All right, you guys, let's go ahead and get started.

I am back.

I know you guys all

missed me. Just a couple of announcements. Since you guys didn't have recitations this week. I want to make sure that you guys remember the Design Project 2 proposals are due next tomorrow in class, as is Hands On 6. Also, Quiz 2 is graded and is ready to be handed back.

If you go to office hours with your TA this afternoon, you can pick it up or you can get it in class tomorrow.

We will post the statistics as soon as we get them.

We are still waiting to get the scores from one of the TAs into the thing, so I do not want to say anything until we know for sure.

All right.

What we have seen so far, we saw last time, we talked about this notion of transactions, we talked about the notion of making actions durable and consistent.

This time what we are going to do is look at a related topic, a slightly more advanced topic that is more related to the notion of atomic actions that we spent two or three lectures previous to the one about transactions talking about.

If you remember, an atomic action, we say it is both recoverable and isolated.

When we say an action is recoverable, that means, remember, it either all happens or it does not happen at all.

When we say an action is isolated, it means, from the point of view of other actions that are running concurrently in the system, the action appears to only be one unit.

So then none of the intermediate states of the action is visible to any other actions that are running concurrently.

And we talked about the use of the logging protocol to allow us to recover actions.

We also, in the text and in class, talked briefly about version histories as an alternative way that we can recover actions.

And we talked about how to do isolation.

We talked about several different methods for doing isolation.

We spent a while talking about locking as a mechanism that we use to isolate actions from each other.

What we are going to talk about today is a related topic, and it has to do with when you have actions that are spread across multiple sites, multiple different computers.

And this is going to tie together some of the concepts that we learned in the previous section on networking with the more recent stuff that we have been talking about with atomic action.

The topic today is multi-site atomicity.

And just to give you a simple example of what we mean by a situation in which you might care about multi-site atomicity, suppose that you are running a travel website.

You have some travel site, and that travel site you have negotiated agreements with various different people who sell airline tickets.

Maybe you have agreements with Jet Blue and USAir and some other set of airlines.

And you want to make it so that when somebody purchases a ticket, they may purchase a set of flights that are purchased together as one unit where there are different flights on different airlines.

I might want to purchase a flight from here to San Francisco on JetBlue and then a flight from San Francisco back to here on USAir.

Sometimes people want to use multiple different vendors when they are purchasing their tickets.

And it may be the way that your travel site talks to JetBlue and USAir is over the Internet.

They use some remote procedure call to ask JetBlue or USAir to reserve a seat on their

behalf.

But when somebody is using your website, you want to create the illusion that these reservations that are made on the website are sort of one atomic unit.

You do not want it to be that a person gets a reservation on JetBlue and does not get a reservation on USAir.

You want them to make a reservation and get all of the reservation or none of the reservation.

In order to make that work, we are going to need some sort of special support from the travel site, JetBlue and USAir.

Suppose, for example, I did not have any special support from the JetBlue website and I said I want to purchase this ticket to San Francisco.

JetBlue goes ahead and says I am ready and I purchased that ticket for you.

And then suppose we cannot get this reservation on USAir.

Now we are in trouble.

We need some way to back out of the action with JetBlue and say wait, never mind, I did not mean to actually purchase that ticket because I could not complete the rest of my purchase.

We are going to talk about how to provide this kind of, what we call, multi-site atomicity where we want this whole action that includes purchases of tickets from these different websites to appear as though it was one atomic action.

Particularly we want to make sure that this whole thing is both recoverable and isolated.

To sneak up on this topic, because there are a bunch of different issues that are going to come up here and are going to be a little bit complicated, we are going to start by looking at a simpler version of this problem.

Let's suppose that we have all of these things running on the same computer together, that is they are not connected over the Internet, there is not this possibility of a message being lost like there would be on the Internet.

So we are going to look at these things as though they are all running together on the same

computer.

In that case, we call these actions that are running together on the same computer "nested atomic actions".

Once we see how this works on just the single computer case then we will sort of look and see how it gets more complicated when we extend it out to the multi-computer case.

Let's simplify this a little bit.

Suppose we had our buy ticket procedure looking something like begin, buy on JetBlue, then buy on USAir and then end.

I am just going to call these two things A and B for now just to sort of simplify it so I do not have to write JetBlue and USAir over and over again.

And the property that we want is that each one of these actions, in and of itself, should be atomic with respect to the other actions.

We want atomic with respect to each other.

That is one property we want.

We want that because, for example, suppose that buying JetBlue requires me to provide some credit card number or debit card number that is going to go out and purchase the ticket.

Well, I do not want to have the action that purchases from JetBlue and the action that purchases from USAir simultaneously decrementing my, say, bank account.

Because we saw how you could imagine getting into trouble where they would both read the balance at the same time and then both decrement and then both write at the same time.

You could get some mixed up balance left in your bank account, for example.

So if these things are, let's say, for example, debiting a bank account, you want to make sure that these actions are atomic with respect to one another.

You also want to make sure that this whole thing is atomic with respect to the outside world.

That is to say that the caller, somebody who invokes this procedure to buy this pair of tickets

never gets to see a state where one of the tickets is purchased and one of the tickets is not purchased.

It either looks like the ticket has been completely purchased or it has not been purchased at all, so we want it to be isolated.

And we also want it to be recoverable.

We want it to be the case that if we crash halfway through here, after we have bought the ticket on JetBlue, if the whole system crashes at that point, when the system comes back up, we do not want the system to be in some halfway state where we paid the money for the JetBlue ticket and have not paid the money for the USAir ticket.

So we should either complete the transaction or we should completely roll back the transaction and abort it.

So this is this notion of nested atomic actions, which is we have this outer action and then it has these two actions that are nested within it.

And each of these actions is, in and of itself, an atomic action.

If you think about what's going on here for a minute, so if you think about what the sort of condition that we want is, we've got A and B.

And we want A to commit if B commits and we want B to commit if A commits.

Because, if A doesn't commit, we need both of these actions to definitely commit or definitely not commit.

Otherwise, we're kind of in trouble for this example.

But the way that I've worded this, it kind of sounds impossible, like how is it A is waiting for B to commit and B is waiting for A to commit so how are we ever going to make any progress?

And the trick is that we're going to introduce a third-party, something that is in charge of deciding whether or not the entire action has committed.

So we're going to introduce a node S which we call the "supervisor".

And S is in charge of deciding whether or not the entire action, the action with both A and B

inside of it actually commits.

So let's see how that works because just seeing that we still have to have some way of S knowing that A is ready to commit and S knowing that B is ready to commit.

And we still need some way of making it so that A and B both make the decision to commit at exactly the same time.

So let's see how we can go about doing that.

The idea is that we're going to introduce something we call "tentative commits".

So what we want to do is get A and B to a point where they're both exactly ready to commit but they haven't yet actually committed their results.

So they haven't actually made their results available, but they want to give up control to S as to the instant that they'll actually commit.

So what we're trying to get is a way in which S can instantaneously make a decision about whether both A and B commit or neither A or B commits.

And so what we're going to do is, the idea with a tentative commit is we're going to run A and B until they tentatively commit.

And what that means is that they're going to do, so we want A and B to do everything except actually commit.

So what does that mean?

It means that A and B are going to read all of the data that they would normally read, they're going to write all the things that they would normally write.

If we are using a locking protocol they would acquire all of the locks that they would need to acquire in order to process the transaction, process their part of the action.

But they're not actually going to commit.

So not committing means that, in particular, they are not going to expose their results out to the outside world.

So we say they don't expose results beyond S.

So a tentatively committed transaction, I'm just going to write as TC, says that it's not going to expose any of the results of its action outside of S.

If we're using a locking protocol, how do we prevent one of these nested actions from being able to expose its results?

Or, how do we make it so that it doesn't expose its results or it makes its results visible outside?

Yeah.

**STUDENT:** Whenever a stop action wants to commit, we would just move the lock to its higher level of action, to the action it belongs to.

Right.

That's essentially what we're going to want to do.

If we just want to make it so that the action's results aren't visible, we're just going to make it so that that action doesn't release any of its locks.

If it doesn't release its locks then nobody else can get locks on any of the data that it updated.

And, therefore, none of its updates will be visible.

The solution that was proposed here is what we're going to work up to, which is that basically we want to make sure that if we want S to be able to see the results of a tentatively committed sub-action, which we will and I'll explain why.

Then what we're going to do is we're going to hand the locks off from the sub-action up to S, the superior action.

We're work through how that works.

The way that this is going to work, this is going to allow us to get, so we can draw a graph that looks like this.

We say S, we've got some action A, we've got some action B, and we may, in principle, have other actions that are sub-actions of these sub-actions. And what we're going to do is we're

going to draw a graph where we put arrows pointing from the sub-action up to its parent action, the action that it is depending on.

And we're going to label the states, we can label each one of these actions in this graph as either tentatively committed.

Or, if it's not tentatively committed, it hasn't finished doing all of its processing yet we might label it as pending.

Let's look in a little bit more detail about how we might actually get this tentative commit thing to work.

And hopefully that will make it a little bit more clear what's going on here.

And, in particular, what I want to do is I want to look at the way in which a log action on this machine might be being maintained.

And this is really going to help us get at how we do recovery via logging.

This is for these nested actions.

Suppose we have some log and this has actions in it.

When S first starts running the transaction, when the transaction first starts running this supervisor module writes a begin transaction message.

And then what it's going to do is it's going to invoke each one of these subatomic actions.

And each one of those subatomic actions is going to write a begin record as well.

And then there's going to be some processing, so these actions are going to execute, they're going to obtain some locks and they are going to update some data.

We're going to write those log records for that data that was updated into the log.

And then, at some point later, we will see tentative commits for A and tentative commits for B.

And then finally what we'll do is, once those guys are tentatively committed, we'll write the commit record for S.

Now let's look at what we can say at various points sort of during it.

If you think of this log as being a timeline about when things happen in the system, let's look and see what we can say at various points.

One thing we can say is that this time when this commit S record was written, this is the commit point for this entire transaction.

So we say this is the commit point for both A and B and anything else that maybe S did.

If you remember, what the commit point is, it's sort of the point of no return.

Once we reach the commit point we've guaranteed that this action is going to persist.

Even if the system crashes, it will recover in the state as though that action had taken effect.

And if we crash prior to the commit point then what we want to guaranty is that this action was not visible.

When we recover we're going to undo any effects that anything in this action did.

If we were to crash right here, just before we wrote the commit record, then we would undo this whole action.

And if we were to crash any time after we write the commit record then we're going to make sure that we redo any updates that the action may have needed to do.

We're going to guaranty that this action is forced to disc.

But notice that there are these two tentative commit records.

What do these tentative commit records correspond to?

What these tentative commit records mean is that A and B, this buy JetBlue and buy USAir, did all of the work that they had to do.

So, in particular, it means that neither A or B is going to have to acquire anymore locks, they're not going to write anymore data.

They're at exactly this point where they're ready to commit.

And the thing that's going to make them commit is the writing of this commit S log record.

Effectively, A and B, the sort of whether or not A and B commit is now out of control of A and B once they write this log record.

And it's completely in the hands of this outer supervisor module S.

And so the outer supervisor module S, of course it can commit, but you also have to realize that it's also OK if this outer module aborts.

And if it aborts then it will write an abort log record and we will have to go and undo the effects of the whole thing.

But we can still do that because we haven't, as of this point here, actually exposed any results outside of this action.

So there is nobody else who has seen the effects of this thing.

We're still isolated with respect to any outside action that might be running on the system.

So it's OK if we abort any time up to this commit record.

It also means that after this commit point, this is sort of the point where we're going to start exposing results.

If we're locking we're going to release our write locks after this commit point.

And the act of releasing the locks is what's going to make it so that other actions outside of this system can see the effects of A and B running.

Just to make it clear, we say A and B commit or abort when S commits or aborts.

I have just written CA here for commit or abort.

There is one other little detail that we need to point out which is that S can commit even if A or B fail.

So this may seem a little bit counterintuitive, but the intuition here is that this action S, suppose that S tries to run A and it cannot get a hold of the tickets on JetBlue that it wanted.

Well, S is free to go and try and make a reservation on some other airline that also satisfies the user's request.

So the fact that A failed doesn't say anything about whether or not S is necessarily going to fail.

S still gets to decide whether it fails or not.

And this is kind of an important property because it means that these actions that are running inside of S are actually, while they are running, are isolated with respect to S and the other actions that are running on S.

Any of the updates that they make while they're running aren't seen by S or any of the other actions.

This is just one little thing to keep in mind.

There is one last detail that I've sort of brushed over here that we hinted at a little bit a minute ago.

And that's the problem which is what if A and B conflict with each other?

We said that A and B might both update the same bank account balance, right?

Well, we have a little bit of a problem if that happens because I've got my S and I've got my A and my B.

And it may be the case that suppose we start running A first, A gets the lock on the bank account and then B starts running and tries to get the lock on the bank account.

And it waits for A because A is still holding this lock on the bank account.

So we may have this situation where B is waiting for A to release the lock on the bank account.

But the way that I've described this so far, it may not be clear that B is ever going to be able to actually obtain the lock.

Because we said that these tentatively committed actions don't release their locks until after the commit point has been reached.

It turns out that we need to sort of modify that statement just a little bit.

And that kind of gets to the comment that was made before which points this out.

What we want to say is that when an action like A tentatively commits, we want to make it so that S and its children can see A's updates.

After A has tentatively committed, the other actions that are underneath S should be able to see the updates that A made.

So A is going to go ahead and withdraw the money from the bank account, and then it's going to say OK, I'm done, I've withdrawn my money.

And now any other action within S that needs to run that maybe also needs to update the bank account will be allowed to go ahead and do that.

So B could go ahead and run and update the bank account.

Notice that we haven't said that A's updates are visible outside of S.

Nobody else gets to see that this bank account balance got changed.

It's just the action B that gets to see that the bank account balance is going to change.

Effectively, what this amounts to is that when A tentatively commits it assigns all of its locks up to the S action.

That's as much as I'm going to say about this notion of nested atomic actions.

What this has given us is this way to have, on a single site, one action that is composed of multiple sub-actions where those sub-actions are isolated from each other.

And all of the sub-actions either commit or abort together as a batch.

But we said what we ultimately wanted was the ability to do this across multiple sites.

Just to draw a simple architecture diagram about the multi-site case seeing what this looks like, suppose we have some action S and suppose we still have our A and our B.

Now, just to conceptualize this, suppose that there is some network in the middle of these things.

And this is a best-effort network so it has all the problems that we talked about that best-effort networks have.

It has congestion, it has delays, it can lose packets.

And the way to think about these things is that these actions are going to interact with each other.

These nodes are going to interact with each other using RPCs.

And they're just going to send actions, they're just going to send requests to each other and responses over these links.

So S is going to send RPC to A saying reserve a seat for me, for example, and then A is going to send a reply back saying OK, I went ahead and reserved that seat.

So what I want to do is sort of go from this informal description of what we want in the multi-site case to an actual description of how the protocol works so you guys can see that there are some pretty subtle details that are involved in this.

And it's worth pointing out that this situation is fairly similar to many of the things that, some of the problems that you're going to have to deal with in the context of the design project so it probably is a good idea for you to pay attention.

So let's talk about how this protocol would work.

The idea here is we want to provide -- Suppose we're still in this same travel site example.

The client wants to make these reservations over this network, and the protocol we're going to use is a protocol called "two-phase commit".

Suppose we have our node S which is the coordinator node, the node that represents the travel site, and we also have our two worker nodes that correspond to JetBlue and USAir A and B.

I just want to show that each of these sites is going to maintain a bit of a log that reflects the state of the actions that it's running.

So I'm going to show the state of the log on each of these nodes.

Suppose at some time S starts executing this action, let's call it T, so it's going to write a log record that says start T, and then it's going to request that each of the subordinate nodes, the sub-nodes goes ahead and does the processing, say purchases the ticket that it wants.

It is going to send a message here saying, for example, say the message consists of do something, do X, purchase this ticket.

And, at the same time, it may also send a message to B telling it to do something else.

Say, for example, do Y.

Now what's going to happen is that each of these guys is going to receive this request to do something, so it's going to write a log record that says start.

It's going to keep some information about what it started doing, it's going to say X, and it's going to remember that maybe this was a part of transaction T.

And, similarly, this guy is going to start Y which was a part of transaction T.

And then these two As and Bs now are just going to start executing the action that they were asked to execute.

So they are going to, for example, purchase the ticket.

They're going to run.

They're going to acquire whatever locks that they need to process and then they are, at some point, going to enter the tentatively committed state.

The same thing is going to happen over here.

They are both going to run these actions.

When they reach the tentatively committed state, what they're going to do is they're going to send a message back that says something like did X?

This message is sometimes called a vote.

Basically it says whether or not this site agrees that it was able to finish the work that it was able to do.

So if it votes yes that means, yes, I'm done, I'm ready to go.

And if it votes no that means sorry, I couldn't do the thing that you wanted.

So similarly B is going to send back its vote that says did Y.

So let's suppose, in both these cases, both of these actions were able to do the work that they wanted to do.

At this point now what's going to happen is that S is going to look at the votes from the actions that it is tasked, and it's going to decide whether or not this action is going to commit or is not going to commit.

So S is the one that's responsible for deciding whether or not this entire action commits.

And suppose it decides it's going to commit, it's going to write a record that says commit this transaction T.

So this is now the commit point.

As soon as S writes the commit record, that means this action is going to commit, it's going to be made visible to the outside world, all the work has been done.

And it's OK if it does this because A and B are both in the tentatively committed state.

They've said I'm ready to go, I've done all the work I need to do, I can commit whenever you tell me it's OK to commit.

So, after this point, everything is going to commit.

The reason this is called the two-phase commit protocol, typically this part is called phase one where we're deciding whether or not we agree to commit.

And then here in this next step we enter phase two.

So what do we have to do in phase two?

Notice that these guys have tentatively committed.

A and B don't actually know whether or not the action has committed so they don't actually know whether they should release their locks and make their updates visible to the outside world.

So we need to make sure that S tells A and B that OK, this action is done, I've committed and

it's OK for you to also go ahead and commit and expose your results to the outside world.

This is slightly different than the protocol we looked at before because these things are on different machines and so we have to pass information from S to A and B to let them know that this action is ready to go.

S is going to send a message to A saying commit, A is going to write a log record that says commit, and then it's going to do the same thing, send the message to B that says commit.

And, similarly, B is going to write a log record that says commit.

This is the basic two-phase commit protocol.

If you count the number of messages that you see here, if you have N sites, the number of messages you have to send in two-phase commit is 3N in the basic protocol without any loss.

Notice I haven't said anything about what happens when a message is lost.

And remember these best-effort networks have this property that messages can be lost, we can lose data, so we want to make sure that we understand how this protocol works in the face of data being lost.

The other thing that we need to do is to make sure that we understand how this protocol works in the event that either S crashes or A and B crash at different points in the execution of the protocol.

So that's what we're going to talk through now, sort of these nitty-gritty details about how we actually get this thing to work in the face of these properties that best-effort networks introduce.

To remind you guys how we deal with loss in best-effort networks, I am just going to very quickly review exactly once RPC protocol that we talked about a few weeks ago.

Exactly once RPC remember is a way to make it so that a procedure call gets executed once, a remote procedure call gets executed once, and only once, between a client and a server.

So if we've got our client and our server, what we do is keep at the client, we keep a list of messages, at the server we keep a list of "nonces." The client sends a request, a message, for example, to the server asking it to do something and it attaches a nonce to it, say N1. So the

client puts in its message table message, N1 stores that information.

The server receives this request, it stores the nonce in its nonce table and, one, sends an acknowledgement and processes the request.

The acknowledgement comes back, and maybe it's lost because these are best-ever networks.

Remember we have this timeout.

After some timeout period the client resends.

This is message, N1 again.

When the message gets resent, the server checks to see if this message is already in its nonce table.

If it is, it doesn't process the message again but it sends the [ACE?] for that message.

Now this [ACE?] message gets received, the client crosses it off its message list because it knows it's done processing.

That's the basic exactly once RPC protocol.

And what this guarantees is that this persistent client is going to retry sending the request until it gets an acknowledgement.

And the problem with that is that it can generate multiple, the server can hear this message multiple times so the server uses this nonce table to filter out duplicate messages.

Let's see how we can use this notion of this exactly once in our two-phase commit protocol.

I'm going to erase this and just redraw a similar example with a [LOSI?] protocol instead of a [LOSLS?] protocol.

And just to sort of make the notation a little bit simpler, let's now just suppose we have one worker site A.

We're not going to show A or B.

But this generalizes completely to as many As, Bs, Cs as we want.

What's going to happen now is, what I want to do is I want to keep a list of pending actions at S, as well as the log of NS.

And I'm also at A going to keep a list of pending actions and the log at A.

At some point S is going to go ahead and start processing the transaction again and it's going to write start T.

It's going to add T to its list of pending actions.

And then it's going to send this message that says do X to A.

Of course, it may be the case that this message gets lost because this is a [LOSI?] network.

So we're just going to use our persistent.

We're going to make S persistently retry so some time later, after some timeout, it's going to resend this do X message.

And it knows that it needs to timeout because it sees that there is this action here that's still pending.

Now this site A is going to receive this request.

It's going to add X for transaction T to its pending list, it's going to write its start X of T record, and it's going to go ahead and process just like it did before until it tentatively commits.

And now, once it is tentatively committed, it's going to go ahead and mark this in its pending table this action is tentatively committed and it's going to send a request back that says did X.

Of course, this request can also be lost.

So, again, remember we have the server persistently retrying.

So, at some point, it didn't hear this did X request.

It's just going to say hey, do it again.

And now when A receives this request, it's going to look up in its pending table, it's going to see that it has already tentatively committed this action.

So it's just going to not process the action at all, it's just going to send back the request that says, this should say do X, it's going to say did X.

Now, once S has received the tentative commits from all of the other actions, it can go ahead and write its commit record and we can go ahead and enter phase two, just like we did before.

You can kind of see how this is going to work out.

The process is just going to continue in the same way.

After S writes its commit record, it's going to go ahead and send the commit message.

And, of course, it is possible for the commit message to be lost.

So, in this case, notice that the server doesn't actually know whether the commit message has been lost or not.

And we haven't shown A sending out any responses back to the server after the commit message has been sent.

So that means that A is the one that actually has to retry in this case.

Now, A is just going to say hey, S, I did X.

And that's OK.

Now what's going to happen is S is going to go ahead and look up in its table, when it wrote the commit message to change the state of this message to committed, this transaction to committed, so S is just going to look up in its table, see that the transaction is committed and is going to send this message again.

And now, hopefully, this time it gets through.

And S can go ahead and change the state of this action to committed, it can release its locks and the protocol is done.

One thing to note is that as soon as A knows that the action is committed, it doesn't need to keep any more state about the action around anymore.

It knows it's committed.

That means that S has definitely heard about the fact that it did X.

And so A can go ahead and just forget any information it had in this pending transaction table about the action.

We haven't quite solved the problem but we still, in S, have to keep this information around about the fact that the transaction committed because we never actually know, in S, whether this final commit message got through or not.

So it's always possible that A could re-request the state of transaction T.

And S needs to be able to answer that correctly.

So this complicates this a little bit, and there are a couple of solutions that people have proposed.

The obvious one is we just add an extra round of acknowledgements onto the end of this.

So that's a simple thing we can do, is just have A acknowledge that it heard the message.

And then, as soon as S has heard acknowledgements from all of its subordinates, it can go ahead and delete the information.

There is another variant of this, which is talked about a little bit in the text, something called presumed commit where basically if A doesn't know anything about the action, it assumes that the action committed.

So A can discard the fact about any committed actions.

Getting that to work is a little bit trickier.

And the details of it are a little bit complicated but you can sort of see the idea.

I just want to quickly spend a couple of minutes talking about what happens in the case when these systems crash during different phases of execution so you guys can get a sense of how recovery would work in this environment.

Let's suppose that S crashes.

And there are two situations we're worried about.

Either it crashes before commit or it crashes after commit.

If it crashes before commit that means that, S is the sort of lead transaction here, we want to treat this just like we would treat this in sort of traditional recoverable systems.

So if we crash before the main commit, well, what we want to do is undo the effects of this transaction completely.

So we're going to undo T.

Notice, however, that if there are multiple As, Bs and Cs it may be the case that S crashes and some of the subordinates are still processing messages and send did finish processing requests to S.

That means when S crashes it recovers, it comes back up, it undoes the transaction and it remembers that T aborted.

So it puts T in its transaction table so that when it gets a request from somebody saying hey, I finished doing this part of transaction T, it can tell that guy oh, by the way, that transaction aborted.

Now, suppose that we crashed after the commit, well, you know what's going to happen.

We want this transaction to be durable.

We said that the commit is the commit point, we want this thing to appear to have happened.

So we need to make sure that we run redo on T and that we remember that T committed.

Now, if A crashes, the situation is a little bit easier.

Basically, there are only two situations we have to worry about in A.

It's either before or after we've gone into the tentative commit state.

If it's before the tentative commit state, well, we're just going to undo the effects of this transaction completely.

We're going to roll it back and basically going to forget about it.

And it's going to be S's responsibility to try and redo this action with us, if it wants to, or S may

go ask somebody else to do this part of the action.

If we crashed after the tentative commit, though, remember that at this point it's up to S.

This is A crashes.

At this point it's up to S.

So, after we've reached the tentative commit, this action needs to go ahead and check with S and see what the final outcome of this thing was.

It crashes, it comes back up, it sees it was in the tentative commit state for T, and so it sends a message to S saying hey, whatever happened to T?

And then S sends a response back if it knows.

This is the basic outline of how we do crash recovery and how we deal with lost messages in this two-phase commit protocol.

With the last few minutes, I want to talk about something.

This two-phase commit protocol seems really great.

It seems like we got in this way we have actions that are distributed across multiple sites.

We've made it so that if the action commits, we have this nice property that if the action commits, we can make it so that either A or B don't commit or they definitely both commit.

We have this way in which S is the ultimate arbiter and authority of what commits.

This seems like a really nice system that we built.

And it is.

And it has all these great properties, but there is one property that it doesn't have.

The question is, say in this environment with A and B, do they make their results visible at the same time.

I had written commit before.

And, in some sense, they do commit at the same time because they're going to definitely

commit at the point that this record gets written.

But the answer to the question are there results visible, the question is are there results visible at the same time, if you stare at this for a little while you realize no because their results become visible at the point at which these A and B receive the commit message from S.

Not at the point that S writes the commit record.

So they're going to expose their results at a slightly different point in time.

And, in fact, it could be a long time because it may be that A crashed after it went into the tentative commit record and it took it two days to recover.

And then it came back up and then it checks with S and says hey, whatever happened to T?

So it could be a very long time before, say, A commits, makes its results visible, whereas B may have done immediately after it received the first commit message from S.

You might ask the question is it possible to guaranty that A and B expose their results to the outside world at exactly the same instant.

And the answer to this question, it turns out, is no.

And this is kind of a fundamental result.

The reason for this, let me just give you, there's a simple example that's talked about in the book.

It's called the "two generals problem".

And the idea is as follows.

Suppose there are two generals and they have their armies.

They've flanking somebody who they are attacking and they're on the sides of a valley and they're both going to dive into the valley with the armies at the same time.

They're trying to agree what time they're going to do this at.

And they want to make sure they both attack at the same time because, if they don't attack at the same time, they're afraid that one of them will lose.

So the only way they have to communicate with each other is by sending these messengers, say, across the valleys.

They have some guy who runs across.

But, of course, this guy may not make it.

He may collapse from exhaustion or he may get shot or whatever.

So the first general sends a runner out to the second general that says we'll attack at dawn.

And maybe that runner gets through.

And then the second general sends a runner back that says yes, we'll attack at dawn.

And maybe that runner gets through, or maybe he doesn't.

The second runner doesn't get through, and the first general says man, I don't know if the second general heard about this or not.

So he sends another runner that says we'll attack at dawn.

And the second general says I already sent a runner but I guess he didn't get through and he sends another one back.

And, ultimately, they both need to agree that they'll both attack at dawn so you need a certain number of runners to successfully get through in order for this to happen.

The issue here, though, is that suppose the general have a fixed number of runners and they want to say what's the maximum number of runners that I could possibly ever need in order to agree on this thing?

And, if you think about this for a minute, you will see that there's no finite bound on the number of runners that could possibly be needed, because it's always the case that a huge number of runners could be lost because this is this best-effort network where there's always some probability of something being lost.

There is always this infinitesimal little chance that a million runners in a row wouldn't make it through.

So this is essentially the two generals problem.

And what it says is it's impossible for these two guys to guaranty that they will achieve consensus about something using a fixed number of messages, using a fixed number of runners.

That means, in the case of two-phase commit, what that suggests is that this S or one of these clients may need to continually retransmit, say, an infinite number of times, a very large number of times before its request is actually processed.

Because there's always this sort of small chance of a message being lost by the best-effort network.

So that's the two generals problem and it's just sort of a nice result to keep in mind.

It's one of those results in computer science that sometimes it turns out to be important.

In practice, in this kind of environment it doesn't matter that much.

The probabilities of loss are small.

And so most of the time this is going to achieve consensus after a limited number of messages.

So that's it for our discussion of fault tolerance and recovery.

Next time we're going to start talking about security and protection of information.

We will see you on Monday.