Concurrency

by Martin Fowler and David Rice

Concurrency is one of the most tricky aspects of software development. Whenever you have multiple processes or threads manipulating the same data, you run into concurrency problems. Concurrency is hard to think about, since it's difficult to enumerate the possible scenarios that can get you into trouble. Whatever you do, there always seems to be something you miss. Furthermore concurrency is hard to test for. We are great fans of a large bevy of automated tests to act as a foundation for software development, but it's hard to get tests to give us the security we need for concurrency problems.

One of the great ironies of enterprise application development is that few branches of software development use concurrency more, yet worry about it less. The reason enterprise developers can get away with a naive view of concurrency is transaction managers. Transactions provide a framework that helps avoid many of the most tricky aspects of concurrency in an enterprise application setting. As long as you do all your manipulation of data within a transaction, no really bad things will happen to you.

Sadly this doesn't mean we can ignore concurrency problems completely. The primary reason for this is that there are many interactions with a system that cannot be placed within a single database transaction. This forces us to manage concurrency in situations where we have data that spans transactions. The term we use is **offline concurrency**: concurrency control for data that's manipulated during multiple database transactions.

The second area where concurrency rears its ugly head for enterprise developers is application server concurrency: supporting multiple threads in an application server system. We've spend much less time on this because dealing with it is much simpler, or you can use server platforms that deal with much of this for you.

Sadly, to understand these concurrency issues, you need to understand at least some of the general issues of concurrency. So we begin this chapter by going over these issues. We don't pretend that this chapter is a general treatment of concurrency in software development, to that we'd need at least a complete book. What this chapter does is introduce concurrency issues for enterprise application development. Once we've done that we'll introduce the patterns for handling offline concurrency and say our brief words on application server concurrency.

In much of this chapter we'll illustrate the ideas with examples from a topic that we hope you are very familiar with, the source code control systems used by teams to coordinate changes to a code base. We do this because it is relatively easy to understand as well as well as familiar. After all, if you aren't familiar with source code control systems, you really shouldn't be developing enterprise applications.

Concurrency Problems

To begin getting into concurrency, we'll start by going through the essential problems of concurrency. We call these the essential problems, because these are the fundamental problems that concurrency control systems try to prevent. They aren't all the problems of concurrency, because often the control mechanisms provide a new set of problems in their solution! But they do focus on the essential point of concurrency control.

Lost updates are the simplest idea to understand. Say Martin edits a file to make some changes to the `checkConcurrency` method - a task that takes a few minutes. While he's doing this David alters the `updateImportantParameter` method in the same file. David starts and finishes his alteration very quickly, so quickly that while he starts after Martin he finishes before Martin. This is unfortunate because when Martin read the file it didn't include David's update, so when Martin writes the file, it writes over the version that David updated and David's update is lost for ever.

An **inconsistent read** occurs when you read two things that were correct pieces of information, but they weren't correct at the same time. Say Martin wishes to know how many classes are in the concurrency package. The concurrency package contains two sub packages for locking and multiphase. Martin looks in the locking package and sees seven classes. At this point he gets a phone call from Roy on some abstruse question. While Martin's answering it David finishes dealing with that pesky bug in the four phase lock code and adds two classes to the locking package and adds three classes to the five that were in the multiphase package. Phone call over Martin now looks in the multiphase package to see how many classes there are and sees eight, producing a grand total of fifteen concurrency classes.

Sadly fifteen classes was never the right answer. The correct answer was twelve before David's update and seventeen afterwards. Either answer would have been correct, even if not current. But fifteen was never correct. This problem is called an inconsistent read because the data that Martin read was inconsistent.

Both of these problems cause a failure of **correctness** (or safety). They result in incorrect behavior that would not have occurred without two people trying to work with the same data at the same time. However is correctness was the only issue, these problems wouldn't be that serious. After all we could arrange things so that only one of us can work the data

at once. While this helps with correctness, it reduces the ability to do things concurrently. The essential problem of any concurrent programming is that it's not enough to worry about correctness, you also have to worry about **liveness**: how much concurrent activity can go on. Often people need to sacrifice some correctness to gain more liveness, depending on the seriousness and likelihood of the failures and the need for people to work on their data concurrently.

These aren't all the problems you get with concurrency, but we think of these as the basic ones. To avoid these problems we use various mechanisms to control concurrency, but sadly there's no free lunch. The solutions to these problems introduce problems of their own. But at least the problems introduced by the control mechanisms are less serious than the basic ones. However this does bring up an important point - if you can tolerate the problems above, you can avoid any form of concurrency control. This is rare, but occasionally you find circumstances that permit it.

Execution Contexts

Whenever processing occurs in a system, it occurs in some form of context, and usually in more than one. There's no standard terminology for execution contexts, so here we'll define the ones that we are assuming in this book.

From the perspective of interacting with the outside world, two important contexts are the request and the session. A **request** corresponds to a single call from the outside world which the software works on and optionally sends back a response. During a request the processing is largely in the servers court and the client is assumed to wait for a response. Some protocols allow the client to interrupt a request before it gets a response, but this is fairly rare. Rather more often a client may issue another request that may interfere with one it has just sent. So a client may request to place an order and then issue a separate request to cancel that same order. From the client's view they may be obviously connected, but depending on your protocol that may not be so obvious to the server.

A **session** is a long running interaction between a client and server. A session may consists of a single request, but more commonly it consists of a series of requests, a series that user regards as a consistent logical sequence. Commonly it will begin with a user logging in, doing various bits of work which may involve issuing queries, and may involve one or more business transactions (see later), at the end the user logs out. Or the user just goes away and assumes the system interprets that as logging out.

Server software in an enterprise application sees both requests and sessions from two angles, as the server from the client and as the client to other systems. As a result you'll often see multiple sessions: http sessions from the client and multiple database sessions with various databases.

Two important terms from operating systems are processes and threads. A **process** is a, usually heavyweight, execution context that provides a lot of isolation for the internal data it works on. A **thread** is a lighter-weight active agent that's set up so that multiple threads can operate in a single process. People like threads because that way you can support multiple requests inside a single process - which is good utilization of resources. However threads usually share memory. Such sharing leads to concurrency problems. Some environments allow you to control what data a thread may access, allowing you to have **isolated threads** that don't share memory.

The difficulty with execution contexts comes when they don't line up as well as we might like. In theory it would be nice if each session had an exclusive relationship with a process for its whole lifetime. Since processes are properly isolated from each other, this helps reduce concurrency conflicts. Currently we don't know of any server tools allow you to work this way. A close alternative is to start a new process for each request, which was the common mode for early Perl web systems. People tend to avoid that now because starting processes tie up a lot of resources, but it's quite common for systems to have a process only handle one request at a time - and that can save many concurrency headaches.

When you're dealing with databases there's another important context - a transaction. Transactions pull together several requests which the client would like to see treated as if they were done in a single request. Transactions can either occur from the application to database: a system transaction, or from the user to an application: a business transaction: we'll dig into these terms more later on.

Isolation and Immutability

The problems of concurrency have been around for a while, and software people have come up with various solutions. For enterprise applications there are two that are particularly important: isolation and immutability.

The various concurrency problems occur when more than one active agent, such as a process or thread, has access to the same piece of data. So one way to deal with concurrency concerns is **isolation**: partition the data so that any piece of data can only be accessed by one active agent. This is indeed how processes work in operating system memory. The operating system allocates memory exclusively to a single process. Only that process can read or write the data linked to it. Similarly you find file locks in many popular productivity applications. If Martin opens a file, nobody else can open that same file. They may be allowed to open a read-only copy of the file as it was when Martin started, but they don't change it and they don't get to see the file between my changes.

Isolation is a vital technique because it reduces the chance of errors. Too often we've seen people get themselves into trouble with concurrency because they try to use a technique that forces everyone to worry about concurrency all the time. With isolation you arrange things so that the programs enters an isolated zone, then within that zone it doesn't have to worry about concurrency issues. So a good concurrency design is to find ways of creating such zones, and ensure as much programming as possible is done in one of these zones.

You only get concurrency problems if the data you're sharing may be modified. So one way to avoid concurrency conflicts is to recognize **immutable** data. Obviously we can't make all data immutable, as the whole point of many systems is to modify data. But by identifying some data that is immutable, or at least immutable as far as almost all users are concerned, we can then relax all the concurrency issues for that data and share it widely. Another option is to separate applications that are only reading data, and have them use copied data sources from which we can relax all concurrency controls.

Optimistic and Pessimistic Concurrency Control

So what happens when we have mutable data that cannot isolate? In broad terms there's two forms of concurrency control that we can use: optimistic and pessimistic.

Let's suppose that Martin and David both want to edit the Customer file at the same time. With **optimistic locking** both of them can make a copy of the file and edit it freely. If David is the first to finish he can check in his work without trouble. The concurrency control kicks in when Martin tries to commit his changes. At this point the source code control system detects that there is a conflict between Martin's changes and David's changes. Martin's commit is rejected and it's up to Martin to figure out how to deal with the situation. With **pessimistic locking** whoever checks out the file first prevents anyone else from editing the file. So if Martin is first to check out, David cannot work with the file until Martin is finished with it and commits his changes.

A good way of thinking about this is that an optimistic lock is about conflict detection, while a pessimistic lock is about conflict prevention. As it turns out real source code control systems can use either style, although these days most developers prefer to work with optimistic locks for source code.

Both approaches have their pros and cons. The problem with the pessimistic locks is that it reduces concurrency. While Martin is working on a file he locks it, so everybody else has to wait. If you've worked with pessimistic source code control mechanisms, you know how frustrating this can be. With enterprise data it's often worse, because if someone is editing data, then nobody else is allowed to read the data, let alone edit it.

Optimistic locks thus allow people to make progress much better, because the lock is only held during the commit. However the problem the optimistic locks is what happens when you get a conflict. Essentially everybody after David's commit has to check out the version of the file that David checked in, figure out how to merge their changes with David's changes, and then check in a newer version. With source code, this happens to be not too difficult to do. Indeed in many cases the source code control system can automatically do the merge for you. Even when it can't auto merge, tools can make it much easier to see the differences. But business data is usually too difficult to automatically merge, so often all you can do is throw away everything and start again.

So the essence of the choice between optimistic and pessimistic is the frequency and severity of conflicts. If conflicts are sufficiently rare, or the consequences are no big deal, then you should usually pick optimistic locks because they give you better concurrency, as well as usually being easier to implement. However if the results of a conflict are painful for the users, then you'll need to avoid them with a pessimistic technique.

Neither of these approaches are exactly free of problems. Indeed by using them you can easily introduce problems which cause almost as much trouble as the basic concurrency problems that you are trying to solve anyway. We'll leave a detailed discussion of all these ramifications to a proper book on concurrency, but here's a few highlights to bear in mind.

Preventing Inconsistent Reads

Consider this situation. Martin edits the Customer class which makes calls on the Order class. Meanwhile David edits the order class and changes the interface. David compiles and checks in, Martin then compiles and checks in. Now the shared code is broken because Martin didn't realize that the order class altered underneath him. Some source code control systems will spot this inconsistent read, but others require some kind of manual discipline to enforce it, such as updating your files from the trunk before you check in.

In essence this is the inconsistent read problem, and it's often easy to miss because most people tend to focus on lost updates as the essential problem in concurrency. Pessimistic locks have a well-worn way of dealing with this problem through read and write locks. To read data you need a read (or shared) lock, to write data you need a write (or exclusive) lock. Many people can have read locks on the data at once, but if anyone has a read lock then nobody can get a write lock. Conversely once somebody has a write lock, then nobody else can have any lock. By doing this you can avoid inconsistent reads with pessimistic locks.

Optimistic locks usually base their conflict detection based on some kind of version marker on the data. this can be a timestamp, or a sequential counter. To detect lost

updates, the system checks the version marker of your update with the version marker of the shared data. If they are the same the system allows the update and updates the version marker.

Detecting an inconsistent read is essentially similar: in this case every bit of data that was read also needs to have its version marker compared with the shared data. Any differences indicate a conflict.

However controlling access to every bit of data that's read often causes unnecessary problems due to conflicts or waits on data that doesn't actually matter that much. You can reduce this burden by separating out data you've *used* from data you've merely read. A pick list of products is something where it doesn't matter if a new product appeared in the list after you started your changes. But a list of charges that you are summarizing for a bill may be more important. The difficulty is that this requires some careful analysis of what exactly is used for what. A zip code in a customer's address may seem innocuous, but if a tax calculation is based on where somebody lives, that address has to be controlled for concurrency. So figuring out what you need to control and what you don't is an involved exercise whichever form of concurrency control you use.

Another way to deal with inconsistent read problems is to use **Temporal Reads** . These prefix each read of data with some kind of time stamp or immutable label. The database then returns the data as it was according to that time or data. Very few databases have anything like this, but developers often come across this in source code control systems. The problem is that the data source needs to provide a full temporal history of changes which takes time and space to process. This is reasonable for source code, but both more difficult and more expensive for databases. You may need to provide this capability for specific areas of your domain logic: see [snodgrass] and [Fowler, temporal patterns] for ideas on how to do that.

Deadlocks

A particular problem with pessimistic techniques is **deadlock**. Say Martin starts editing the customer file and David is editing the order file. David realizes that to complete his task he needs to edit the customer file too, but since Martin has a lock on it he can't and has to wait. Then Martin realizes he has to edit the order file, which David has locked. They are now deadlocked, neither can make progress until the other completes. Said like this, they sound easy to prevent, but deadlocks can occur with many people involved in a complex chain, and that makes it more tricky.

There are various techniques you can use to deal with deadlocks. One is to have software that can detect a deadlock when it occurs. In this case you pick a **victim** who has to throw away his work and his locks so the others can make progress. Deadlock detection is very

difficult to do and causes pain for the victims. A similar approach is to give every lock a time limit. Once you hit the time limit you lose your locks and your work - essentially becoming a victim. Time outs are easier to implement than a deadlock detection mechanism, but if anyone holds locks for a while you'll get some people being victimized when there wasn't actually a deadlock present.

Time outs and detection look to deal with a deadlock when it occurs, other approaches try to stop deadlocks occurring at all. Deadlocks essentially occur when people who already have locks try to get more (or to upgrade from read to write locks.) So one way of preventing deadlocks is to force people to acquire all their locks at once at the beginning of their work, and once they have locks to prevent them gaining more.

You can force an order on how everybody gets locks, an example might be to always get locks on files in alphabetical order. This way once David had a lock on the Order file, he can't try to get a lock on the Customer file because it's earlier in the sequence. At that point he becomes a victim.

You can also make it so that if Martin tries to acquire a lock and David already has a lock, then Martin automatically becomes a victim. It's a drastic technique, but is simple to implement. And in many cases such a scheme actually works just fine.

If you're very conservative you can use multiple schemes. So you force everyone to get all their locks at the beginning, but add a timeout as well in case something goes wrong. That may seem like using a belt and braces, but such conservatism is often wise with deadlocks because deadlocks are pesky things that are easy to get wrong.

It's very easy to think you have a deadlock-proof scheme, and then to find some chain of events you didn't consider. As a result for enterprise application development we prefer very simple and conservative schemes. They may cause unnecessary victims, but that's usually much better than the consequences of missing a deadlock scenario.

Transactions

The primary tool for handling concurrency in enterprise applications is the transaction. The word transaction often brings to mind an exchange of money or goods. Walking up to an ATM machine, entering your PIN, and withdrawing cash is a transaction. Paying the three dollar toll at the Golden Gate Bridge is a transaction. Buying a beer at the local pub is a transaction.

Looking at typical financial dealings such as these provides a good definition for the word transaction. First, a transaction is a bounded sequence of work. Both start and end points are well defined. An ATM transaction begins when the card is inserted and ends when

cash is delivered or an inadequate balance is discovered. Second, all participating resources are in a consistent state both when the transaction begins and when the transaction ends. A man purchasing a beer has a few bucks less in his wallet but has a nice pale ale in front of him. The sum value his assets has not changed. Same for the pub - pouring free beer would be no way to run a business.

In addition, each transaction must complete on and all-or-nothing basis. The bank cannot subtract from an account holder's balance unless the ATM machine actually delivers the cash. While the human element might make this last property optional during the above transactions, there is no reason software cannot make a guarantee on this front.

Software transactions are often described in terms of the ACID properties:

- **Atomicity**: Each step in the sequence of actions performed within the boundaries of a transaction must complete successfully or all work must rollback. Partial completion is not a transactional concept. So if Martin is transferring some money from his savings to his checking account and the server crashes after he's withdrawn the money from his savings, the system behaved as if he never did the withdrawal. Committing says both things occurred, a rollback says neither occurred, it has to be both or neither.
- **Consistency**: A system's resources must be in a consistent, non-corrupt state at both the start and completion of a transaction.
- **Isolation**: The result of an individual transaction must not be visible to any other open transactions until that transaction commits successfully.
- **Durability**: Any result of a committed transaction must be made permanent. This translates to 'must survive a crash of any sort.'

Most enterprise applications run into transactions in terms of databases. But there are plenty of other things than can be controlled using transactions, such as message queues, printers, ATMs, and the like. As a result technical discussions of transactions use the term 'transactional resource' to mean anything that is transactional: that is uses transactions to control concurrency. Transactional resource is a bit of a mouthful, so we just tend to use database, since that's the most common case. But when we say database, the same applies for any other transactional resource.

To handle the greatest throughput, modern transaction systems are designed around the transactions being as short as possible. As a result the general advice is to never make a transaction span multiple requests, making a transaction span multiple requests is generally known as a **long transaction**.

As a result a common approach is to start a transaction at the beginning of a request and complete it at the end. This **request transaction** is a nice simple model and a number of environments make it easy to do declaratively, by just tagging methods as transactional.

A variation on this is to open a transaction as late as possible. With a **late transaction** you may do all the reads outside of a transaction and only open up a transaction when you do updates. This has the advantage of minimizing the time spent in a transaction. If there's a lengthy time lag between the opening of the transaction and the first write, this may improve liveness. However this does mean that you don't have any concurrency control until you begin the transaction, which leaves you open to inconsistent reads. As a result it's usually not worth doing this unless you have very heavy contention, or it you are doing it anyway due to business transactions that span multiple requests (which is the next topic).

When you use transactions, you need be somewhat aware of what exactly is being locked. For many database actions the transaction system locks the rows that are involved, which allows multiple transactions to access the same table. However if a transaction locks a lot of rows in a table, then the database finds it's got more locks than it can handle and escalates the lock to the entire table - locking out other transactions. This **lock escalation** can have a serious effect on concurrency. In particular it's a reason why you shouldn't have some "object" table for data at the domain's *Layer Supertype* level. Such a table is a prime candidate for lock escalation, and locking that table shuts everybody else out of the database.

Reducing Transaction Isolation for Liveness

It's common to restrict the full protection of transactions so that you can get better liveness. This is particularly the case when it comes to handling isolation. If you have full isolation you get serializable transactions. Transactions are **serializable** if they can be executed concurrently and get a result that is the same as you would get from some sequence of executing the transactions serially. So if we take our earlier example of Martin counting his files, serializable would guarantee that he would get a result that corresponds to either completing his transaction entirely before David's transaction starts (twelve) or after David's finishes (seventeen) Serializability cannot guarantee which result, as in this case, but it at least guarantees a correct one.

Most transactional systems use the SQL standard which defines four levels of isolation. Serializable is the strongest level, each level below allows a particular kind of inconsistent read to enter the picture. We'll explore these with the example of Martin counting files while David modifies them. There are two packages: locking and multiphase. Before David's update there are 7 files in the locking package and 5 in the multiphase package, after his update there are 9 in the locking package and 8 in the multiphase package. Martin looks at the locking package, David then updates both, then Martin looks at the multiphase package.

If the isolation level is serializable then the system guarantees that Martin's answer is either twelve or seventeen, both of which are correct. Serializability cannot guarantee that every run through this scenario would give the same result, but it would always get either the number before David's update, or the number afterwards.

The first level below serializability is the isolation level of **repeatable read** that allows **phantoms**. Phantoms occur when you are adding some elements to a collection and the reader sees some of them, but not all of them. The case here is that Martin looks at the files in the locking package and sees 7. David then commits his transaction. Martin then looks at the multiphase package and sees 8. Hence Martin gets an incorrect result. Phantoms occur because they are valid for some of Martin's transaction but not all of it, and they are always things that are inserted.

Next down the list is the isolation level of **read committed** that allows **unrepeatable reads**. Imagine that Martin looks at a total rather than the actual files. An unrepeatable read would allow him to read a total of 7 for locking, then David commits, then he reads a total of 8 for multiphase. It's called an unrepeatable read because if Martin were to re-read the total for the locking package after David committed he would get the new number of 9. His original read of 7 can't be repeated after David's update. It's easier for databases to spot unrepeatable reads than it is to spot phantoms, so the repeatable read isolation level gives you more correctness than read committed but with less liveness.

The lowest level of isolation is **read uncommitted** which allows **dirty reads**. Read uncommitted allow you to read data that another transaction hasn't actually committed yet. This causes two kinds of errors. Martin might look at the locking package when David adds the first of his files but before he adds the second. As a result he sees 8 files in the locking package. The second kind of error would come if David added his files, but then rolled back his transaction - in which case Martin would see files that were never really there.

Isolation Level	Dirty Read	unrepeatable read	phantom
Read Uncommitted	Yes	Yes	Yes
Read Committed	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

To be sure of correctness you should always use the isolation level of serializable. But the problem is that choosing a serializable level of transaction isolation really messes up the liveness of a system. So much so that often you have to reduce the serializability in order

to increase throughput. You have to decide what risks you want take and make your own trade-off of errors versus performance.

You don't have to use the same isolation level for all transactions, so you should look at each transaction you have decide how to balance liveness versus correctness for that transaction.

Business and System Transactions

What we've talked about so far, and most of what most people talk about, are what we call **system transactions**, or transactions supported by RDBMS systems and transaction monitors. A database transaction is a group of SQL commands delimited by instructions to begin and end the transaction. If the fourth statement in the transaction results in an integrity constraint violation the database must rollback the effects of the first three statements in the transaction and notify the caller that the transaction has failed. If all four statements had completed successfully all would be made visible to others at the same time, rather than one at a time. Support for system transactions, in the form of RDBMS systems and application server transaction managers, are so commonplace that they can pretty much be taken for granted. They work well and are well understood by application developers.

However, a system transaction has no meaning to the user of a business system. To the user of an online banking system a transaction consists of logging in, selecting an account, setting up some bill payments, and finally clicking the 'OK' button to pay the bills. This is what we call a **business transaction**. That a business transaction displays the same ACID properties as a system transaction seems a reasonable expectation. If the user cancels before paying the bills any changes made on previous screens should be cancelled. Setting up payments should not result in a system-visible balance change until the 'OK' button is pressed.

The obvious answer to supporting the ACID properties of a business transaction is to execute the entire business transaction within a single system transaction. Unfortunately business transactions often take multiple requests to complete. So using a single system transaction to implement a business transaction results in a long system transaction. Most transaction systems don't work very efficiently with long transactions.

This doesn't mean that you should never use long transactions. If your database has only modest concurrency needs, then you may well be able to get away with it. And if you can get away with it, we'd suggest you do it. Using a long transaction means you avoid a lot of awkward problems. However the application won't be scalable, because those long transactions will turn the database into a major bottleneck. In addition the refactoring from long to short transactions is both complex and not well understood.

As a result many enterprise applications cannot risk long transactions. In this case you have to break the business transaction down into a series of short transactions. This means that you are left to your own devices to support the ACID properties of business transactions between system transactions, the problem that we call **offline concurrency**. System transactions are sill very much part of the picture. Whenever the business transaction interacts with a transactional resource, such as a database, that interaction will execute within a system transaction in order to maintain the integrity of that resource. However, as you'll read below it is not enough to string together a series of system transactions to properly support a business transaction. The business application must provide a bit of glue between the system transactions.

Atomicity and durability are the ACID properties most easily supported for business transactions. Both are supported by running the commit phase of the business transaction, when the user hits 'save,' runs within a system transaction. Before the session attempts to commit all its changes to the record set it will first open a system transaction. The system transaction guarantees that the changes will commit as a unit and that they will be made permanent. The only potentially tricky part here is maintaining an accurate change set during the life of the business transaction. If the application uses a *Domain Model* a *Unit of Work* can track changes accurately. Placing business logic in a *Transaction Script* requires a more manual tracking of changes. But that is probably not much of a problem as the use of transaction scripts implies rather simple business transactions.

The tricky ACID property to enforce with business transactions is isolation. Failures of isolation lead to failures of consistency. Consistency dictates that a business transaction not leave the record set in an invalid state. Within a single transaction the application's responsibility in supporting consistency is to enforce all available business rules. Across multiple transactions the application's responsibility is to ensure that one session doesn't step all over another session's changes, leaving the record set in the invalid state of having lost a user's work.

As well as the obvious problems of clashing updates, there is the more subtle problems of inconsistent reads. When data is read over several system transactions, there's no guarantee that the data will be consistent. The different reads could even introduce data in memory that's sufficiently inconsistent to cause application failures.

Business Transactions are closely tied to sessions. In the user's view of the world, each session is a sequence of business transactions (unless they are only reading data). So as a result we usually make the assumption that all business transactions execute in a single client session. While it's certainly possible to design a system that has multiple sessions in for one business transaction, that's a very good way of getting yourself badly confused - so we'll assume you won't do that.

Patterns for Offline Concurrency Control

As much as possible, you should let the transaction system you use deal with concurrency problems. Handling concurrency control that spans system transactions plonks you firmly into the murky waters of doing your own concurrency. Waters full of virtual sharks, jellyfish, piranhas, and other less friendly creatures. Unfortunately the mismatch between business and system transactions means you sometimes just have to wade into these waters. The patterns that we've provided here are some techniques that we've found helpful in dealing with concurrency control that spans system transactions.

Remember that these are techniques that you should only use if you have to. If you can make all your business transaction fit into a system transaction by ensuring that all your business transactions fit within a single request, then do that. If you can get away with long transactions by forsaking scalability, then do that. By leaving concurrency control in the hands of your transaction software you'll avoid a great deal of trouble. These techniques are what you have to do when you can't do that. Because of the tricky nature of concurrency, we have to stress again that the patterns are a starting point not a destination. We've found these useful, but we don't claim to have found a cure for the ills of concurrency.

Our first choice for handling offline concurrency problems is *Optimistic Offline Lock*, which essentially uses optimistic concurrency control across the business transactions. We like this as a first choice because it's an easier approach to program and yields the best liveness. The limitation of *Optimistic Offline Lock* is that you only find out that a business transaction is going to fail when you try to commit that transaction. In some circumstances the pain of that late discovery is too much: users may have put an hour's work into entering details about a lease, or you get lots of failures and users lose faith in the system. Your alternative then is *Pessimistic Offline Lock*, this way you find out early if you are in trouble, but lose out because it's harder to program and it reduces your liveness.

With either of these approaches you can save considerable complexity by not trying to manage locks on every object. A *Coarse-Grained Lock* allows you to manage the concurrency of a group of objects together. Another step you can do to make life easier for application developers is to use *Implicit Lock* which saves them from having to manage locks directly. Not just does this save work, it also avoids bugs when people forget - and these bugs are hard to find.

A common statement about concurrency is that it's something that is a purely technical decision that can be decided on after requirements are complete. We disagree. The choice of optimistic or pessimistic controls affect the whole user experience of the system. Intelligent design of *Pessimistic Offline Lock* needs a lot of input about the domain from

the users of the system. Similarly domain knowledge is needed to choose good *Coarse-Grained Lock*.

Whatever you do, futzing with concurrency is one of the most difficult programming tasks you can do. It's very difficult to test concurrent code with confidence. Concurrency bugs are hard to reproduce and very difficult to track down. These patterns have worked for us so far, but this is particularly difficult territory. If you need to go down this path it's particularly worth getting some experienced help. At the very least consult the books we've mentioned at the end of this chapter..

Application Server Concurrency

So far we've talked about concurrency mainly in terms of the concurrency of multiple sessions running against a shared data source. Another form of concurrency is the process concurrency of the application server itself: how does that server handle multiple requests concurrently and how this affects the design of the application on the server. The big difference with the other concurrency issues we've talked about so far is that application server concurrency doesn't involve transactions, so working with this means stepping away from the relatively controlled transactional world.

Explicit multi-threaded programming, with locks and synchronization blocks, is complicated to do well. It's easy to introduce defects that are very hard to find, since concurrency bugs are almost impossible to reproduce - resulting in a system that works correctly 99% of the time, but throws random fits. Such software is incredibly frustrating both to use and to debug. As a result our policy is to avoid the need for explicit handling of synchronization and locks as much as possible. Application developers should almost never have to deal with these explicit concurrency mechanisms.

The simplest way to handle this is to use **process-per-session** where each session runs in its own process. The great advantage of this is that the state of each process is completely isolated from the other processes, so the application programmers don't have to worry at all about multi-threading. As far as memory isolation goes, it's almost equally effective to have each request start a new process, or to have one process tied to the session that's idle between requests. Many early web systems would start a new Perl process for each request.

The problem with process-per-session is that it uses up a lot resources, since processes are expensive beasties. To be more efficient you can pool the processes, such that each process only handles a single request at one time, but can handle multiple requests form different sessions in a sequence. This approach of pooled **process-per-request** will use much less processes to support a given amount of sessions. Your isolation is almost as good: you don't have many of the nasty multi-threading issues. The main problem over

process-per-session is that you have to ensure any resources used to handle a request are released at the end of the request. The current Apache mod-perl uses this scheme, as do a lot of serious large scale transaction processing systems.

Even process-per-request will need many processes running to handle a reasonable load, you can further improve throughput with by having a single process run multiple threads. With this **thread-per-request** approach, each request is handled by a single thread within a process. Since threads use much less server resources than a process, you can handle more requests with less hardware this way, so your server is more efficient. The problem with using thread-per-request is that there's no isolation between the threads, any thread can touch any piece of data that it can get access to.

In our view there's a lot to be said for using process-per-request. Although it leads to less efficiency than thread-per-request, using process-per-request is equally scalable. You also get better robustness, if one thread goes haywire it can bring down an entire process, so using a process-per-request limits the damage. Particularly with a less experienced team, the reduction of threading headaches (and the time and cost of fixing bugs) is worth the extra hardware costs. We find that few people actually do any performance testing to assess the relative costs of thread-per-request and process-per-request for their application.

Some environments provide a middle ground of allowing isolated areas of data to be assigned to a single thread, COM does this with the single-threaded apartment and J2EE does this with Enterprise Java Beans (and in the future with isolates). If your platform has something like this available, this can allow you to have your cake and eat it - whatever that means..

If you use thread-per-request then the most important thing is to create and enter an isolated zone where application developers can mostly ignore multi-threaded issues. The usual way to do this is have the thread create new objects as it starts handling the request, and ensure these objects aren't put anywhere (such as a static variable) where other threads can see them. That way the objects are isolated because other threads have no way of referencing them.

Many developers are concerned about creating new objects because they've been told that object creation is an expensive process. As a result people often pool objects. The problem with pooling is that you have to synchronize access to the pooled objects in some way. But the cost of object creation is very dependent upon the virtual machine and memory management strategies. With modern environments object creation is actually pretty fast[missing reference] (off the top of your head: how many Java date objects do you think we can create in one second on Martin's 600Mhz P3 with Java 1.3? We'll tell you later.) Creating fresh objects for each session avoids a lot of concurrency bugs and can actually improve scalability.

While this tactic works for many cases, there are still some areas that developers need to avoid. One is any form of static, class based variables or global variables. Any use of these has to be synchronized, so it's usually best to avoid them. This is also true of singletons. If you need some kind of global memory use a *Registry*, which you can implement in such a way that it looks like a static variable, but actually uses thread-specific storage.

Even if you're able to create objects for the session, and thus make a comparatively safe zone, there are some objects that are expensive to create and thus need to be handled differently - the most common example of this is a database connection. To deal with this you can place these objects in an explicit pool where you acquire a connection while you need it and return it to the pool when done. These operations will need to be synchronized.

Further Reading

In many ways, this chapter only skims the surface of a much more complex topic. To investigate further we'd suggest starting with [Bernstein and Newcomer], [Lea], and [Schmidt].