MITOCW | ocw-6.033-lec20

0:00:00 All right, you guys, let's go ahead and get started. 0:00:04.466 I am back. I know you guys all missed me. 0:00:07.968 Just a couple of announcements. Since you guys did not have 0:00:13.048 recitation this week, I want to make sure that you 0:00:17.338 guys remember the Design Proposal 2 reports are due next 0:00:22.155 tomorrow in class, as is Hands On 6. 0:00:25.22 Also, Quiz 2 is graded and is ready to be handed back. 0:00:31 If you go to office hours with your TA this afternoon, 0:00:34.548 you can pick it up or you can get it in class tomorrow. 0:00:38.164 We will post the statistics as soon as we get them. 0:00:41.511 We are still waiting to get the scores from one of the TAs into 0:00:45.663 the thing, so I do not want to say anything until we know for 0:00:49.68 sure. All right. 0:00:50.684 What we have seen so far, we saw last time, 0:00:53.496 we talked about this notion of transactions, 0:00:56.375 we talked about the notion of making actions durable and 0:01:00.058 consistent. This time what we are going to 0:01:04.866 do is look at a related topic, a slightly more advanced topic 0:01:10.598 that is more related to the notion of atomic actions that we 0:01:16.235 spent two or three lectures previous to the one about 0:01:21.203 transactions talking about. If you remember, 0:01:25.312 an atomic action, we say it is both recoverable 0:01:29.707 and isolated. 0:01:32 0:01:37 When we say an action is recoverable, that means, 0:01:39.773 remember, it either all happens or it does not happen at all. 0:01:43.241 When we say an action is isolated, it means, 0:01:45.726 from the point of view of other actions that are running 0:01:48.904 concurrently in the system, the action appears to only be 0:01:52.14 one unit. So then none of the 0:01:53.758 intermediate states of the action is visible to any other 0:01:56.994 actions that are running concurrently. 0:02:00 And we talked about the use of the logging protocol to allow us 0:02:03.497 to recover actions. We also, in the text and in 0:02:06.092 class, talked briefly about version histories as an 0:02:08.912 alternative way that we can recover actions. 0:02:11.338 And we talked about how to do isolation. 0:02:13.538 We talked about several different methods for doing 0:02:16.358 isolation. We spent a while talking about 0:02:18.615 locking as a mechanism that we use to isolate actions from each 0:02:22.112 other. What we are going to talk about 0:02:24.199 today is a related topic, and it has to do with when you 0:02:27.302 have actions that are spread across multiple sites, 0:02:30.123 multiple different computers. And this is going to tie 0:02:34.578 together some of the concepts that we learned in the previous 0:02:38.526 section on networking with the more recent stuff that we have 0:02:42.473 been talking about with atomic action. 0:02:44.907 The topic today is multi-site atomicity. 0:02:48 0:02:53 And just to give you a simple example of what we mean by a 0:02:56.754 situation in which you might care about multi-site atomicity, 0:03:00.706 suppose that you are running a travel website. 0:03:04 You have some travel site, and that travel site you have 0:03:08.384 negotiated agreements with various different people who 0:03:12.688 sell airline tickets. Maybe you have agreements with 0:03:16.753 Jet Blue and USAir and some other set of airlines. 0:03:20.659 And you want to make it so that when somebody purchases a 0:03:25.123 ticket, they may purchase a set of flights that are purchased 0:03:29.905 together as one unit where there are different flights on 0:03:34.369 different airlines. I might want to purchase a 0:03:38.754 flight from here to San Francisco on JetBlue and then a 0:03:42.397 flight from San Francisco back to here on USAir. 0:03:45.568 Sometimes people want to use multiple different vendors when 0:03:49.549 they are purchasing their tickets. 0:03:51.775 And it may be the way that your travel site talks to JetBlue and 0:03:56.026 USAir is over the Internet. They use some remote procedure 0:03:59.872 call to ask JetBlue or USAir to reserve a seat on their behalf. 0:04:05 But when somebody is using your website, you want to create the 0:04:07.983 illusion that these reservations that are made on the website are 0:04:11.062 sort of one atomic unit. You do not want it to be that a 0:04:13.709 person gets a reservation on JetBlue and does not get a 0:04:16.307 reservation on USAir. You want them to make a 0:04:18.424 reservation and get all of the reservation or none of the 0:04:21.119 reservation. In order to make that work, 0:04:22.995 we are going to need some sort of special support from the 0:04:25.738 travel site, JetBlue and USAir. Suppose, for example, 0:04:29.176 I did not have any special support from the JetBlue website 0:04:32.425 and I said I want to purchase this ticket to San Francisco. 0:04:35.674 JetBlue goes ahead and says I am ready and I purchased that 0:04:38.923 ticket for you. And then suppose we cannot get 0:04:41.443 this reservation on USAir. Now we are in trouble. 0:04:44.132 We need some way to back out of the action with JetBlue and say 0:04:47.605 wait, never mind, I did not mean to actually 0:04:50.014 purchase that ticket because I could not complete the rest of 0:04:53.375 my purchase. We are going to talk about how 0:04:56.486 to provide this kind of, what we call, 0:04:58.318 multi-site atomicity where we want this whole action that 0:05:01.092 includes purchases of tickets from these different websites to 0:05:04.114 appear as though it was one atomic action. 0:05:06.145 Particularly we want to make sure that this whole thing is 0:05:08.969 both recoverable and isolated. 0:05:11 0:05:18 To sneak up on this topic, because there are a bunch of 0:05:20.454 different issues that are going to come up here and are going to 0:05:23.318 be a little bit complicated, we are going to start by 0:05:25.681 looking at a simpler version of this problem. 0:05:28 Let's suppose that we have all of these things running on the 0:05:31.165 same computer together, that is they are not connected 0:05:33.961 over the Internet, there is not this possibility 0:05:36.44 of a message being lost like there would be on the Internet. 0:05:39.552 So we are going to look at these things as though they are 0:05:42.559 all running together on the same computer. 0:05:44.722 In that case, we call these actions that are 0:05:46.99 running together on the same computer "nested atomic 0:05:49.681 actions". 0:05:51 0:05:57 Once we see how this works on just the single computer case 0:06:01.772 then we will sort of look and see how it gets more complicated 0:06:06.791 when we extend it out to the multi-computer case. 0:06:10.74 Let's simplify this a little bit. 0:06:13.373 Suppose we had our buy ticket procedure looking something like 0:06:18.392 begin, buy on JetBlue, then buy on USAir and then end. 0:06:22.753 I am just going to call these two things A and B for now just 0:06:27.689 to sort of simplify it so I do not have to write JetBlue and 0:06:32.544 USAir over and over again. And the property that we want 0:06:39.362 is that each one of these actions, in and of itself, 0:06:45.275 should be atomic with respect to the other actions. 0:06:51.072 We want atomic with respect to each other. 0:06:55.826 That is one property we want.

other actions. It does not have the kind of atomic with respect to each other properties that we want. We want that because, 0:07:01.067 for example, suppose that buying JetBlue 0:07:03.05 requires me to provide some credit card number or debit card 0:07:06.05 number that is going to go out and purchase the ticket. 0:07:08.796 Well, I do not want to have the action that purchases from 0:07:11.694 JetBlue and the action that purchases from USAir 0:07:14.084 simultaneously decrementing my, say, bank account. 0:07:16.576 Because we saw how you could imagine getting into trouble 0:07:19.423 where they would both read the balance at the same time and 0:07:22.372 then both decrement and then both write at the same time. 0:07:25.22 You could get some mixed up balance left in your bank 0:07:27.864 account, for example. So if these things are, 0:07:31.676 let's say, for example, debiting a bank account, 0:07:35.101 you want to make sure that these actions are atomic with 0:07:39.109 respect to one another. You also want to make sure that 0:07:43.044 this whole thing is atomic with respect to the outside world. 0:07:48 0:07:54 That is to say that the caller, somebody who invokes this 0:07:57.36 procedure to buy this pair of tickets never gets to see a 0:08:00.72 state where one of the tickets is purchased and one of the 0:08:04.139 tickets is not purchased. It either looks like the ticket 0:08:07.485 has been completely purchased or it has not been purchased at 0:08:10.359 all, so we want it to be isolated. 0:08:11.94 And we also want it to be recoverable. 0:08:13.712 We want it to be the case that if we crash halfway through 0:08:16.443 here, after we have bought the ticket on JetBlue, 0:08:18.742 if the whole system crashes at that point, when the system 0:08:21.473 comes back up, we do not want the system to be 0:08:23.628 in some halfway state where we paid the money for the JetBlue 0:08:26.502 ticket and have not paid the money for the USAir ticket. 0:08:30 So we should either complete the transaction or we should 0:08:32.984 completely roll back the transaction and abort it. 0:08:35.595 So this is this notion of nested atomic actions, 0:08:38.1 which is we have this outer action and then it has these two 0:08:41.244 actions that are nested within it. 0:08:43.003 And each of these actions is, in and of itself, 0:08:45.454 an atomic action. 0:08:47 0:08:52 If you think about what's going on here for a minute, 0:08:57.652 so if you think about what the sort of condition that we want 0:09:04.173 is, we've got A and B. And we want A to commit if B 0:09:10.455 commits and we want B to commit if A commits. 0:09:15.885 Because, if A doesn't commit, we need both of these actions 0:09:23.042 to definitely commit or definitely not commit. 0:09:28.595 Otherwise, we're kind of in trouble for this example. 0:09:36 But the way that I've worded this, it kind of sounds 0:09:39.383 impossible, like how is it A is waiting for B to commit and B is 0:09:43.562 waiting for A to commit so how are we ever going to make any 0:09:47.476 progress? And the trick is that we're 0:09:49.864 going to introduce a third-party, something that is 0:09:53.181 in charge of deciding whether or not the entire action has 0:09:56.963 committed. So we're going to introduce a 0:09:59.55 node S which we call the "supervisor". 0:10:03 And S is in charge of deciding whether or not the entire 0:10:07.094 action, the action with both A and B inside of it actually 0:10:11.337 commits. So let's see how that works 0:10:13.942 because just seeing that we still have to have some way of S 0:10:18.334 knowing that A is ready to commit and S knowing that B is 0:10:22.503 ready to commit. And we still need some way of 0:10:25.853 making it so that A and B both make the decision to commit at 0:10:30.32 exactly the same time. So let's see how we can go 0:10:34.772 about doing that. 0:10:36 0:10:50 The idea is that we're going to introduce something we call 0:10:57.25 "tentative commits". So what we want to do is get A 0:11:02.018 and B to a point where they're both exactly ready to commit but 0:11:06.19 they haven't yet actually committed their results. 0:11:09.487 So they haven't actually made their results available, 0:11:13.053 but they want to give up control to S as to the instant 0:11:16.686 that they'll actually commit. So what we're trying to get is 0:11:20.656 a way in which S can instantaneously make a decision 0:11:24.088 about whether both A and B commit or neither A or B 0:11:27.452 commits. And so what we're going to do 0:11:31.225 is, the idea with a tentative commit is we're going to run A 0:11:35.752 and B until they tentatively commit. 0:11:38.438 And what that means is that they're going to do, 0:11:42.045 so we want A and B to do everything except actually 0:11:45.882 commit. So what does that mean? 0:11:48.184 It means that A and B are going to read all of the data that 0:11:52.712 they would normally read, they're going to write all the 0:11:56.932 things that they would normally write. 0:12:01 If we are using a locking protocol they would acquire all 0:12:04.914 of the locks that they would need to acquire in order to 0:12:08.759 process the transaction, process their part of the 0:12:12.185 action. But they're not actually going 0:12:14.772 to commit. So not committing means that, 0:12:17.498 in particular, they are not going to expose 0:12:20.434 their results out to the outside world. 0:12:24 0:12:32 So we say they don't expose results beyond S. 0:12:36.704 So a tentatively committed transaction, I'm just going to 0:12:42.691 write as TC, says that it's not going to expose any of the 0:12:48.786 results of its action outside of S. 0:12:52.421 If we're using a locking protocol, how do we prevent one 0:12:58.301 of these nested actions from being able to expose its 0:13:03.861 results? Or, how do we make it so that 0:13:08.188 it doesn't expose its results or it makes its results visible 0:13:12.716 outside? Yeah. 0:13:14 STUDENT: Whenever a stop action wants to commit, 0:13:17.455 we would just move the lock to its higher level of action, 0:13:21.647 to the action it belongs to. Right. 0:13:24.341 That's essentially what we're going to want to do. 0:13:27.127 If we just want to make it so that the action's results aren't 0:13:30.597 visible, we're just going to make it so that that action 0:13:33.725 doesn't release any of its locks. 0:13:36 If it doesn't release its locks then nobody else can get locks 0:13:39.597 on any of the data that it updated. 0:13:41.602 And, therefore, none of its updates will be 0:13:44.079 visible. The solution that was proposed 0:13:46.32 here is what we're going to work up to, which is that basically 0:13:49.976 we want to make sure that if we want S to be able to see the 0:13:53.456 results of a tentatively committed sub-action, 0:13:56.11 which we will and I'll explain why. 0:13:59 Then what we're going to do is we're going to hand the locks 0:14:03.009 off from the sub-action up to S, the superior action. 0:14:06.543 We're work through how that works. 0:14:08.786 The way that this is going to work, this is going to allow us 0:14:12.864 to get, so we can draw a graph that looks like this. 0:14:16.33 We say S, we've got some action A, we've got some action B, 0:14:20.271 and we may, in principle, have other actions that are 0:14:23.805 sub-actions of these sub-actions. 0:14:27 And what we're going to do is we're going to draw a graph 0:14:29.868 where we put arrows pointing from the sub-action up to its 0:14:32.787 parent action, the action that it is depending 0:14:35.092 on. And we're going to label the 0:14:36.68 states, we can label each one of these actions

in this graph as 0:14:39.856 either tentatively committed. Or, if it's not tentatively 0:14:42.724 committed, it hasn't finished doing all of its processing yet 0:14:45.797 we might label it as pending. 0:14:48 0:15:00 Let's look in a little bit more detail about how we might 0:15:05 actually get this tentative commit thing to work. 0:15:09.285 And hopefully that will make it a little bit more clear what's 0:15:14.732 going on here. And, in particular, 0:15:17.678 what I want to do is I want to look at the way in which a log 0:15:23.035 action on this machine might be being maintained. 0:15:27.321 And this is really going to help us get at how we do 0:15:31.875 recovery via logging. This is for these nested 0:15:38.103 actions. Suppose we have some log and 0:15:42.758 this has actions in it. When S first starts running the 0:15:49.741 transaction, when the transaction first starts running 0:15:56.594 this supervisor module writes a begin transaction message. 0:16:05 And then what it's going to do is it's going to invoke each one 0:16:09.654 of these subatomic actions. And each one of those subatomic 0:16:14.009 actions is going to write a begin record as well. 0:16:17.612 And then there's going to be some processing, 0:16:20.915 so these actions are going to execute, they're going to obtain 0:16:25.495 some locks and they are going to update some data. 0:16:30 We're going to write those log records for that data that was 0:16:35.26 updated into the log. And then, at some point later, 0:16:39.731 we will see tentative commits for A and tentative commits for 0:16:44.991 B. And then finally what we'll do 0:16:47.797 is, once those guys are tentatively committed, 0:16:51.742 we'll write the commit record for S. 0:16:54.81 Now let's look at what we can say at various points sort of 0:16:59.895 during it. If you think of this log as 0:17:04.111 being a timeline about when things happen in the system, 0:17:08.413 let's look and see what we can say at various points. 0:17:12.48 One thing we can say is that this time when this commit S 0:17:16.86 record was written, this is the commit point for 0:17:20.536 this entire transaction. So we say this is the commit 0:17:24.603 point for both A and B and anything else that maybe S did. 0:17:30 If you remember, what the commit point is, 0:17:32.534 it's sort of the point of no return. 0:17:34.697 Once we reach the commit point we've guaranteed that this 0:17:38.158 action is going to persist. Even if the system crashes, 0:17:41.496 it will recover in the state as though that action had taken 0:17:45.143 effect. And if we crash prior to the 0:17:47.306 commit point then what we want to guaranty is that this action 0:17:51.077 was not visible. When we recover we're going to 0:17:53.92 undo any effects that anything in this action did. 0:17:58 If we were to crash right here, just before we wrote the commit 0:18:01.659 record, then we would undo this whole action. 0:18:04.256 And if we were to crash any time after we write the commit 0:18:07.621 record then we're going to make sure that we redo any updates 0:18:11.163 that the action may have needed to do. 0:18:13.347 We're going to guaranty that this action is forced to disc. 0:18:16.77 But notice that there are these two tentative commit records. 0:18:20.312 What do these tentative commit records correspond to? 0:18:23.381 What these tentative commit records mean is that A and B, 0:18:26.687 this buy JetBlue and buy USAir, did all of the work that they 0:18:30.229 had to do. So, in particular, 0:18:33.047 it means that neither A or B is going to have to acquire anymore 0:18:36.715 locks, they're not going to write anymore data. 0:18:39.392 They're at exactly this point where they're ready to commit. 0:18:42.827 And the thing that's going to make them commit is the writing 0:18:46.32 of this commit S log record. Effectively, 0:18:48.648 A and B, the sort of whether or not A and B commit is now out of 0:18:52.316 control of A and B once they write this log record. 0:18:55.226 And it's completely in the hands of this outer supervisor 0:18:58.486 module S. And so the outer supervisor 0:19:01.363 module S, of course it can commit, but you also have to 0:19:04.09 realize that it's also OK if this outer module aborts. 0:19:06.767 And if it aborts then it will write an abort log record and we 0:19:09.848 will have to go and undo the effects of the whole thing. 0:19:12.626 But we can still do that because we haven't, 0:19:14.797 as of this point here, actually exposed any results 0:19:17.323 outside of this action. So there is nobody else who has 0:19:20.05 seen the effects of this thing. We're still isolated with 0:19:22.878 respect to any outside action that might be running on the 0:19:25.757 system. So it's OK if we abort any time 0:19:27.676 up to this commit record. It also means that after this 0:19:32.119 commit point, this is sort of the point where 0:19:35.336 we're going to start exposing results. 0:19:38.04 If we're locking we're going to release our write locks after 0:19:42.426 this commit point. And the act of releasing the 0:19:45.789 locks is what's going to make it so that other actions outside of 0:19:50.467 this system can see the effects of A and B running. 0:19:55 0:20:00 Just to make it clear, we say A and B commit or abort 0:20:08.187 when S commits or aborts. I have just written CA here for 0:20:17.004 commit or abort. There is one other little 0:20:23.459 detail that we need to point out which is that S can commit even 0:20:33.378 if A or B fail. So this may seem a little bit 0:20:38.996 counterintuitive, but the intuition here is that 0:20:42.232 this action S, suppose that S tries to run A 0:20:45.193 and it cannot get a hold of the tickets on JetBlue that it 0:20:49.118 wanted. Well, S is free to go and try 0:20:51.596 and make a reservation on some other airline that also 0:20:55.245 satisfies the user's request. So the fact that A failed 0:20:59.547 doesn't say anything about whether or not S is necessarily 0:21:03.075 going to fail. S still gets to decide whether 0:21:05.798 it fails or not. And this is kind of an 0:21:08.15 important property because it means that these actions that 0:21:11.74 are running inside of S are actually, while they are 0:21:14.897 running, are isolated with respect to S and the other 0:21:18.116 actions that are running on S. Any of the updates that they 0:21:21.705 make while they're running aren't seen by S or any of the 0:21:25.172 other actions. This is just one little thing 0:21:27.833 to keep in mind. 0:21:30 0:21:35 There is one last detail that I've sort of brushed over here 0:21:39.726 that we hinted at a little bit a minute ago. 0:21:43.171 And that's the problem which is what if A and B conflict with 0:21:47.977 each other? We said that A and B might both 0:21:51.342 update the same bank account balance, right? 0:21:54.787 Well, we have a little bit of a problem if that happens because 0:21:59.754 I've got my S and I've got my A and my B. 0:22:04 And it may be the case that suppose we start running A 0:22:07.503 first, A gets the lock on the bank account and then B starts 0:22:11.402 running and tries to get the lock on the bank account. 0:22:14.906 And it waits for A because A is still holding this lock on the 0:22:18.938 bank account. So we may have this situation 0:22:21.714 where B is waiting for A to release the lock on the bank 0:22:25.349 account. But the way that I've described 0:22:27.927 this so far, it may not be clear that B is ever going to be able 0:22:32.091 to actually obtain the lock. Because we said that these 0:22:36.534 tentatively committed actions don't release

their locks until 0:22:40.076 after the commit point has been reached. 0:22:42.378 It turns out that we need to sort of modify that statement 0:22:45.743 just a little bit. And that kind of gets to the 0:22:48.458 comment that was made before which points this out. 0:22:52 What we want to say is that when an action like A 0:22:56.594 tentatively commits, we want to make it so that S 0:23:01.188 and its children can see A's updates. 0:23:04.634 After A has tentatively committed, the other actions 0:23:09.516 that are underneath S should be able to see the updates that A 0:23:15.355 made. So A is going to go ahead and 0:23:18.609 withdraw the money from the bank account, and then it's going to 0:23:24.639 say OK, I'm done, I've withdrawn my money. 0:23:30 And now any other action within S that needs to run that maybe 0:23:33.7 also needs to update the bank account will be allowed to go 0:23:37.218 ahead and do that. So B could go ahead and run and 0:23:40.191 update the bank account. Notice that we haven't said 0:23:43.284 that A's updates are visible outside of S. 0:23:45.772 Nobody else gets to see that this bank account balance got 0:23:49.229 changed. It's just the action B that 0:23:51.352 gets to see that the bank account balance is going to 0:23:54.507 change. Effectively, 0:23:55.659 what this amounts to is that when A tentatively commits it 0:23:59.117 assigns all of its locks up to the S action. 0:24:03 0:24:07 That's as much as I'm going to say about this notion of nested 0:24:11.425 atomic actions. What this has given us is this 0:24:14.69 way to have, on a single site, one action that is composed of 0:24:19.044 multiple sub-actions where those sub-actions are isolated from 0:24:23.47 each other. And all of the sub-actions 0:24:26.154 either commit or abort together as a batch. 0:24:30 But we said what we ultimately wanted was the ability to do 0:24:36.744 this across multiple sites. Just to draw a simple 0:24:42.325 architecture diagram about the multi-site case seeing what this 0:24:49.534 looks like, suppose we have some action S and suppose we still 0:24:56.627 have our A and our B. Now, just to conceptualize 0:25:02.402 this, suppose that there is some network in the middle of these 0:25:08.13 things. And this is a best-effort 0:25:11.086 network so it has all the problems that we talked about 0:25:16.076 that best-effort networks have. It has congestion, 0:25:20.603 it has delays, it can lose packets. 0:25:23.744 And the way to think about these things is that these 0:25:28.548 actions are going to interact with each other. 0:25:34 These nodes are going to interact with each other using 0:25:36.832 RPCs. And they're just going to send 0:25:38.668 actions, they're just going to send requests to each other and 0:25:41.868 responses over these links. So S is going to send RPC to A 0:25:44.857 saying reserve a seat for me, for example, 0:25:47.008 and then A is going to send a reply back saying OK, 0:25:49.631 I went ahead and reserved that seat. 0:25:51.467 So what I want to do is sort of go from this informal 0:25:54.194 description of what we want in the multi-site case to an actual 0:25:57.446 description of how the protocol works so you guys can see that 0:26:00.646 there are some pretty subtle details that are involved in 0:26:03.583 this. And it's worth pointing out 0:26:07.095 that this situation is fairly similar to many of the things 0:26:11.597 that, some of the problems that you're going to have to deal 0:26:16.176 with in the context of the design project so it probably is 0:26:20.677 a good idea for you to pay attention. 0:26:23.471 So let's talk about how this protocol would work. 0:26:27.197 The idea here is we want to provide -- 0:26:31 Suppose we're still in this same travel site example. 0:26:34.502 The client wants to make these reservations over this network, 0:26:38.611 and the protocol we're going to use is a protocol called 0:26:42.316 "two-phase commit". 0:26:44 0:26:53 Suppose we have our node S which is the coordinator node, 0:26:58.34 the node that represents the travel site, and we also have 0:27:03.775 our two worker nodes that correspond to JetBlue and USAir 0:27:09.115 A and B. I just want to show that each 0:27:12.644 of these sites is going to maintain a bit of a log that 0:27:17.793 reflects the state of the actions that it's running. 0:27:22.657 So I'm going to show the state of the log on each of these 0:27:28.092 nodes. Suppose at some time S starts 0:27:32.493 executing this action, let's call it T, 0:27:35.761 so it's going to write a log record that says start T, 0:27:40.319 and then it's going to request that each of the subordinate 0:27:45.307 nodes, the sub-nodes goes ahead and does the processing, 0:27:50.036 say purchases the ticket that it wants. 0:27:53.304 It is going to send a message here saying, for example, 0:27:57.948 say the message consists of do something, do X, 0:28:01.904 purchase this ticket. And, at the same time, 0:28:06.446 it may also send a message to B telling it to do something else. 0:28:10.586 Say, for example, do Y. 0:28:12.032 Now what's going to happen is that each of these guys is going 0:28:16.042 to receive this request to do something, so it's going to 0:28:19.723 write a log record that says start. 0:28:21.957 It's going to keep some information about what it 0:28:25.112 started doing, it's going to say X, 0:28:27.347 and it's going to remember that maybe this was a part of 0:28:30.962 transaction T. And, similarly, 0:28:33.878 this guy is going to start Y which was a part of transaction 0:28:37.334 T. And then these two As and Bs 0:28:39.091 now are just going to start executing the action that they 0:28:42.429 were asked to execute. So they are going to, 0:28:44.947 for example, purchase the ticket. 0:28:46.822 They're going to run. They're going to acquire 0:28:49.457 whatever locks that they need to process and then they are, 0:28:52.854 at some point, going to enter the tentatively 0:28:55.431 committed state. The same thing is going to 0:28:57.891 happen over here. They are both going to run 0:29:01.945 these actions. When they reach the tentatively 0:29:05.311 committed state, what they're going to do is 0:29:08.528 they're going to send a message back that says something like 0:29:13.017 did X? This message is sometimes 0:29:15.336 called a vote. Basically it says whether or 0:29:18.478 not this site agrees that it was able to finish the work that it 0:29:23.192 was able to do. So if it votes yes that means, 0:29:26.558 yes, I'm done, I'm ready to go. 0:29:30 And if it votes no that means sorry, I couldn't do the thing 0:29:33.776 that you wanted. So similarly B is going to send 0:29:36.784 back its vote that says did Y. So let's suppose, 0:29:39.791 in both these cases, both of these actions were able 0:29:43.056 to do the work that they wanted to do. 0:29:46 0:29:56 At this point now what's going to happen is that S is going to 0:30:00.211 look at the votes from the actions that it is tasked, 0:30:03.801 and it's going to decide whether or not this action is 0:30:07.461 going to commit or is not going to commit. 0:30:10.291 So S is the one that's responsible for deciding whether 0:30:14.02 or not this entire action commits. 0:30:16.298 And suppose it decides it's going to commit, 0:30:19.267 it's going to write a record that says commit this 0:30:22.65 transaction T. So this is now the commit 0:30:25.342 point. As soon as S writes the commit 0:30:28.718 record, that means this action is going to commit, 0:30:31.526 it's going to be made visible to the outside world, 0:30:34.39 all the work has been done. And

commit, clients not going to be made visible to the outside world, clients see all the work has been done. And it's OK if it does this 0:30:37.484 because A and B are both in the tentatively committed state. 0:30:40.864 They've said I'm ready to go, I've done all the work I need 0:30:44.187 to do, I can commit whenever you tell me it's OK to commit. 0:30:47.51 So, after this point, everything is going to commit. 0:30:50.432 The reason this is called the two-phase commit protocol, 0:30:53.583 typically this part is called phase one where we're deciding 0:30:56.963 whether or not we agree to commit. 0:31:00 And then here in this next step we enter phase two. 0:31:02.918 So what do we have to do in phase two? 0:31:05.077 Notice that these guys have tentatively committed. 0:31:07.937 A and B don't actually know whether or not the action has 0:31:11.206 committed so they don't actually know whether they should release 0:31:14.941 their locks and make their updates visible to the outside 0:31:18.21 world. So we need to make sure that S 0:31:20.311 tells A and B that OK, this action is done, 0:31:22.762 I've committed and it's OK for you to also go ahead and commit 0:31:26.322 and expose your results to the outside world. 0:31:30 This is slightly different than the protocol we looked at before 0:31:34.772 because these things are on different machines and so we 0:31:38.939 have to pass information from S to A and B to let them know that 0:31:43.712 this action is ready to go. S is going to send a message to 0:31:48.106 A saying commit, A is going to write a log 0:31:51.212 record that says commit, and then it's going to do the 0:31:55.227 same thing, send the message to B that says commit. 0:32:00 And, similarly, B is going to write a log 0:32:03.269 record that says commit. This is the basic two-phase 0:32:07.438 commit protocol. If you count the number of 0:32:10.871 messages that you see here, if you have N sites, 0:32:14.713 the number of messages you have to send in two-phase commit is 0:32:19.7 3N in the basic protocol without any loss. 0:32:23.051 Notice I haven't said anything about what happens when a 0:32:27.547 message is lost. And remember these best-effort 0:32:31.502 networks have this property that messages can be lost, 0:32:34.156 we can lose data, so we want to make sure that we 0:32:36.56 understand how this protocol works in the face of data being 0:32:39.515 lost. The other thing that we need to 0:32:41.318 do is to make sure that we understand how this protocol 0:32:44.023 works in the event that either S crashes or A and B crash at 0:32:46.978 different points in the execution of the protocol. 0:32:49.432 So that's what we're going to talk through now, 0:32:51.736 sort of these nitty-gritty details about how we actually 0:32:54.49 get this thing to work in the face of these properties that 0:32:57.395 best-effort networks introduce. 0:33:00 0:33:18 To remind you guys how we deal with loss in best-effort 0:33:21.857 networks, I am just going to very quickly review exactly once 0:33:26.142 RPC protocol that we talked about a few weeks ago. 0:33:30 Exactly once RPC remember is a way to make it so that a 0:33:34.362 procedure call gets executed once, a remote procedure call 0:33:38.967 gets executed once, and only once, 0:33:41.633 between a client and a server. So if we've got our client and 0:33:46.48 our server, what we do is keep at the client, 0:33:50.034 we keep a list of messages, at the server we keep a list of 0:33:54.72 "nonces." The client sends a request, a message, 0:33:58.517 for example, to the server asking it to do 0:34:01.829 something and it attaches a nonce to it, say N1. 0:34:07 So the client puts in its message table message, 0:34:11.059 N1 stores that information. The server receives this 0:34:15.465 request, it stores the nonce in its nonce table and, 0:34:19.87 one, sends an acknowledgement and processes the request. 0:34:24.621 The acknowledgement comes back, and maybe it's lost because 0:34:29.631 these are best-ever networks. Remember we have this timeout. 0:34:35.168 After some timeout period the client resends. 0:34:38.349 This is message, N1 again. 0:34:40.156 When the message gets resent, the server checks to see if 0:34:44.204 this message is already in its nonce table. 0:34:47.24 If it is, it doesn't process the message again but it sends 0:34:51.433 the ACK for that message. Now this ACK message gets 0:34:55.481 received, the client crosses it off its message list because it 0:34:59.963 knows it's done processing. That's the basic exactly once 0:35:04.828 RPC protocol. And what this guarantees is 0:35:07.349 that this persistent client is going to retry sending the 0:35:10.879 request until it gets an acknowledgement. 0:35:13.401 And the problem with that is that it can generate multiple, 0:35:17.057 the server can hear this message multiple times so the 0:35:20.398 server uses this nonce table to filter out duplicate messages. 0:35:25 0:35:35 Let's see how we can use this notion of this exactly once in 0:35:40.057 our two-phase commit protocol. I'm going to erase this and 0:35:44.942 just redraw a similar example with a [LOSI?] protocol instead 0:35:50.085 of a [LOSLS?] protocol. And just to sort of make the 0:35:54.457 notation a little bit simpler, let's now just suppose we have 0:35:59.599 one worker site A. We're not going to show A or B. 0:36:04.912 But this generalizes completely to as many As, 0:36:09.14 Bs, Cs as we want. What's going to happen now is, 0:36:13.651 what I want to do is I want to keep a list of pending actions 0:36:19.288 at S, as well as the log of NS. And I'm also at A going to keep 0:36:25.114 a list of pending actions and the log at A. 0:36:30 At some point S is going to go ahead and start processing the 0:36:34.972 transaction again and it's going to write start T. 0:36:39.033 It's going to add T to its list of pending actions. 0:36:43.176 And then it's going to send this message that says do X to 0:36:47.9 A. Of course, it may be the case 0:36:50.469 that this message gets lost because this is a [LOSI?] 0:36:54.779 network. So we're just going to use our 0:36:57.928 persistent. We're going to make S 0:37:01.683 persistently retry so some time later, after some timeout, 0:37:06.253 it's going to resend this do X message. 0:37:09.299 And it knows that it needs to timeout because it sees that 0:37:13.869 there is this action here that's still pending. 0:37:17.556 Now this site A is going to receive this request. 0:37:21.405 It's going to add X for transaction T to its pending 0:37:25.493 list, it's going to write its start X of T record, 0:37:29.421 and it's going to go ahead and process just like it did before 0:37:34.312 until it tentatively commits. And now, once it is tentatively 0:37:41.179 committed, it's going to go ahead and mark this in its 0:37:46.615 pending table this action is tentatively committed and it's 0:37:52.564 going to send a request back that says did X. 0:37:57.076 Of course, this request can also be lost. 0:38:02 So, again, remember we have the server persistently retrying. 0:38:05.628 So, at some point, it didn't hear this did X 0:38:08.228 request. It's just going to say hey, 0:38:10.345 do it again. And now when A receives this 0:38:12.764 request, it's going to look up in its pending table, 0:38:15.848 it's going to see that it has already tentatively committed 0:38:19.356 this action. So it's just going to not 0:38:21.593 process the action at all, it's just going to send back 0:38:24.859 the request that says, this should say do X, 0:38:27.46 it's going to say did X. Now, once S has received the 0:38:32.58 tentative commits from all of the other actions, 0:38:36.911 it can go

ahead and write its commit record and we can go 0:38:42.071 ahead and enter phase two, just like we did before. 0:38:46.679 You can kind of see how this is going to work out. 0:38:51.194 The process is just going to continue in the same way. 0:38:57 After S writes its commit record, it's going to go ahead 0:39:00.84 and send the commit message. And, of course, 0:39:03.842 it is possible for the commit message to be lost. 0:39:07.193 So, in this case, notice that the server doesn't 0:39:10.475 actually know whether the commit message has been lost or not. 0:39:14.734 And we haven't shown A sending out any responses back to the 0:39:18.853 server after the commit message has been sent. 0:39:21.995 So that means that A is the one that actually has to retry in 0:39:26.184 this case. Now, A is just going to say 0:39:30.111 hey, S, I did X. And that's OK. 0:39:32.457 Now what's going to happen is S is going to go ahead and look up 0:39:37.383 in its table, when it wrote the commit 0:39:40.277 message to change the state of this message to committed, 0:39:44.656 this transaction to committed, so S is just going to look up 0:39:49.27 in its table, see that the transaction is 0:39:52.398 committed and is going to send this message again. 0:39:56.229 And now, hopefully, this time it gets through. 0:40:01 And S can go ahead and change the state of this action to 0:40:04.551 committed, it can release its locks and the protocol is done. 0:40:08.357 One thing to note is that as soon as A knows that the action 0:40:12.099 is committed, it doesn't need to keep any 0:40:14.636 more state about the action around anymore. 0:40:17.3 It knows it's committed. That means that S has 0:40:20.154 definitely heard about the fact that it did X. 0:40:23.008 And so A can go ahead and just forget any information it had in 0:40:26.94 this pending transaction table about the action. 0:40:31 We haven't quite solved the problem but we still, 0:40:33.906 in S, have to keep this information around about the 0:40:36.994 fact that the transaction committed because we never 0:40:40.082 actually know, in S, whether this final commit 0:40:42.807 message got through or not. So it's always possible that A 0:40:46.258 could re-request the state of transaction T. 0:40:48.862 And S needs to be able to answer that correctly. 0:40:51.708 So this complicates this a little bit, and there are a 0:40:54.917 couple of solutions that people have proposed. 0:40:57.642 The obvious one is we just add an extra round of 0:41:00.488 acknowledgements onto the end of this. 0:41:04 So that's a simple thing we can do, is just have A acknowledge 0:41:06.998 that it heard the message. And then, as soon as S has 0:41:09.553 heard acknowledgements from all of its subordinates, 0:41:12.06 it can go ahead and delete the information. 0:41:14.124 There is another variant of this, which is talked about a 0:41:16.877 little bit in the text, something called presumed 0:41:19.236 commit where basically if A doesn't know anything about the 0:41:22.086 action, it assumes that the action committed. 0:41:24.249 So A can discard the fact about any committed actions. 0:41:26.854 Getting that to work is a little bit trickier. 0:41:30 And the details of it are a little bit complicated but you 0:41:34.384 can sort of see the idea. I just want to quickly spend a 0:41:38.615 couple of minutes talking about what happens in the case when 0:41:43.23 these systems crash during different phases of execution so 0:41:47.692 you guys can get a sense of how recovery would work in this 0:41:52.153 environment. Let's suppose that S crashes. 0:41:55.307 And there are two situations we're worried about. 0:42:00 Either it crashes before commit or it crashes after commit. 0:42:04.887 If it crashes before commit that means that, 0:42:08.511 S is the sort of lead transaction here, 0:42:11.713 we want to treat this just like we would treat this in sort of 0:42:16.853 traditional recoverable systems. So if we crash before the main 0:42:22.078 commit, well, what we want to do is undo the 0:42:25.702 effects of this transaction completely. 0:42:30 So we're going to undo T. Notice, however, 0:42:33.881 that if there are multiple As, Bs and Cs it may be the case 0:42:39.372 that S crashes and some of the subordinates are still 0:42:44.295 processing messages and send did finish processing requests to S. 0:42:50.355 That means when S crashes it recovers, it comes back up, 0:42:55.562 it undoes the transaction and it remembers that T aborted. 0:43:02 So it puts T in its transaction table so that when it gets a 0:43:06.089 request from somebody saying hey, I finished doing this part 0:43:10.178 of transaction T, it can tell that guy oh, 0:43:13.019 by the way, that transaction aborted. 0:43:15.514 Now, suppose that we crashed after the commit, 0:43:18.633 well, you know what's going to happen. 0:43:21.198 We want this transaction to be durable. 0:43:23.831 We said that the commit is the commit point, 0:43:26.811 we want this thing to appear to have happened. 0:43:29.93 So we need to make sure that we run redo on T and that we 0:43:33.811 remember that T committed. Now, if A crashes, 0:43:38.424 the situation is a little bit easier. 0:43:41.272 Basically, there are only two situations we have to worry 0:43:45.702 about in A. It's either before or after 0:43:48.708 we've gone into the tentative commit state. 0:43:52.031 If it's before the tentative commit state, 0:43:55.275 well, we're just going to undo the effects of this transaction 0:44:00.101 completely. We're going to roll it back and 0:44:04.411 basically going to forget about it. 0:44:07.055 And it's going to be S's responsibility to try and redo 0:44:11.255 this action with us, if it wants to, 0:44:13.977 or S may go ask somebody else to do this part of the action. 0:44:18.566 If we crashed after the tentative commit, 0:44:21.677 though, remember that at this point it's up to S. 0:44:25.411 This is A crashes. At this point it's up to S. 0:44:30 So, after we've reached the tentative commit, 0:44:32.863 this action needs to go ahead and check with S and see what 0:44:36.637 the final outcome of this thing was. 0:44:38.915 It crashes, it comes back up, it sees it was in the tentative 0:44:42.819 commit state for T, and so it sends a message to S 0:44:46.008 saying hey, whatever happened to T? 0:44:48.221 And then S sends a response back if it knows. 0:44:51.084 This is the basic outline of how we do crash recovery and how 0:44:54.989 we deal with lost messages in this two-phase commit protocol. 0:45:00 With the last few minutes, I want to talk about something. 0:45:03.294 This two-phase commit protocol seems really great. 0:45:06.127 It seems like we got in this way we have actions that are 0:45:09.364 distributed across multiple sites. 0:45:11.271 We've made it so that if the action commits, 0:45:13.757 we have this nice property that if the action commits, 0:45:16.82 we can make it so that either A or B don't commit or they 0:45:20.057 definitely both commit. We have this way in which S is 0:45:23.121 the ultimate arbiter and authority of what commits. 0:45:26.011 This seems like a really nice system that we built. 0:45:30 And it is. And it has all these great 0:45:36.146 properties, but there is one property that it doesn't have. 0:45:46.048 The question is, say in this environment with A 0:45:53.902 and B, do they make their results visible at the same 0:46:02.78 time. 0:46:05 0:46:10 I had written commit before. And, in some sense, 0:46:12.727 they do commit at the same time because they're going to 0:46:15.919 definitely commit at the point

that this record gets written. 0:46:19.401 But the answer to the question are there results visible, 0:46:22.651 the question is are there results visible at the same 0:46:25.669 time, if you stare at this for a little while you realize no 0:46:29.093 because their results become visible at the point at which 0:46:32.401 these A and B receive the commit message from S. 0:46:36 Not at the point that S writes the commit record. 0:46:39.011 So they're going to expose their results at a slightly 0:46:42.335 different point in time. And, in fact, 0:46:44.656 it could be a long time because it may be that A crashed after 0:46:48.483 it went into the tentative commit record and it took it two 0:46:52.121 days to recover. And then it came back up and 0:46:54.881 then it checks with S and says hey, whatever happened to T? 0:46:58.52 So it could be a very long time before, say, A commits, 0:47:01.907 makes its results visible, whereas B may have done 0:47:04.981 immediately after it received the first commit message from S. 0:47:10 You might ask the question is it possible to guaranty that A 0:47:14.417 and B expose their results to the outside world at exactly the 0:47:18.983 same instant. And the answer to this 0:47:21.604 question, it turns out, is no. 0:47:23.775 And this is kind of a fundamental result. 0:47:26.77 The reason for this, let me just give you, 0:47:29.839 there's a simple example that's talked about in the book. 0:47:34.032 It's called the "two generals problem". 0:47:38 0:47:43 And the idea is as follows. Suppose there are two generals 0:47:45.705 and they have their armies. They've flanking somebody who 0:47:48.363 they are attacking and they're on the sides of a valley and 0:47:51.117 they're both going to dive into the valley with the armies at 0:47:53.965 the same time. They're trying to agree what 0:47:55.958 time they're going to do this at. 0:47:58 And they want to make sure they both attack at the same time 0:48:01.035 because, if they don't attack at the same time, 0:48:03.401 they're afraid that one of them will lose. 0:48:05.511 So the only way they have to communicate with each other is 0:48:08.495 by sending these messengers, say, across the valleys. 0:48:11.17 They have some guy who runs across. 0:48:12.919 But, of course, this guy may not make it. 0:48:14.977 He may collapse from exhaustion or he may get shot or whatever. 0:48:18.167 So the first general sends a runner out to the second general 0:48:21.254 that says we'll attack at dawn. And maybe that runner gets 0:48:24.186 through. And then the second general 0:48:25.987 sends a runner back that says yes, we'll attack at dawn. 0:48:30 And maybe that runner gets through, or maybe he doesn't. 0:48:32.677 The second runner doesn't get through, and the first general 0:48:35.55 says man, I don't know if the second general heard about this 0:48:38.471 or not. So he sends another runner that 0:48:40.322 says we'll attack at dawn. And the second general says I 0:48:43 already sent a runner but I guess he didn't get through and 0:48:45.823 he sends another one back. And, ultimately, 0:48:47.868 they both need to agree that they'll both attack at dawn so 0:48:50.692 you need a certain number of runners to successfully get 0:48:53.37 through in order for this to happen. 0:48:56 The issue here, though, is that suppose the 0:48:58.73 general have a fixed number of runners and they want to say 0:49:02.5 what's the maximum number of runners that I could possibly 0:49:06.206 ever need in order to agree on this thing? 0:49:08.871 And, if you think about this for a minute, 0:49:11.537 you will see that there's no finite bound on the number of 0:49:15.242 runners that could possibly be needed, because it's always the 0:49:19.208 case that a huge number of runners could be lost because 0:49:22.783 this is this best-effort network where there's always some 0:49:26.489 probability of something being lost. 0:49:30 There is always this infinitesimal little chance that 0:49:32.68 a million runners in a row wouldn't make it through. 0:49:35.309 So this is essentially the two generals problem. 0:49:37.731 And what it says is it's impossible for these two guys to 0:49:40.618 guaranty that they will achieve consensus about something using 0:49:43.814 a fixed number of messages, using a fixed number of 0:49:46.391 runners. That means, in the case of 0:49:48.144 two-phase commit, what that suggests is that this 0:49:50.618 S or one of these clients may need to continually retransmit, 0:49:53.711 say, an infinite number of times, a very large number of 0:49:56.546 times before its request is actually processed. 0:50:00 Because there's always this sort of small chance of a 0:50:03.459 message being lost by the best-effort network. 0:50:06.453 So that's the two generals problem and it's just sort of a 0:50:10.245 nice result to keep in mind. It's one of those results in 0:50:13.97 computer science that sometimes it turns out to be important. 0:50:17.962 In practice, in this kind of environment it 0:50:20.756 doesn't matter that much. The probabilities of loss are 0:50:24.349 small. And so most of the time this is 0:50:26.81 going to achieve consensus after a limited number of messages. 0:50:32 So that's it for our discussion of fault tolerance and recovery. 0:50:36.186 Next time we're going to start talking about security and 0:50:39.907 protection of information. We will see you on Monday.