RDBMS to get unique IDs through a sequence generator - you'd be adding even more to the already high cost of XA transactions.
4. A TM may be dealing with an Xid which it doesn't own. Consider a case of clustered application servers, where the vendor supports distributed transactions across the cluster (e.g. a transaction started on server 1 involving some EJBs may invoke an EJB on server 2, with the whole kit 'n' kaboodle operating under one transaction). In this sort of scenario, server 2 is going to receive a global transaction ID for which it must generate branch qualifiers - branch qualifiers which have to be unique across the entire cluster. And, typically, you don't want servers in a cluster coordinating this sort of activity on a regular basis - you don't want server 2 going to server 1 (let's call server 1 "Joe"), "Hey, Joe, I need 2 unique branch qualifiers here!". Instead, you want each server to be able to generate a unique branch all on its own without having to consult other members of the cluster, or some master source.
5. A TM has 64 bytes to work with on the global transaction ID, and 64 more bytes for the branch qualifier. Whatever scheme you use to generate Xid's, it all has to fit in this size.

Now, as it turns out, the XA specification has a little to say on Xid generation (as usual, the JTA has nothing to say at all). On the XA specification side, it strongly recommends that TMs use "OSI CCR atomic action identifiers". From the XA specification, section 4.2:

> The only requirement is that both gtrid and bqual, taken together, must be globally unique. The recommended way of achieving global uniqueness is to use the naming rules specified for OSI CCR atomic action identifiers (see the referenced OSI CCR specification). If OSI CCR naming is used, then the XID?s formatID element should be set to 0; if some other format is used, then the formatID element should be greater than 0. A value of -1 in formatID means that the XID is null.

I don't know about you, but personally I don't have the slightest clue what an "OSI CCR atomic action identifier" is, nor am I inclined to pay ISO the required 142.00 Swiss Francs (~113 USD) for the spec to find out.

Fortunately for me, apparently none of the TM vendors in the world wanted to, either. From what I have found out in the field, nobody seems to use this strongly recommended standard, and instead each vendor rolls their own Xid format. Specifically, each vendor picks their own format ID, does some spying around the industry to make sure that no one else is using that ID, and then encodes the global transaction ID and branch qualifier byte arrays however they want.

For me personally, given the requirements that we have in hand, this is how I would go about generating Xid's. In outline form, I'd generate Xid's in the following manner:

- formatID: Vendor Unique ID
- global TID: [unique server ID] [millitime] [seq number]
- global TID: [unique server ID] [millitime] [seq number]

The following sections will explain the various bits.

The FormatID

Pick a format ID that no one is using. The best approach I've seen is the one Geronimo is using - encode part of your TM name into the format ID. Their is captured in the code:

```
  private static int FORMAT_ID = 0x4765526f; // Gero
```

Yes, they use the ASCII code for the first four letters of Geronimo. The vaporware Pyrasun Transaction Manager, coming to an IT enterprise datacenter near you in the year 2047, will likely use 0x4B726973 - so hands off!

With the format ID code in hand, you've just guaranteed your Xid's are unique from any other TM product's Xids. Now you just have to distinguish your TM process from other TM processes in the cluster.

Get a Unique ID per TM server process

Distinguishing your processes' Xids from those of another process is conceptually simple - you use a unique ID for your process that's different from every other server in the cluster. However, as anyone who's ever done clustering software knows, it's often difficult to come up with truly unique ID. Generally, speaking there are two approaches here. Approach one is to autogenerate the ID. Most often, this involves the IP address or DNS name of the machine the server is running on, possibly including a listening port number. The second approach is to use a logical server name which is set via configuration. There are plusses and minuses to each. The autogeneration feature is attractive, in that the user (that's you!) doesn't have to configure anything to use it. The downside is that they rarely work very well, and are prone to collisions. Geronimo uses the machine's raw IP address (although they say they will add in configurable support in the future). JBoss does a similar thing, except they use the DNS name, with a "setGlobalIdNumber" option to override this (I don't know if this is actually used right now). JOTM seems to take a middle road - it uses the IP address in conjunction with "serverName" (I'm not sure what the serverName actually is).

The configured logical name approach is a pain, in that you have to go out and set all of your servers names. However, assuming you are competent to assign all unique names, this approach is bulletproof. The problem with generated server IDs is that quite often they collide, or cause recovery problems. In the Geronimo world, if you run two Geronimo servers on one box, their Xid's could conceivably collide (because they share IP addresses). Ditto for JBoss, unless the *setGLobalIdNumber* method is actually used. This approach also has problems with recovery, a problem which JOTM may share by encoding the IP address (I say may because I don't know how this IP is generated). Here's the problem: let's say your server gets fried, but you were smart and used a shared RAID array. You bring up a new process on an alternate machine that shares this array, and you find that you can't recover the in-flight transactions. Why? Because most likely this alternate machine will have a different DNS name or IP address. Such approaches require that the TM always be run on the same IP address or DNS name, which can require complex Virtual IP Address solutions to affect recovery.

Whichever you choose, auto-generation, configured logical ID, or some combination thereof, you have to make sure that you use this server ID when you generate the global ID, *and* when you generate branch qualifiers. The reason you need it in the branch qualifier? Because the global ID may not have come from you, but from some other TM on the cluster. You need to use your own ID in the branch qualifier to make sure that your branch IDs don't collide with another server's which is participating in the same global transaction.

Generating Unique Sequence Numbers for Global ID and Branch Qualifier

OK, here's where we are so far. We've got our unique format ID firmly in hand. We've used some method to ensure that each process has its own unique server ID (either through a crappy auto-generation technique, or the proven configuration logical ID approach). Now, we have to start generating unique IDs per server process.

My approach to this problem is to use a combination of time and a sequence number. In pseudo-code form, it looks like this (with the unique server ID thrown in for good measure):

```
byte uniqueSequence[64] = new byte[64];
String serverID = getServerID();
long part1 = System.currentTimeMillis();
long part2 = getNextSequenceNumber();
appendArrayBytes (uniqueSequence, serverID);
appendArrayBytes (uniqueSequence, part1);
appendArrayBytes (uniqueSequence, part2);
```

In the above, I assume "getServerID()" returns your unique per-process serverID, that getNextSequenceNumber() is a sequence number generator that stars at 0 when the server starts and increases by 1 with each call, and that appendArrayBytes is a magic function which appends various variables onto the end of the passed in byte array, uniqueSequence.

The big advantage of this approach is that you'll get unique IDs even after server restarts, without having any kind of central sequence number authority. The key is using System.currentTimeMillis() - you're tying the Xid to the time it was generated. There are, however, two downsides to this approach:

1. It could potentially wrap. For whatever your resolution is for System.currentTimeMillis(), you can only allow 2 trillion or so transactions during that time period. In practical terms, your system is limited to doing only several tens of trillions of transactions per second.
2. If you care about doing recovery on alternate boxes, you must take care that your clocks are reasonably in-sync. You don't need millisecond synchronization, but they should be close enough so that by the time the server is started on the alternate machine, your clock reads sometime after when the original process went down on the original box.

To bring things back full-circle, this is what where we began:

- formatID: Vendor Unique ID
- global TID: [unique server ID] [millitime] [seq number]
- global TID: [unique server ID] [millitime] [seq number]

You use a unique formatID to get uniqueness across TM vendors. You use a unique per-TM-process ID to get uniqueness per process. And you use the current time in millis plus a sequence number to get uniqueness within a process.

The TM Transaction Log

I've touched upon TM Transaction logging a bit in both Part I and Part II of this series. Here in Part III, I'm going to expand upon those efforts and get more into the specifics. You can look at this section as a design outline for a Transaction Manager's transaction log.

It should be noted that the design presented here is only one way to approach this problem out of many possibilities. I believe this particular design makes some pretty good trade offs and takes full advantage of some of the peculiarities of TM XA logging, but don't consider it the be-all, end-all approach.

Part II of the XA Exposed series went into great detail about presumed abort, the fact that a TM only needs to force a "COMMITTING..." record, and the costs of disk forcing. Some might say the detail was so great that I might have an obsession over this :-) Well, perhaps there's a grain of truth in that, but I'm going to leverage that obsession for my TM design. Bits and pieces of this material have been posted here and there before, but I'm going to bring all of that scattered information into focus in one spot here. For those interested in the original discussions, check out the TSS threads in the References section, and also the January 2004 portion of the HOWL mailing list. In fact, right after I wrote my first draft of this section I became involved with the HOWL list. Pieces from the draft of this article directly applied to what was being discussed there, and those mailing list discussions, in turn, suggested changes to my draft. Odd how synchronicity can just spontaneously happen in life at times.

In all, my TM Transaction log design relies on several interrelated key facts about XA logging, and also about Java I/O in general:

1. As you may have heard, the TM needs to only force a "COMMITTING..." record
2. The TM should also write a "DONE" record if it succeeds in committing. This record does not need to be forced
3. Both the "COMMITTING" and "DONE" records need global transaction ID recorded along with them
4. The only content needed in these records is the global transaction ID. By definition in the JTA and XA specs, the global transaction ID can be a maximum of 64 bytes in length. Hence, our "payload" in these records is quite small indeed.
5. On average, transactions are comparatively short lived. The time from writing a "COMMITTING..." record to writing a "DONE" record is typically measured in a few tens of milliseconds. However, at the same time, this is only an average. In certain failure scenarios, the distance between these records could be measured in seconds or minutes. Take into account also that a J2EE application may have a mixed transactional load, with varying XAResources being enlisted. With that in mind, a set of one flavor of transactions could be stuck in a "COMMITTING..." state if an XAResource experiences a failure, while other flavors which don't use those transactions could be completing normally and quickly. This means you have to support the concept of a dangling "COMMITTING..." record hanging around for an unusually long time.
6. You want to minimize unnecessary disk seeking - ideally, you'd like to eliminate unnecessary seeks entirely, since this can elongate your force times.
7. You have to be wary of transaction log corruption - the machine could fail with data being written out to disk. As such you want to avoid file system meta-data twiddling as much as possible (e.g. avoid adding blocks to a file, avoid creating new files, etc). This also means that your recovery reader should be able to detect partial records and "back up" to a last-known-good record.
8. You need to store sufficient information to reconstitute XAResources somewhere. However, that "somewhere" does not necessarily have to be within the transaction log. I'd suggest an alternate file for this.
9. You want to avoid unnecessary recovery scans on startup. This implies that you want to have a "normal shutdown" piece of code which indicates that the transaction logs do not have to be read at startup.

Taking all of the above into account, my own solution is to use dual fixed-size, pre-initialized log files which use a write-forward-only strategy. Here's the idea in outline form:

- You have two files which represent your transaction log. You pre-initialize these with some reasonable byte pattern at startup if they don't exist. You want to pick a byte pattern that can aid you in detecting corruption - I recommend using the initials of a hated ex-significant-other who did you wrong sometime in the past. That way you can directly associate their initials with corruption :-)
- You may want to smack a header at the front of these files so you can recognize that they're proper transaction logs, perhaps with version information as well. This further aids in corruption detection, and avoids trying to use someone's recipe list as your transaction log.
- The log is used in a write-forward-only manner. You start at the beginning, sans any header, and as records come into your logging system you write them forward only. Don't try to delete records, or seek back to an entry to change its results. Instead, write forward only. This means a recovery reader has to read the log in order, and that for related records they need to "replay" as if they are a state machine.
- Use a third file as an indicator as to which file is active, which is passive at the moment. I'll call this the *indicator file* This file only needs to hold a single byte - the number of the active log (1 or 2).
- Dealing with a full file - it's log switch time!:
    - When the primary log is full, you need to pull a log switch. This requires either some cooperation from the TM, or knowledge within the logger as to what records are "live and dangling" and which are completed out. You need to pick one of these approaches because a log switch means you have to copy dangling records from the old log to the new, which implies that someone needs to track danglers (preferably in memory - you don't want to log scan on every switch :-0).
    - The copy simply copies the dangling records to the passive log, right at the top. Note that this may overwrite old data, but that's OK.
    - When the copy is done, do a disk force on the passive log.
    - When the above disk force is done, update the indicator file and force.

  The ordering of the above is important - if you crash anywhere along the way, no data has been lost. Do it out of order and you can mangle something, or lose data along the chain.

  A critical assumption behind this approach is that the total number of in-flight transactions at any given time will be fairly small. Since the algorithm involves copying records which are still "live", the time for a switch is somewhat proportional to the number of usch live records there are. I say "somewhat proportional" because disk forcing is still going to be a dominant factor, and TM XA records are tiny. This means that whether you're copying 3 records or 50 records, the disk force time is going to be much longer than the copy time, and in practice you won't notice a difference. Since JTA XA is generally slow, we don't expect many simultaneous in-flights, and the small records are also a big contributing factor to this design working. If either of these assumptions were removed - say, XA logging could be done asynchronously w/ completion callbacks possibly coming back to the TM at a much later time, or if we needed to log

more information, then the copying operation could represent a significant spike in transaction durations. As it is, this design works well in the very specific JTA TM XA case, but may not be applicable to other problem domains outside of TM logging.
- As a first cut you may want to use fixed-sized records. This makes coding *alot* easier, and can actually be somewhere more time efficient. But you will waste some disk space. But, keep in mind that you could use, say, a 200-byte fixed sized records, and have 136 bytes for your bookkeeping purposes (200 - the maximum global transaction ID size of 64).
- A really paranoid person might want to run a check sum over the record and record it in the header.
- In general, for records I would use a format like this:

```
[RECORD_TYPE] [RECORD_LEN] [HEADER_LEN] [System.currentTimeMillis] [Sequence number]
[Checksum] [Payload] [END_RECORD_INDICATOR]
```

Where [RECORD_TYPE] is a passed-in record type from the TM. [RECORD_LEN] is the overall record length (sans [RECORD_TYPE and [RECORD_LEN]). [HEADER_LEN] is the length of the remainder of the header - important if you want to support easy upgrades of your format. The remaining pieces are the rest of the header, and the payload. The header at least should have [System. currentTimeMillis] and [Sequence number], with the [sequence number] coming from some monotically increasing sequence generator unique to the process. The [checksum] is optional for the paranoid among us. The time information can be *very* useful for profiling and tracking down problems in production, and in conjunction with the sequence number it can give you precise ordering. This doesn't give you much in this solution, but can be priceless if you ever move to a system with multiple dual log file pairs to lessen single-threading on a single log file pair. Finally, I like having an [END_RECORD_INDICATOR] as an extra corruption detector device - I'm a suspenders and belt kind of guy. Actually, the END_RECORD_INDICATOR and [RECORD_LEN] in conjunction are very useful in development, as well, to catch programming mistakes in the log system early.
- You've *got to* batch disk forces. Given how slow disk forces are, if you don't batch you're going to max out your transaction rate between 50 and 100 transactions/second.
- Avoid Java serialization at all costs - hand code your format. This will fantastically lessen the CPU cost for writing out log records, and reading them in later.

That really is all there is to it. You have individual binary records, possibly fixed sized written, out in a write-forward fashion, with batch disk forces "hardening" it all for you at the request of the TM. When a log fills up, you pull a switcheroo and swap the passive and active log files. The *log indicator* file in this case really is optional - there are other ways to do the same thing, but I find it the simplest solution which doesn't allow for errors to creep in. This gives you a basic transaction log. Overall, this solution has several benefits:

- Pre-initialization of the files can boost your performance a bit. You don't have to wait around for the OS to repeatedly extend the file size.
- Pre-initialization also means that your logger can't scramble the file system if your machine goes down in the middle of a logging operation - all of the file system work of organizing the file is done in advance.
- It's pretty fast. By using a write-forward strategy and utilizing batch-forces, disk head seeking is minimized and quite good throughput results can be obtained.
- It has minimal complexity (note that I say *minimal complexity*, not *no* complexity :-)

Layered on top of the basic design, you of course also need to store your XAResource reconstitution information. And you need an indicator for safe shutdown to avoid unnecessary recovery scans at startup. In the interests of conserving article space and my own time, I'll leave those as an exercise for the student.

On recovery scans, this work required by this design is pretty simple. On startup, you consult the *log indicator* to find the active log. You (hopefully) verify that the file is really a transaction log, then start reading from the beginning and work your way forward. With this sort of design, you have to read the whole log to figure out what happened - but that's why we're using fixed sized logs. I recommend a size of 1MB-5MB or so - this seems to work very well in practice. The recovery scan itself, I find, is best done in a callback "event" manner - you pass in the log system a callback object, get it to initiate the recovery scan, and then get called back on each record as its read. Way up in the TM, it parses these records, divvying them up into "COMMITTING..." and "DONE" records, using a Map or something to track these via Xid. On "COMMITTING", you add it to the map, on "DONE" you remove it from the Map. When the scan is done, what's left are potentially dangling "COMMITTING" records.

Variations on the Base Logging Design, and Other Alternatives

As I mentioned in the intro to the logging design section, the described mechanism isn't the only way to approach logging - not by a long shot. There are many variations that can be added to the base design, and in addition there are entirely different strategies that could be used. I'll briefly describe here some of the possibilities (well, briefly for _me_).

Multiple Log Pairs

An obvious optimization to the base design, which I touched upon briefly, is to eliminate the single threading on one transaction log pair, and use multiple transaction log pairs. This can allow users to possibly put different pairs on different disks, boosting I/O rates overall. Even if they're on the same disk, though, eliminating single-threaded contention for the logs at the Java level may still be a win. Such a scheme could use a simple round-robin strategy to distribute transactions over the available set of log file pairs.

Of course, for this to work reasonably well, I'd imagine you'd want a "sticky bit" so that related records go to the same physical dual log set e.g. enforce "COMMITTING" and "DONE" records for a given Xid to go to the same physical pair. This would make the higher level TM event-driven recovery scanner work identically with one log set or many. However, in this scenario the TM (or maybe the log system?) would want to do ordering before finding final dangling "COMMITTING.." records - you'd generally want to try to commit things in the order they came into the TM, not necessarily in the order they were distributed in various files. This is where the [System.currentTimeMillis] and [sequence

number] in the record headers come in - you can use these two bits of information to reliably order the transactions properly.

Ditching the Indicator File

Using such the active log indicator file mentioned in the base design is not the only possible approach - indeed, some people seem to find the idea outright repulsive. An alternative is to embed this information in the transaction log files themselves. You have to be careful with such an approach - you have to make sure that your design is robust enough to stand a total machine failure at any point during a log-switching event and still be correct. The best approach that I've seen for this is to use timestamps in the headers of the transaction logs themselves to indicate which was active. At initialization time, you write a long into the header containing *System.currentTimeMillis*. At log-switch time, the last step of the log switch would be to write *System.currentTimeMillis* in the alternate log file. In such a scheme, the log with the later timestamp in the header would be the active one. A huge advantage to this approach is that it avoids the extra indicator file disk force at switch time.

Using a Single Circular Log File

Some people dislike the dual log file approach because of the "stall" at log-switch time. One solution which avoids this is to use a single, logically circular log file instead. In this scenario, you have just one log file which wraps around from the end back to the beginning. The logical start and end of the file, from a user's perspective, moves as information gets logged, and when the end of the physical file is reached, you wrap back to the physical beginning of the file.

A big advantage of this approach is that, logically speaking, there is no "end of file", and the user is not forced to do a copy in such a situation. However, TANSTAAFL - as with any other complex program, no solution is universally perfect. In this case, the issue with a circular log file is that it has to worry about overwriting data that is still live. For example, you could end up with failure scenarios where certain transactions are "stuck" for an unusual amount of time, and meanwhile other transactions are still flowing normally. In this sort of situation, over time you might wrap around and eventually overwrite the live record. The general solution to this is for the user of the log system to regularly write out all of the "live" transactions - you can think of this as a sort of "refresh" to make sure that live transaction information always stays relatively close to the logical end of the circular file. This works somewhat like a checkpoint system.

The astute reader will, of course, note that this periodic refresh looks and acts suspiciously alot like what happens at log switch time. The only practical difference between copying in the two is that in the dual log scenario, copying is forced when you reach end of file, where as in the circular system copying happens at the direction of the user of the log (usually via a timer, or a count of the number of records logged). In addition, at recovery time the circular solution requires code to find the logical start of file (or, alternatively, find the checkpoint-like refresh record and scan from there to logical end).

In my experience, the performance of this approach is pretty much the same as it is with a dual log system. The reason I don't advocate it myself is the added complexity that it involves - it does about the same thing in the same amount of time, but it requires more complex code to do it. With that said, circular logs (and also general checkpointing solutions) do have their uses. For example, checkpointing is a much better fit for traditional RDBMS' transaction log than dual fixed sized logs would be, due to the nature of the work involved. April 03, 2004 02:04 PM EST **Permalink**

Comments [3]

PyraLog v0.1 Alpha 2

For the 7 or so people in the world who care, I've uploaded version 0.1 Alpha 2 of the PyraLog source and related files. **You can get the download here**.

NOTE: my ftp server is slower than frozen sludge at the moment, so it might be awhile for the 7 or so of you to download the gzipped tarball package. :-/

NOTE2: OK, the ftp server is back and sane again.

This version implements an on-demand pool of NIO Buffers so there are no buffer overflow issues, changes the XA side of the house to use the logger's built-in record keys to reduce the size of records being forced, and has a number of tweaks for performance and for reducing internal lock contention. There's also some tiny stats-gathering code built in whose concept I unashamedly stole from Michael Giroux. Sample output looks like this:

```
     [java] All 1100 driver threads running
     [java] XADriver finished.  Total TPS is a whopping 10000 transactions/second (220000
transactions in 22 seconds)
     [java] Avg transaction/second per thread: 9 tps.
     [java] Avg transaction duration: 111 milliseconds (extrapolated).
     [java] Log Statistics:
     [java] ---------------

     [java] ----- xalogs/xalog_gaggle_0.tlog -----
     [java]     Num Batches (forces):         432
     [java]     Total bytes forced:           37620432
     [java]     Total # of force requestors:  219987
     [java]     Avg. Requests/Batch:          509
     [java]     Avg. Bytes/Batch              87084
     [java]     Avg. Disk Force Time (millis) 46

     [java]     Max. Requests/Batch:          1019
     [java]     Max. Bytes/Batch              2370034
     [java]     Max. Disk Force Time (millis) 187
```

```
     [java]     Min. Requests/Batch:          1
     [java]     Min. Bytes/Batch              55
     [java]     Min. Disk Force Time (millis) 15


     [java] ----- xalogs/xalog_gaggle_1.tlog -----
     [java]     Num Batches (forces):         0
     [java]     Total bytes forced:           0
     [java]     Total # of force requestors:  0
     [java]     Avg. Requests/Batch:          -1
     [java]     Avg. Bytes/Batch              -1
     [java]     Avg. Disk Force Time (millis) -1

     [java]     Max. Requests/Batch:          0
     [java]     Max. Bytes/Batch              0
     [java]     Max. Disk Force Time (millis) 0

     [java]     Min. Requests/Batch:          -1
     [java]     Min. Bytes/Batch              -1
     [java]     Min. Disk Force Time (millis) -1
```

The above sample shows the logger going full bore with 1,100 threads going as fast as possible. This shows great throughput - 10,000 transactions per second (or 20,000 log entries per second), and fairly abysmal latency (avg. 111 milliseconds latency per transaction). This sample also maxes out the CPU on my 1CPU 2.66MHz system. To show you something a bit more reasonable, here's the same sort of test but with only 200 threads:

```
     [java] All 200 driver threads running
     [java] XADriver finished.  Total TPS is a whopping 3703 transactions/second (100000
transactions in 27 seconds)
     [java] Avg transaction/second per thread: 18 tps.
     [java] Avg transaction duration: 55 milliseconds (extrapolated).
     [java] Log Statistics:
     [java] ---------------

     [java] ----- xalogs/xalog_gaggle_0.tlog -----
     [java]     Num Batches (forces):         934
     [java]     Total bytes forced:           17100934
     [java]     Total # of force requestors:  99632
     [java]     Avg. Requests/Batch:          106
     [java]     Avg. Bytes/Batch              18309
     [java]     Avg. Disk Force Time (millis) 27

     [java]     Max. Requests/Batch:          200
     [java]     Max. Bytes/Batch              507703
     [java]     Max. Disk Force Time (millis) 93

     [java]     Min. Requests/Batch:          1
     [java]     Min. Bytes/Batch              109
     [java]     Min. Disk Force Time (millis) 15


     [java] ----- xalogs/xalog_gaggle_1.tlog -----
     [java]     Num Batches (forces):         0
     [java]     Total bytes forced:           0
     [java]     Total # of force requestors:  0
     [java]     Avg. Requests/Batch:          -1
     [java]     Avg. Bytes/Batch              -1
     [java]     Avg. Disk Force Time (millis) -1

     [java]     Max. Requests/Batch:          0
     [java]     Max. Bytes/Batch              0
     [java]     Max. Disk Force Time (millis) 0

     [java]     Min. Requests/Batch:          -1
     [java]     Min. Bytes/Batch              -1
     [java]     Min. Disk Force Time (millis) -1
```

This test gives us around 3,700 TPS total throughput, with average latency per transaction around 55 milliseconds, which is a bit more down to earth due to the more reasonable load of 200 simultaneous requestor threads. CPU usage in this case averages out to around 20%.

I happen to not have access to big-iron systems at the moment, for reasons which should be obvious to long-time readers, so I can't give you figures on multi-CPU machines or systems with super fast disks.

My numbers seem to be in the same ballpark as the logging work being done by Michael Giroux on the HOWL project on similar types of

hardware. He's checked in a greatly revised logger into the HOWL CVS yesterday which is much more closely aligned to the XA logging requirements of JOTM, and is vaguely similar to mine own in a few ways (we both use NIO - and there ain't that many ways to use NIO :-).

Since interest in the PyraLog source has been pretty low, and HOWL appears to be moving forward on its own, this will likely be my final release of the PyraLog stuff. I've released this version mainly to get something out which is more complete and to get some closure. And who knows, maybe some Masters student somewhere will find it useful to his thesis, or some company will pay me $10MM for it out of a misunderstanding of the BSD-style license or some incredible fantasy version of corporate charity :-) If by some miracle someone wants to do anything serious with this, I might throw it up on sourceforge in my little cubby hole up there.

Many thanks and kudos to Michael Giroux, with whom I had an engaging and enlightening e-mail discussion on the finer points of binary logging and trying to wring the most out of the JVM. Michael helped make PyraLog better than it could've been, and now he's getting revenge on me by doing it even better himself in HOWL ;-)

Next up on the open source agenda will be EmberIO, an NIO library which automates handling of NIO sockets. This uses a pluggable worker pool, where you can configure the library on individual reader, writer, and processing workers, and automates all of the dispatching around NIO selectors. Initial development on this is about 75% done at this point, and I hope to get something out in the next week. And hopefully it will have a slightly wider appeal than transaction logging did :-)

April 03, 2004 02:02 PM EST **Permalink**

Comments [3]