We are continuing our discussion of fault-tolerance and atomicity.

And sort of teaching these lectures makes me feel, in the beginning, like those TV shows where they always, before a new episode, tell you everything that happened so far in the season.

So we will do the same thing.

The story so far here is that in order to deal with failures, we came up with this idea of making modules have this property of atomicity, which actually has two aspects to it, one of which is an all or nothing aspect, which we call recoverability, and the other is a way to coordinate multiple concurrent activities so you get this illusion that they are all separate from each other.

And we call that isolation.

And the basic rule here for achieving recoverability was we repeatedly applied this one rule that we called the "Golden Rule of Recoverability" which is to never modify the only copy.

And we used that rule to build up this idea of recoverable sector and we used that idea of recoverable sector to come up with two schemes for achieving recoverability, one using version histories where you had a special case of this "Never Modify the Only Copy Rule" which was never modify anything.

So, for any given variable, you must create lots and lots of versions never updating anything in place.

And then we decided that it is inefficient, so we came up with a different way of achieving recoverability using logging.

Then for isolation, we did this last time where we talked about serializability where the goal was to allow the steps of the different actions to run in such a way that the result is as if they ran in some serial order.

And we talked about a way of achieving that with cell storage.

In particular, we talked about using locks as an abstraction, as a programming primitive to achieve isolation.

And, in particular, the key idea that we saw here was that serializability implied that there were no cycles in this data structure called the action graph.

And, as long as you could argue that for a given method of locking, as long as you could argue that the resulting action graph had no cycles, you were guaranteed serializability.

And, therefore, the scheme provided isolation.

And, in particular, a scheme we looked at near the end was two-phase locking.

Where the idea is that you never acquire a lock for any item if you have done a release of any other lock in the atomic action so far.

And that's the reason why this is called two-phase locking because if you look at the two phases being a lock acquisition phase where the only thing that is happening is locks are being acquired and nothing is being released so the number of locks is strictly increasing with time.

And then there is a certain point of the action after which you can only release locks.

And you cannot acquire a lock the moment you have released any given lock.

And the way we argued this protocol achieved isolation was to consider the action graph resulting from some execution of two-phase locking and argued that if there was a cycle in that resulting action graph then two-phase locking gets violated.

And, therefore, two-phase locking provides an action graph which does not have cycles and, therefore, achieves serializability. Two-phase locking is fine and is a really good idea if you're into using locks.

It has the property that you do not actually need to know, the action does not need to know at the beginning which data items it is going to access which means that all you need to do is to make sure that you do not release anything until everything has been acquired.

But you do not have to know which ones to acquire before the start of the action.

You just have to keep acquiring sort of on demand until typically you get to the commit point.

And once you commit you can release the locks.

Now, in theory you can release a lock at any given time once you are sure that you are not going to acquire anymore locks, but that theoretical approach only works if you are guaranteed that there will be no aborts happening.

Now, in general, you cannot know beforehand when an action might abort.

I mean the system might decide to abort an action for a variety of reasons, and we will see some reasons today.

In practice, what ends up happening is that when you abort you have to go back and look through the log and

undo the actions, run the undo steps associated with steps that happened in the action.

Which means that because the undo goes ahead into cell storage and uninstalls whatever changes were made, it better be the case that when abort starts undoing things, it better be the case that the cell storage items that are being undone have locks owned by this action that is doing the undo.

What this means is that if you have a set of statements here that are doing read of x and a write of y and things like that and then you have commit here -- But this is the last point in time at which you are reading and writing.

And after that you are doing some computation here not involving any reads or writes.

The action might abort anywhere here because the process, I mean this thread might be terminated and the action will have to abort.

What that means is that a release for a data item that is required by abort in order to undo the state changes that have been made, that lock had better not be released before here.

Because if the lock got released here then some other action could have acquired the lock and gone ahead and started working with the changes made by this action.

And now it is too late to abort.

Someone else has already seen the changes that have been made.

In fact, you cannot be guaranteed now that later on you actually can regain the lock and the results would be wrong.

So, in fact, the two-phase locking rule really is that you cannot release any lock until all of the locks have been acquired.

And, moreover, any locks that are needed in order for abort to successfully run had better not be released until you are sure that the action won't abort anymore.

And the only time you can be sure that the action won't abort anymore is once commit has been done.

What that really means is that the release of the locks of all of the items that are required for undoing this action had better happen after the commit point.

And, moreover, no item locks should be released until all of the acquires have been done.

Now the reason I've said this in two parts is that if you are just reading a data item, you actually don't need to hold

onto the lock of that item in order to do the undo because all you did was read x.

There is no change that happened to the variable x, which means although you need to acquire the lock of x in order to read it because you don't want to have other people making changes to it while you are reading it, you don't actually need to hold onto the lock of x in order to do the undo because you are not actually writing x during the undo step.

So that's the amendment to the two-phase locking rule.

Things that you need in order to do undos for should only be released after you are sure that no aborts will be done, which means after the commit point.

This way of doing two-phase locking is actually a pretty good scheme.

And it turns out that, in many ways, it is the most efficient and most general method.

What that means is that there might be special cases where other ways of other protocols for doing locks of objects perform better under certain special cases compared to two-phase locking, but if you've bought into using locks in order to do concurrency control and you don't know very much about the nature of the actions involved then two-phase locking is quite efficient.

I mean there are variants of two-phase locking but, by and large, this idea, it's very hard to do much better than this in a very general sense if you're using locking for doing concurrency control.

But there are a set of problems.

It's not two-phase locking, as we have described it so far, completely solves the problem of insuring that actions perform well.

And a particular problem that happens any time you use locks like here is deadlocks.

And we have actually seen deadlocks before in an earlier chapter when we talked about synchronization of threads, and it is exactly the same problem.

And the way you deal with it pretty much is almost the same.

What is the problem here?

Well, what could happen is that one action does read x and write y and the other action does read y and write x.

And now you intersperse the acquires and releases so you do an acquire of lx here and maybe you do an acquire of ly here and here you do an acquire of ly and you do an acquire of lx.

And what could happen is that once you get to this stage where this action has come this far and is about to run this and this other action has come up to here, now you are stuck because this action has to wait until that is released and this action has to wait until that is released and neither can make progress.

So there are a few different ways of dealing with it.

And the simplest way and the way that turns out to be one that is often used in practice both because it is simple and because once you implement the technique you don't have to do very much else is to just set timers on actions.

So it's just to timeout.

And if you notice that for a period of time an action has not made any progress then have a timeout that is associated with the action.

And if the action itself notices that it hasn't made any progress, perhaps in another thread, then just go ahead and abort this thread.

Now, it is perfectly OK to abort.

And, in this particular case, aborting either of these actions is enough and the other will make progress and then you are done.

And then the action that got aborted can retry.

So the first solution is to just use a timer.

And there is a school of thought that believes that in practice deadlocks should not be very common.

And the reason is that deadlocks occur if there is, you know, there has to be a contention for resources and there has to be contention for multiple threads for the same resources.

And it has to be more than one resource, because if you just have one resource you cannot really get a deadlock which means that you are sort of running multiple actions that are contending for a number of different shared objects.

And what that suggests is if there is a high degree of concurrency like that and shared contention then it may be

hard for you to get high performance.

A lot of people think that the right way to be designing applications is to try hard to insure that the degree of sharing between objects is actually quite small.

For example, rather than set up a lock on an entire big database table, you might set up locks at final granularities.

And if you set up locks at final granularities the chances of multiple actions wanting to gain access to the same exact fine-grained entry in a table might be small.

And in that situation, given that the chances of a deadlock occurring are rare, timing out every once in a while and aborting an action is not going to be catastrophic.

It's OK.

It is a rare event.

So rather than spend a whole lot of complexity dealing with that rare event, just go ahead and let something abort.

Let an action that hasn't made any progress abort.

Moreover, these timers are necessary anyway because an action might end up getting stuck in an infinite loop or it might end up getting stuck in a situation where it is not really waiting for a lock there is just a bug in it.

There is a problem with it, it is not really making any progress and maybe it is consuming resources and no one else can make progress.

So the system anyway needs a way to abort those actions.

And it needs a timeout mechanism anyway.

So why not just use that same mechanism to deal with deadlocks as well.

Probably somewhat a minority, but some other people believe that deadlocks might happen.

And, when they do happen, perhaps because the granularity of locking in your system is not fine-grained then you do not want to get stuck.

And you want to optimize, at least you want to do reasonably well rather than waiting for some long timeout period before aborting an action.

And people who believe that build a data structured called the "Waits-For Graph".

And the best way to understand this is imagine you have a database system that supports isolation and any time you want to acquire a lock you send a message to this entity in the database system called lock manager asking to acquire a lock.

And any time you release it you do the same thing.

What that lock manager can do, for each lock it can keep track of which actions running concurrently has acquired that lock and which action is waiting for a lock.

And what you can do now is build up a graph of actions and locks and look to see whether there is some kind of cycle where you have action A waiting for lock B and lock B is being held by action C and action C is waiting for lock D and lock D is being held by action A.

When you have a cycle in this graph then you know that you have a deadlock and none of those actions can make progress so go ahead and kill one.

And you can be sophisticated about deciding which one to kill.

You might kill the one, for example, that has been waiting the shortest amount of time because the others have been waiting longer so they might make progress, or you might have other policies for deciding which ones to kill.

In practice, both these systems are used sometimes by the same system combining these ideas.

For example, if you look at like an Oracle database system, it uses primarily timers.

At least from what I could tell, it does not seem to have any mechanisms for really doing this check of a Waits-For graph.

It just uses timers.

And one of the oldest transaction processing systems was a system called CICS from IBM which also basically used timers, but there are other systems.

For instance, IBM has this system called DB2 and Microsoft Sequence server that both use this Waits-For data structure.

And, in fact, Microsoft's system seems to have a hundred thousand different knobs for deciding how to turn off deadlocks, including the ability to set various priorities on different actions that might be running.

And it is not actually apparent that those knobs actually are usual for anything or how you set them but that they have a lot of things that you could set.

Sounds familiar.

Now, you can combine these two.

And I think certain products combine these two ideas.

One decision you have to make is to decide when to check this Waits-For graph.

And an aggressive way of doing it is the moment anybody does an acquire or anybody does a release, in particular an acquire, you update your lock manager's data structure and immediately look to see if you have a cycle.

Of course that takes time and effort.

You might decide not to both but rather periodically look for cycles in this Waits-For graph when a timer fires, so every three seconds go ahead and look for cycles.

So you might combine these ideas in a bunch of different ways.

Now, if you recall from several lectures ago, another way of dealing with deadlock is to order all of the locks that an action might be able to acquire in a particular order and insure that all of the actions acquire the locks in exactly the same order.

And that will insure there are no cycles because you have to go in the same order, but that idea requires you to know beforehand which data items you wish to gain access to.

And that's often not possible in many systems in which you care about isolations.

So that's usually not adopted at least in any database system.

OK, so we talked about deadlocks.

We talked about when you can release a lock that you acquire in order to abort because you cannot release it typically, in reality, until the commit point is done.

The last issue we need to talk about is an interaction between logs and locks.

And this interaction has to do with, so we already saw what happens when you abort.

When you abort you need to undo so you better make sure that to do the undo you have the locks for those cell items.

You don't have to abort but suppose you crash.

Suppose the system crashes and recovers.

At that point, when it recovers, it is going to run a recovery procedure which has some combination of redoing the winners and undoing the losers.

Now, when it's undoing things and redoing things it needs access to items in the cell store.

And we've already seen when the system is normally running, in order to change items in your cell store you need to gain access to locks.

The question now is during crash recovery when the system is running this redo undo thing, where do you get these locks from and do you need to gain access to the locks?

Now, in general, the answer to the question might be that you need to be very careful and perhaps need access to the locks when you're running recovery.

But there is one simplification that systems typically make that eliminates that requirement.

And that simplification is that during crash recovery you don't really allow new actions to run on your system.

So when a system crashes and it is recovering, do not allow new actions to run until recovery is complete.

And only then do you start new actions.

What this means is now we just have to worry about insuring isolation clearly during recovery without having new actions coming in and muddling things up.

The question really to think about is whether before the crash, because the log is the only thing you have in order to do recover, whether in the log you actually need to keep track of which locks were being held when the system was running just fine.

And if it turns out that the log has to encode in it the locks that were being held, it could be quite complicated and a little bit messy.

But if you think about it, the nice thing is that we don't actually have to encode the locks at all, store the locks at all.

The locks can be completely involved in the storage.

And that is because when you start off, when you have a log which has various redo items and undo items, in any element of the log, let's say an item x has been updated in that log entry.

Then you know for sure that at the time this log entry was written, the action that was making this update did hold onto this lock and that this change being made here that got written to the log was, in fact, isolated assuming the locking protocol was correct, was, in fact, isolated from everything else concurrently that was going on.

And so, although the locks are not explicit, the log encodes in it the actual serial order, some serial order of execution that did provide isolation before the crash.

Therefore, if you just blindly go back through the log and make those changes in sequential order then you are assured that the changes you make are, in fact, going to be isolated from one another.

So you do not actually have to worry about storing the locks before the crash into the log, and that makes life quite simple.

That wraps up the discussion of atomicity and, in particular, isolations.

For the rest of today and next time we are going to be talking about some uses of atomicity.

And the plan is the following.

The plan is the first application of atomicity which actually is the umbrella for a number of things we are going to be looking at is a transaction.

And a transaction is defined as an atomic action that has a few other properties that it holds.

And the first property is consistency and the second property is durability.

And the second thing we are going to look at, next lecture actually, is atomicity when you have a distributed system.

It is using atomicity on one computer to build out a system that provides atomicity in a distributed system.

So we will talk about consistency the rest of today and the recitation for tomorrow looks at a paper for reconciling replicas, which is a particular aspect of consistency.

And then next lecture next week we will talk about multi-site atomicity.

And the recitation next week we will talk about durability.

And once we do all of that, that kind of wraps up this fault-tolerance part of 6.033. Let me first talk a little bit about transactions.

Transaction is an atomic action that has two other properties associated with it.

And people in the literature often, in collegial terms, refer to transactions as having a property called the ACID property where ACID stands for atomicity, consistency, isolation and durability.

And you will see this term a great deal in the literature and people will use this all the time.

And, for various reasons, the way we have done things in this class, some of these terms are used in slightly different ways from the ACID term.

When most people, at least in distributed systems and database systems, use the word atomicity, what they mean is what we meant by recoverability.

So it is all or nothing.

When they use the letter I here for isolation, they mean exactly the same thing here that we did.

And consistency and durability unfortunately are going to mean the exact same thing.

But really the point to notice is that these two properties, atomicity and isolation are things that are independent of an application.

They just are properties of atomic actions that an atomic action can be recoverable and can be isolated.

And you do not have to worry about what the application is.

It could be an application in database systems.

It could be something in a processor where you are trying to provide recoverability or isolation for instructions.

These are properties that are, in some sense, somewhat more fundamental and lower layer properties than these other two properties.

What consistency means is the property of an atomic action that is some application-specific invariant.

Consistency of a transaction says that if you have a transaction that commits then some set of consistency invariants must hold.

I will describe some examples of what this means.

Consistent just says that there is some application-specific invariants that must hold.

And durability says that if a transaction commits then the state changes that it has made, that the data items that it has changed has to last for some period of time.

And the period of time that they have to last for is defined by the application.

And there are many examples.

A simple example of durability might be that the changes made by an atomic action just have to last until the entire thread finishes.

And, at the other extreme, you could get into semantics of durability which say that the changes made by an atomic action have to last for three years or for five years or for forever which is a really hard thing to solve.

But you might define semantics that relates to the permanence of data.

For how long do you want the changes that you made to last and be visible to other atomic actions?

There are two cases for consistency that we need to talk about.

The first one is consistency in a centralized system.

An example of this, and the most common example of this is in database systems that support transaction where you might have rules that are also called integrity rules for deciding whether you are allowing a transaction to commit or not.

Let me give you a couple of examples of this.

Let's say that you have a type of database system as a relational database system where all of the data is stored in tables.

For example, you might have a table storing a student ID, a student name and let's say the department that the student belongs to.

And let's say the departments have IDs.

And you might have another table in your system that stores a department ID and a department name.

Now, you might have a transaction that makes updates to entries in this table, you know, one or more rows in this table could actually make updates to just specific cells of this table.

It could add a new student ID, add a name and add some department ID.

Now, the kind of constraint we are worried about, the kind of invariants we are worried about are things where the person who has designed this database might say that you are not allowed to add a department ID that is nonexistent.

And what that means is that there are these two tables.

And you should not allow any transaction to write the department ID which is not already in this table.

So if 43 might be in this table and 25 might be on this table, but a number that is not in this table should not be added here.

And so the transaction processing system might decide, will, in fact, not allow this transaction to commit if it is writing a value that is not in this other table.

And, for those familiar with databases, relation databases, there are these two tables called T1 and T2. This might be a primary key.

Department ID might be a primary key of T2 defined as what is called a foreign key in T1, which means that you are not actually allowed to add something to a foreign key if it is not already in the other table where that same column is a primary key.

So there are rules like this in most relational database systems and there are a variety of rules like this that all have to do with maintaining the integrity of the data that you add here.

Now, this has nothing to do with isolation.

It has to do with atomicity because these rules are typically checked at the commit point, because until then anything could happen.

So, right before you commit, there are these invariants on the data that are application-specific that you need to check.

But it has nothing to do with locks.

It has nothing to do with anything.

It sort of presumes atomicity, and after that it checks these application-specific rules.

And you can get quite sophisticated.

Some of these things about primary keys and secondary keys are things that are checked by most transaction processing systems, but you could get quite sophisticated about these rules.

For example, you could have rules.

Let's say you have a database storing employees and their salaries.

You could have rules that say any time an employee gets a raise then everybody else in the same peer group also gets some kind of raise.

And so you wouldn't allow any transaction to commit that did not insure that invariant to hold.

And checking these things could be quite difficult, and most systems do not actually do a really good job of checking these things.

The sets of rules they allow you to write is quite limited because checking it is quite hard, because when you are trying to commit a transaction now you might have to check a large number of rules.

And some of them could be both time-consuming and complicated.

But the main point here is that these rules are application-specific. And that is what defines consistency of the data that you have.

The more interesting case for consistency and the thing that is going to occupy us for the rest of today and tomorrow is consistency in distributed systems.

In particular, when the same data gets distributed, typically for fault-tolerance and for availability, to insure that the data is available at different locations, you end up with consistency problems.

And we have already seen a few examples of this.

One example of this is in the "Domain Name System" which maintains mapping between domain names and IP addresses.

And, if you remember, in order to achieve availability and good performance, these mappings between DNS names and IP addresses where cached essentially on demand.

Whenever a name server on the Internet made an access to that name it address cached the mapping results.

And so now you have to worry about whether the data that is cached somewhere out on the Internet is, in fact, the correct data where correct is defined as the data that is being maintained by the primary name server.

And if you think about DNS did, it actually used a mechanism of expiration times to keep this cache consistent.

And what that means is that the only time you are guaranteed that the data in a cache is, in fact, the data that is stored at the primary name server for that name is when this expiration time finishes.

And the first access after the expiration time requires the name server to go to the original primary name server and do a look up of the name.

So the rest of the time you cannot actually be guaranteed that the data is consistent.

And, in other words, you are not getting what is considered strong consistency.

What is strong consistency?

One way to define the semantics of what it means for data to be consistent in a distributed system is it is sort of a natural definition which is to see that any time you do a read anywhere, any node does a read of some data, read returns the result of the late write.

That is one notion of consistency.

And a system provides strong consistency if you can insure that every read returns the result of the last write that was done on the data.

And this is really hard to provide because what it typically means is that the data is widely replicated or cached.

Any time anybody changes the data you have to make sure that all of the copies get that change.

And, even if you work really hard to invalidate all the entries and make changes to it, there are these small windows of vulnerability where -- In fact, in DNS, for example, even the first access that you make the server after the expiration time may not guaranty that when the response returns, the response is, in fact, the newest response because the primary name server could send a response.

And, while it is coming back to the person who made the query, the data could get changed at the primary name

server so it is really hard to guaranty this, at all points in time, in a distributed system.

And it gets much harder when there are failures making certain copies unavailable or making access to a primary in the DNS case unavailable.

In practice, in most systems, the kind of consistency that people try to get is eventual consistency or they try to approximate strong consistency in some other way.

And eventually consistency just -- It is a little bit of a loser notion, but what it says is that there might be periods of time where things are consistent or that the system is doing work in the background to make sure that all of the copies of a given data item are, in fact, the same and are the result of the last write to that data.

Again, the notion of eventual consistency depends a lot on the application.

So, really, to specify this precisely you have to look at in the context of the application.

Different applications you have different notions of consistency and eventual consistency.

So we looked at DNS as an example.

Another example to look at is something you might be familiar with which is "Web caches".

Web caches, for example, your browser has a cache in it.

And there might be Web caches located elsewhere in the network that capture your requests.

And people use Web caches to save latency or to prevent slamming a Web server that might otherwise get overloaded.

The semantics here are usually that you do not just return stale data.

If the data has changed on the Web server, it might be that you actually want to return good data to the client.

The way this is normally done is for the client or for any cache to first check with the Web server to see if the data has been changed since the last cached version.

Let's say that the cache went to the Web server at 9:00 in the morning and had to go there because it did not have the data in the cache.

And it got some data back.

The data has a timestamp on it.

Then the next time somebody makes a request to the cache, the cache does not just return the data immediately.

What the cache usually does is to go to the Web server and ask the Web server if the data has changed since 9:00 in the morning.

If the data has changed since 9:00 in the morning you might retrieve the data.

You would retrieve the data for the server.

If not then go ahead and return the data to the client.

This is also called "If-Modified-Since" because what you are saying is the cache is telling the server send me the data if it has been modified since the last time I know the version of the data that I have.

And a convenient way to represent that is as a timestamp.

It's just a version of the data.

So you can see that this actually provides a more stronger consistency semantics than DNS.

Because in DNS the data could have changed and your cache just has outdated data.

But for the application that DNS is used for it is perfectly OK for that to be the case.

Now, in general, in distributed systems there is a tradeoff between the consistency of data at the different replicas and availability.

Availability just means that clients wanting data should get some copy of the data.

Now, if the system is strongly consistent then the copy of data that you get is, in fact, the result of the last write.

But the tradeoff occurs between availability and consistency because in many distributed systems your networks are not reliable or nodes themselves are not reliable and they might fail.

So in the presence of failures, say network partitions or failures of nodes, it turns out to be really hard to guaranty both high availability and strong consistency.

As sort of a trivial existent example of this, if you have three copies of the data and you were not very careful about figuring out your write protocol.

Let's say that your write protocol was to sort of write to one version and then your read protocol was to just read from some other version and for some process in the background to transfer the replica from the first version that the client wrote to, to all of the other copies, then there would be periods of time of the network where partitioned you could end up in a situation where the version that a given client is reading is not actually the last version of the data that was written.

In fact, if you started thinking about DP2, Design Project 2, really, one part of it gets at how you manage replicated data.

For example, when the utility that does the archiving publishes data, one approach it might take is to publish the data that it wants to archive to all of the copies, to all of the replica machines.

And the read protocol might be to read from one of them.

Now, if you insure that the write protocol finishes and succeeds only when all of the replica machines are updated then you can try to get at a decent version of consistency.

But you need to be able to do that when failures occur.

The network might fail or nodes might fail, and you need to figure out how to do that.

But you might decide that writing to end copies and reading from one copy is difficult or has high overhead so you might think about ways of writing to certain subsets, writing to a subset of the machines and reading from a subset of the machines to try to see whether you could come up with ways to get a consistent version of the data.

Or you might decide that the right way to solve the problem is not to try to achieve really strong consistency in all situations but to relax the kind of consistency you want and maybe a different version of semantics.

As long as you are precise about the semantics that your system provides, it might be a different solution or reasonable solution to the problem.

So one interesting way in which people achieve reasonable strong consistency in tightly coupled distributed systems, and distributed systems that are not across the Internet where a network could arbitrarily fail, but in more tightly coupled systems is in a multiprocessor.

If you have a computer that has many processors -- And the abstraction here for this multiprocessor is that of shared memory.

You actually have memory sitting outside here, and these processors are reading and writing data to this memory.

The latency to get to memory and back is high.

So, as you know, processors have caches on them.

As long as the memory locations that are being written and read are not shared between them these caches could function just fine.

And when there is an instruction running on one of these processors that wants to access some memory location, you could just read and write from the cache so things would just work out.

The problem arises when there is a memory location being read here that actually was previously written by this processor.

And, if you read it here, then you might get an old version of the data.

And if you think of just memory as the basic abstraction, virtual memory then this is bad semantics because your programs wouldn't function the same way as they did when you just had one processor or when you didn't have the caches at all and you just went directly to memory from multiple processors.

The question is how do you know whether the data in a cache is good or bad?

Now, like in the Web caches case, checking on every access whether the data has changed is not going to be useful here because the amount of work it takes to check something is about the same as the amount of work it takes to read or write something because you have taken the latency hit for that.

So that approach is not going to work.

The solution that is followed in many systems is to use two ideas.

The first idea is that of a "Write-Thru Cache".

What a write-thru cache says is if there is a write that happens here, or store instruction, the cache gets updated.

But, in addition to the cache getting updated, the data also gets written through on the bus to the memory location here.

So that is the first idea, to use a write-thru cache.

The second is because this is a bus all of these nodes can actually snoop on this bus and see what activity there is on the bus because it is a shared bus.

It is a very special kind of network, as I said.

You cannot apply this idea in general.

It is a very special kind of network where because it is a bus and nothing fails, or the assumption is nothing fails, everybody can check to see what is going on on the bus.

And any time there is any activity on the bus that corresponds to something that is stored in any node's cache you can do two things.

You can actually invalidate that cache entry but you can actually also see what the update is and go ahead and look at the change that was being done and update your cache.

And this idea is sometimes called a "Snoopy Cache" because you have these caches that are snooping on activity that is occurring in your system.

And this is one way in which you can achieve something that resembles strong consistency.

But it actually turns out, if you think hard about it, a precise version of strong consistency is really hard to achieve.

In fact, it is very, very hard to even define what it means for any read to see the result of the last write because when you have multiple people reading and writing things, when you get down to the instruction level, it turns out to be really hard to even define the right semantics.

A lot of people are working on this kind of thing.

But this is a little bit of a special case because this kind of solution applies only in a very tightly coupled system where you do not really have failures and everybody can listen to everything else.

But it is interesting to note that there are cases when you can achieve it and that is why this is interesting.

It is practically useful.

So the main thing about Design Project 2 that relates to the consistency discussion is for you to try to, I mean at least one part of it, in case it was not clear from the description of the project is for you to think about what kind of consistency you want and come up with ways to manage these different replicas.

We are going to stop here.

Next week we will talk about multi-site atomicity.

Tomorrow's recitation is on a system called Unison which also looks at consistency when you have mobile computers that are trying to synchronize data with servers.