MITOCW | ocw-6.033-lec19

0:00:00 We are continuing our discussion of fault-tolerance 0:00:04.078 and atomicity. And sort of teaching these 0:00:07.34 lectures makes me feel, in the beginning, 0:00:10.602 like those TV shows where they always, before a new episode, 0:00:15.414 tell you everything that happened so far in the season. 0:00:19.819 So we will do the same thing. The story so far here is that 0:00:25.457 in order to deal with failures, we came up with this idea of 0:00:30.457 making modules have this property of atomicity, 0:00:34.355 which actually has two aspects to it, one of which is an all or 0:00:39.61 nothing aspect, which we call recoverability, 0:00:43.338 and the other is a way to coordinate multiple concurrent 0:00:48 activities so you get this illusion that they are all 0:00:52.406 separate from each other. And we call that isolation. 0:00:58 And the basic rule here for achieving recoverability was we 0:01:02.082 repeatedly applied this one rule that we called the "Golden Rule 0:01:06.517 of Recoverability" which is to never modify the only copy. 0:01:10.53 And we used that rule to build up this idea of recoverable 0:01:14.542 sector and we used that idea of recoverable sector to come up 0:01:18.766 with two schemes for achieving recoverability, 0:01:21.933 one using version histories where you had a special case of 0:01:26.016 this "Never Modify the Only Copy Rule" which was never modify 0:01:30.24 anything. So, for any given variable, 0:01:34.305 you must create lots and lots of versions never updating 0:01:39.002 anything in place. And then we decided that it is 0:01:43.101 inefficient, so we came up with a different way of achieving 0:01:48.139 recoverability using logging. Then for isolation, 0:01:52.238 we did this last time where we talked about serializability 0:01:57.191 where the goal was to allow the steps of the different actions 0:02:02.4 to run in such a way that the result is as if they ran in some 0:02:07.608 serial order. And we talked about a way of 0:02:12.375 achieving that with cell storage. 0:02:15.09 In particular, we talked about using locks as 0:02:18.824 an abstraction, as a programming primitive to 0:02:22.557 achieve isolation. And, in particular, 0:02:25.696 the key idea that we saw here was that serializability implied 0:02:30.872 that there were no cycles in this data structure called the 0:02:35.793 action graph. And, as long as you could argue 0:02:40.25 that for a given method of locking, as long as you could 0:02:44.244 argue that the resulting action graph had no cycles, 0:02:47.947 you were guaranteed serializability. 0:02:50.488 And, therefore, the scheme provided isolation. 0:02:53.755 And, in particular, a scheme we looked at near the 0:02:57.313 end was two-phase locking. Where the idea is that you 0:03:01.547 never acquire a lock for any item if you have done a release 0:03:05.059 of any other lock in the atomic action so far. 0:03:07.738 And that's the reason why this is called two-phase locking 0:03:11.13 because if you look at the two phases being a lock acquisition 0:03:14.761 phase where the only thing that is happening is locks are being 0:03:18.452 acquired and nothing is being released so the number of locks 0:03:22.023 is strictly increasing with time. 0:03:23.928 And then there is a certain point of the action after which 0:03:27.38 you can only release locks. And you cannot acquire a lock 0:03:32.14 the moment you have released any given lock. 0:03:35.315 And the way we argued this protocol achieved isolation was 0:03:39.523 to consider the action graph resulting from some execution of 0:03:43.953 two-phase locking and argued that if there was a cycle in 0:03:48.087 that resulting action graph then two-phase locking gets violated. 0:03:52.812 And, therefore, two-phase locking provides an 0:03:56.06 action graph which does not have cycles and, therefore, 0:04:00.046 achieves serializability. Two-phase locking is fine and 0:04:04.686 is a really good idea if you're into using locks. 0:04:07.476 It has the property that you do not actually need to know, 0:04:10.79 the action does not need to know at the beginning which data 0:04:14.22 items it is going to access which means that all you need to 0:04:17.651 do is to make sure that you do not release anything until 0:04:20.906 everything has been acquired. But you do not have to know 0:04:24.162 which ones to acquire before the start of the action. 0:04:28 You just have to keep acquiring sort of on demand until 0:04:32.561 typically you get to the commit point. 0:04:35.686 And once you commit you can release the locks. 0:04:39.487 Now, in theory you can release a lock at any given time once 0:04:44.471 you are sure that you are not going to acquire anymore locks, 0:04:49.539 but that theoretical approach only works if you are guaranteed 0:04:54.692 that there will be no aborts happening. 0:04:59 Now, in general, you cannot know beforehand when 0:05:02.1 an action might abort. I mean the system might decide 0:05:05.53 to abort an action for a variety of reasons, and we will see some 0:05:09.751 reasons today. In practice, 0:05:11.467 what ends up happening is that when you abort you have to go 0:05:15.358 back and look through the log and undo the actions, 0:05:18.656 run the undo steps associated with steps that happened in the 0:05:22.614 action. Which means that because the 0:05:26.235 undo goes ahead into cell storage and uninstalls whatever 0:05:30.705 changes were made, it better be the case that when 0:05:34.616 abort starts undoing things, it better be the case that the 0:05:39.246 cell storage items that are being undone have locks owned by 0:05:43.955 this action that is doing the undo. 0:05:46.669 What this means is that if you have a set of statements here 0:05:51.379 that are doing read of x and a write of y and things like that 0:05:56.248 and then you have commit here -- But this is the last point in 0:06:01.966 time at which you are reading and writing. 0:06:04.745 And after that you are doing some computation here not 0:06:08.338 involving any reads or writes. The action might abort anywhere 0:06:12.474 here because the process, I mean this thread might be 0:06:16 terminated and the action will have to abort. 0:06:18.983 What that means is that a release for a data item that is 0:06:22.779 required by abort in order to undo the state changes that have 0:06:26.915 been made, that lock had better not be released before here. 0:06:32 Because if the lock got released here then some other 0:06:35.632 action could have acquired the lock and gone ahead and started 0:06:39.894 working with the changes made by this action. 0:06:42.968 And now it is too late to abort. 0:06:45.133 Someone else has already seen the changes that have been made. 0:06:49.395 In fact, you cannot be guaranteed now that later on you 0:06:53.167 actually can regain the lock and the results would be wrong. 0:06:57.289 So, in fact, the two-phase locking rule 0:06:59.944 really is that you cannot release any lock until all of 0:07:03.716 the locks have been acquired. And, moreover, 0:07:07.928 any locks that are needed in order for abort to successfully 0:07:11.843 run had better not be released until you are sure that the 0:07:15.625 action won't abort anymore. And the only time you can be 0:07:19.274 sure that the action won't abort anymore is once commit has been 0:07:23.454 done.

What that really means is that 0:07:25.777 the release of the locks of all of the items that are required 0:07:29.824 for undoing this action had better happen after the commit 0:07:33.606 point. And, moreover, 0:07:35.801 no item locks should be released until all of the 0:07:38.551 acquires have been done. Now the reason I've said this 0:07:41.586 in two parts is that if you are just reading a data item, 0:07:44.794 you actually don't need to hold onto the lock of that item in 0:07:48.23 order to do the undo because all you did was read x. 0:07:51.151 There is no change that happened to the variable x, 0:07:54.015 which means although you need to acquire the lock of x in 0:07:57.222 order to read it because you don't want to have other people 0:08:00.602 making changes to it while you are reading it, 0:08:03.179 you don't actually need to hold onto the lock of x in order to 0:08:06.673 do the undo because you are not actually writing x during the 0:08:10.109 undo step. So that's the amendment to the 0:08:14.219 two-phase locking rule. Things that you need in order 0:08:18.065 to do undos for should only be released after you are sure that 0:08:22.652 no aborts will be done, which means after the commit 0:08:26.424 point. This way of doing two-phase 0:08:28.865 locking is actually a pretty good scheme. 0:08:31.824 And it turns out that, in many ways, 0:08:34.413 it is the most efficient and most general method. 0:08:39 What that means is that there might be special cases where 0:08:42.495 other ways of other protocols for doing locks of objects 0:08:45.869 perform better under certain special cases compared to 0:08:49.12 two-phase locking, but if you've bought into using 0:08:52.125 locks in order to do concurrency control and you don't know very 0:08:55.989 much about the nature of the actions involved then two-phase 0:08:59.607 locking is quite efficient. I mean there are variants of 0:09:03.761 two-phase locking but, by and large, 0:09:05.964 this idea, it's very hard to do much better than this in a very 0:09:09.865 general sense if you're using locking for doing concurrency 0:09:13.514 control. But there are a set of 0:09:15.402 problems. It's not two-phase locking, 0:09:17.667 as we have described it so far, completely solves the problem 0:09:21.442 of insuring that actions perform well. 0:09:23.77 And a particular problem that happens any time you use locks 0:09:27.483 like here is deadlocks. 0:09:30 0:09:35 And we have actually seen deadlocks before in an earlier 0:09:39.255 chapter when we talked about synchronization of threads, 0:09:43.51 and it is exactly the same problem. 0:09:46.14 And the way you deal with it pretty much is almost the same. 0:09:50.704 What is the problem here? Well, what could happen is that 0:09:55.037 one action does read x and write y and the other action does read 0:09:59.988 y and write x. And now you intersperse the 0:10:03.734 acquires and releases so you do an acquire of lx here and maybe 0:10:07.718 you do an acquire of ly here and here you do an acquire of ly and 0:10:11.831 you do an acquire of lx. And what could happen is that 0:10:15.236 once you get to this stage where this action has come this far 0:10:19.156 and is about to run this and this other action has come up to 0:10:23.012 here, now you are stuck because this action has to wait until 0:10:26.867 that is released and this action has to wait until that is 0:10:30.53 released and neither can make progress. 0:10:34 0:10:39 So there are a few different ways of dealing with it. 0:10:42.545 And the simplest way and the way that turns out to be one 0:10:46.363 that is often used in practice both because it is simple and 0:10:50.386 because once you implement the technique you don't have to do 0:10:54.477 very much else is to just set timers on actions. 0:10:57.681 So it's just to timeout. And if you notice that for a 0:11:01.87 period of time an action has not made any progress then have a 0:11:05.947 timeout that is associated with the action. 0:11:08.753 And if the action itself notices that it hasn't made any 0:11:12.428 progress, perhaps in another thread, then just go ahead and 0:11:16.304 abort this thread. Now, it is perfectly OK to 0:11:19.244 abort. And, in this particular case, 0:11:21.582 aborting either of these actions is enough and the other 0:11:25.258 will make progress and then you are done. 0:11:29 And then the action that got aborted can retry. 0:11:32.557 So the first solution is to just use a timer. 0:11:35.96 And there is a school of thought that believes that in 0:11:40.058 practice deadlocks should not be very common. 0:11:43.461 And the reason is that deadlocks occur if there is, 0:11:47.327 you know, there has to be a contention for resources and 0:11:51.581 there has to be contention for multiple threads for the same 0:11:56.144 resources. And it has to be more than one 0:11:59.923 resource, because if you just have one resource you cannot 0:12:03.576 really get a deadlock which means that you are sort of 0:12:06.974 running multiple actions that are contending for a number of 0:12:10.756 different shared objects. And what that suggests is if 0:12:14.153 there is a high degree of concurrency like that and shared 0:12:17.807 contention then it may be hard for you to get high performance. 0:12:21.782 A lot of people think that the right way to be designing 0:12:25.307 applications is to try hard to insure that the degree of 0:12:28.833 sharing between objects is actually quite small. 0:12:33 For example, rather than set up a lock on an 0:12:35.512 entire big database table, you might set up locks at final 0:12:38.844 granularities. And if you set up locks at 0:12:41.181 final granularities the chances of multiple actions wanting to 0:12:44.746 gain access to the same exact fine-grained entry in a table 0:12:48.136 might be small. And in that situation, 0:12:50.298 given that the chances of a deadlock occurring are rare, 0:12:53.512 timing out every once in a while and aborting an action is 0:12:56.844 not going to be catastrophic. It's OK. 0:13:00 It is a rare event. So rather than spend a whole 0:13:02.33 lot of complexity dealing with that rare event, 0:13:04.611 just go ahead and let something abort. 0:13:06.446 Let an action that hasn't made any progress abort. 0:13:08.876 Moreover, these timers are necessary anyway because an 0:13:11.504 action might end up getting stuck in an infinite loop or it 0:13:14.38 might end up getting stuck in a situation where it is not really 0:13:17.504 waiting for a lock there is just a bug in it. 0:13:19.685 There is a problem with it, it is not really making any 0:13:22.363 progress and maybe it is consuming resources and no one 0:13:25.041 else can make progress. So the system anyway needs a 0:13:27.57 way to abort those actions. And it needs a timeout 0:13:31.369 mechanism anyway. So why not just use that same 0:13:34.232 mechanism to deal with deadlocks as well. 0:13:36.721 Probably somewhat a minority, but some other people believe 0:13:40.331 that deadlocks might happen. And, when they do happen, 0:13:43.63 perhaps because the granularity of locking in your system is not 0:13:47.551 fine-grained then you do not want to get stuck. 0:13:50.414 And you want to optimize, at least you want to do 0:13:53.402 reasonably well rather than waiting for some long timeout 0:13:56.887 period before aborting an action. 0:14:00 And people who believe that build a data structured called 0:14:04.584 the "Waits-For Graph". And the best way to understand 0:14:08.766 this

build a data structure called the "Waits-For Graph". And the best way to understand this is imagine you have a database system that supports 0:14:13.19 isolation and any time you want to acquire a lock you send a 0:14:17.935 message to this entity in the database system called lock 0:14:22.439 manager asking to acquire a lock. 0:14:25.013 And any time you release it you do the same thing. 0:14:30 What that lock manager can do, for each lock it can keep track 0:14:33.944 of which actions running concurrently has acquired that 0:14:37.436 lock and which action is waiting for a lock. 0:14:40.217 And what you can do now is build up a graph of actions and 0:14:43.903 locks and look to see whether there is some kind of cycle 0:14:47.524 where you have action A waiting for lock B and lock B is being 0:14:51.468 held by action C and action C is waiting for lock D and lock D is 0:14:55.607 being held by action A. When you have a cycle in this 0:14:59.764 graph then you know that you have a deadlock and none of 0:15:03.11 those actions can make progress so go ahead and kill one. 0:15:06.517 And you can be sophisticated about deciding which one to 0:15:09.863 kill. You might kill the one, 0:15:11.566 for example, that has been waiting the 0:15:13.817 shortest amount of time because the others have been waiting 0:15:17.406 longer so they might make progress, or you might have 0:15:20.57 other policies for deciding which ones to kill. 0:15:23.368 In practice, both these systems are used 0:15:25.741 sometimes by the same system combining these ideas. 0:15:30 For example, if you look at like an Oracle 0:15:32.464 database system, it uses primarily timers. 0:15:34.929 At least from what I could tell, it does not seem to have 0:15:38.296 any mechanisms for really doing this check of a Waits-For graph. 0:15:42.084 It just uses timers. And one of the oldest 0:15:44.549 transaction processing systems was a system called CICS from 0:15:48.096 IBM which also basically used timers, but there are other 0:15:51.462 systems. For instance, 0:15:52.725 IBM has this system called DB2 and Microsoft Sequence server 0:15:56.272 that both use this Waits-For data structure. 0:16:00 And, in fact, Microsoft's system seems to 0:16:02.542 have a hundred thousand different knobs for deciding how 0:16:06.038 to turn off deadlocks, including the ability to set 0:16:09.216 various priorities on different actions that might be running. 0:16:13.093 And it is not actually apparent that those knobs actually are 0:16:16.906 usual for anything or how you set them but that they have a 0:16:20.593 lot of things that you could set. 0:16:22.627 Sounds familiar. Now, you can combine these two. 0:16:25.614 And I think certain products combine these two ideas. 0:16:30 One decision you have to make is to decide when to check this 0:16:33.292 Waits-For graph. And an aggressive way of doing 0:16:35.815 it is the moment anybody does an acquire or anybody does a 0:16:38.943 release, in particular an acquire, you update your lock 0:16:41.906 manager's data structure and immediately look to see if you 0:16:45.088 have a cycle. Of course that takes time and 0:16:47.392 effort. You might decide not to both 0:16:49.313 but rather periodically look for cycles in this Waits-For graph 0:16:52.715 when a timer fires, so every three seconds go ahead 0:16:55.458 and look for cycles. So you might combine these 0:16:57.982 ideas in a bunch of different ways. 0:17:01 Now, if you recall from several lectures ago, 0:17:03.593 another way of dealing with deadlock is to order all of the 0:17:07.011 locks that an action might be able to acquire in a particular 0:17:10.548 order and insure that all of the actions acquire the locks in 0:17:14.084 exactly the same order. And that will insure there are 0:17:17.208 no cycles because you have to go in the same order, 0:17:20.155 but that idea requires you to know beforehand which data items 0:17:23.75 you wish to gain access to. And that's often not possible 0:17:27.051 in many systems in which you care about isolations. 0:17:31 So that's usually not adopted at least in any database system. 0:17:36.218 OK, so we talked about deadlocks. 0:17:38.955 We talked about when you can release a lock that you acquire 0:17:44.002 in order to abort because you cannot release it typically, 0:17:48.879 in reality, until the commit point is done. 0:17:52.471 The last issue we need to talk about is an interaction between 0:17:57.69 logs and locks. And this interaction has to do 0:18:02 with, so we already saw what happens when you abort. 0:18:05.4 When you abort you need to undo so you better make sure that to 0:18:09.533 do the undo you have the locks for those cell items. 0:18:12.933 You don't have to abort but suppose you crash. 0:18:15.933 Suppose the system crashes and recovers. 0:18:18.533 At that point, when it recovers, 0:18:20.599 it is going to run a recovery procedure which has some 0:18:24.133 combination of redoing the winners and undoing the losers. 0:18:29 Now, when it's undoing things and redoing things it needs 0:18:32.645 access to items in the cell store. 0:18:34.793 And we've already seen when the system is normally running, 0:18:38.569 in order to change items in your cell store you need to gain 0:18:42.41 access to locks. The question now is during 0:18:45.144 crash recovery when the system is running this redo undo thing, 0:18:49.18 where do you get these locks from and do you need to gain 0:18:52.826 access to the locks? Now, in general, 0:18:55.169 the answer to the question might be that you need to be 0:18:58.685 very careful and perhaps need access to the locks when you're 0:19:02.591 running recovery. But there is one simplification 0:19:07.201 that systems typically make that eliminates that requirement. 0:19:11.463 And that simplification is that during crash recovery you don't 0:19:15.866 really allow new actions to run on your system. 0:19:19.133 So when a system crashes and it is recovering, 0:19:22.329 do not allow new actions to run until recovery is complete. 0:19:26.448 And only then do you start new actions. 0:19:30 What this means is now we just have to worry about insuring 0:19:34.285 isolation clearly during recovery without having new 0:19:38.054 actions coming in and muddling things up. 0:19:41.009 The question really to think about is whether before the 0:19:45.073 crash, because the log is the only thing you have in order to 0:19:49.507 do recover, whether in the log you actually need to keep track 0:19:54.014 of which locks were being held when the system was running just 0:19:58.596 fine. And if it turns out that the 0:20:01.722 log has to encode in it the locks that were being held, 0:20:05.043 it could be quite complicated and a little bit messy. 0:20:08.242 But if you think about it, the nice thing is that we don't 0:20:11.748 actually have to encode the locks at all, 0:20:14.208 store the locks at all. The locks can be completely 0:20:17.283 involved in the storage. And that is because when you 0:20:20.482 start off, when you have a log which has various redo items and 0:20:24.295 undo items, in any element of the log, let's say an item x has 0:20:28.047 been updated in that log entry. Then you know for sure that at 0:20:33.138 the time this log entry was written, the action that was 0:20:37.059 making this update did hold onto this lock and that this change 0:20:41.479 being made here that got written to the log was, 0:20:44.829 in fact, isolated assuming the locking protocol was correct, 0:20:49.035 was, in fact, isolated from everything else 0:20:52.029 concurrently that was

going on. And so, although the locks are 0:20:56.378 not explicit, the log encodes in it the 0:20:59.087 actual serial order, some serial order of execution 0:21:02.651 that did provide isolation before the crash. 0:21:07 Therefore, if you just blindly go back through the log and make 0:21:10.455 those changes in sequential order then you are assured that 0:21:13.687 the changes you make are, in fact, going to be isolated 0:21:16.696 from one another. So you do not actually have to 0:21:19.315 worry about storing the locks before the crash into the log, 0:21:22.603 and that makes life quite simple. 0:21:25 0:21:37 That wraps up the discussion of atomicity and, 0:21:40.795 in particular, isolations. 0:21:42.903 For the rest of today and next time we are going to be talking 0:21:48.048 about some uses of atomicity. 0:21:51 0:21:58 And the plan is the following. The plan is the first 0:22:02.231 application of atomicity which actually is the umbrella for a 0:22:07.21 number of things we are going to be looking at is a transaction. 0:22:12.438 And a transaction is defined as an atomic action that has a few 0:22:17.582 other properties that it holds. And the first property is 0:22:22.229 consistency and the second property is durability. 0:22:26.295 And the second thing we are going to look at, 0:22:29.946 next lecture actually, is atomicity when you have a 0:22:34.095 distributed system. It is using atomicity on one 0:22:39.4 computer to build out a system that provides atomicity in a 0:22:44.371 distributed system. So we will talk about 0:22:47.799 consistency the rest of today and the recitation for tomorrow 0:22:52.942 looks at a paper for reconciling replicas, which is a particular 0:22:58.342 aspect of consistency. And then next lecture next week 0:23:03.649 we will talk about multi-site atomicity. 0:23:06.982 And the recitation next week we will talk about durability. 0:23:11.94 And once we do all of that, that kind of wraps up this 0:23:16.47 fault-tolerance part of 6.033. Let me first talk a little bit 0:23:21.598 about transactions. Transaction is an atomic action 0:23:25.871 that has two other properties associated with it. 0:23:31 And people in the literature often, in collegial terms, 0:23:34.583 refer to transactions as having a property called the ACID 0:23:38.366 property where ACID stands for atomicity, consistency, 0:23:41.883 isolation and durability. And you will see this term a 0:23:45.4 great deal in the literature and people will use this all the 0:23:49.382 time. And, for various reasons, 0:23:51.372 the way we have done things in this class, some of these terms 0:23:55.421 are used in slightly different ways from the ACID term. 0:24:00 When most people, at least in distributed systems 0:24:03.428 and database systems, use the word atomicity, 0:24:06.571 what they mean is what we meant by recoverability. 0:24:10.071 So it is all or nothing. When they use the letter I here 0:24:14 for isolation, they mean exactly the same 0:24:16.857 thing here that we did. And consistency and durability 0:24:20.642 unfortunately are going to mean the exact same thing. 0:24:24.357 But really the point to notice is that these two properties, 0:24:28.571 atomicity and isolation are things that are independent of 0:24:32.642 an application. They just are properties of 0:24:36.47 atomic actions that an atomic action can be recoverable and 0:24:39.629 can be isolated. And you do not have to worry 0:24:42.026 about what the application is. It could be an application in 0:24:45.239 database systems. It could be something in a 0:24:47.581 processor where you are trying to provide recoverability or 0:24:50.74 isolation for instructions. These are properties that are, 0:24:53.845 in some sense, somewhat more fundamental and 0:24:56.187 lower layer properties than these other two properties. 0:25:00 What consistency means is the property of an atomic action 0:25:05.659 that is some application-specific invariant. 0:25:09.929 Consistency of a transaction says that if you have a 0:25:14.992 transaction that commits then some set of consistency 0:25:20.156 invariants must hold. I will describe some examples 0:25:25.12 of what this means. Consistent just says that there 0:25:30.112 is some application-specific invariants that must hold. 0:25:33.791 And durability says that if a transaction commits then the 0:25:37.675 state changes that it has made, that the data items that it has 0:25:41.899 changed has to last for some period of time. 0:25:44.829 And the period of time that they have to last for is defined 0:25:48.849 by the application. And there are many examples. 0:25:52.052 A simple example of durability might be that the changes made 0:25:56.14 by an atomic action just have to last until the entire thread 0:26:00.228 finishes. And, at the other extreme, 0:26:03.857 you could get into semantics of durability which say that the 0:26:08.142 changes made by an atomic action have to last for three years or 0:26:12.642 for five years or for forever which is a really hard thing to 0:26:16.928 solve. But you might define semantics 0:26:19.5 that relates to the permanence of data. 0:26:22.214 For how long do you want the changes that you made to last 0:26:26.285 and be visible to other atomic actions? 0:26:30 0:26:42 There are two cases for consistency that we need to talk 0:26:46.7 about. The first one is consistency in 0:26:49.863 a centralized system. 0:26:52 0:26:57 An example of this, and the most common example of 0:27:01.443 this is in database systems that support transaction where you 0:27:06.975 might have rules that are also called integrity rules for 0:27:12.053 deciding whether you are allowing a transaction to commit 0:27:17.132 or not. Let me give you a couple of 0:27:20.215 examples of this. Let's say that you have a type 0:27:24.477 of database system as a relational database system where 0:27:29.465 all of the data is stored in tables. 0:27:34 For example, you might have a table storing 0:27:38.684 a student ID, a student name and let's say 0:27:43.256 the department that the student belongs to. 0:27:47.94 And let's say the departments have IDs. 0:27:52.178 And you might have another table in your system that stores 0:27:58.646 a department ID and a department name. 0:28:04 Now, you might have a transaction that makes updates 0:28:07.979 to entries in this table, you know, one or more rows in 0:28:12.193 this table could actually make updates to just specific cells 0:28:16.874 of this table. It could add a new student ID, 0:28:20.308 add a name and add some department ID. 0:28:23.195 Now, the kind of constraint we are worried about, 0:28:26.94 the kind of invariants we are worried about are things where 0:28:31.544 the person who has designed this database might say that you are 0:28:36.459 not allowed to add a department ID that is nonexistent. 0:28:42 And what that means is that there are these two tables. 0:28:44.986 And you should not allow any transaction to write the 0:28:47.862 department ID which is not already in this table. 0:28:50.517 So if 43 might be in this table and 25 might be on this table, 0:28:53.891 but a number that is not in this table should not be added 0:28:57.044 here. And so the transaction 0:28:58.537 processing system might decide, will, in fact, 0:29:01.026 not allow this transaction to commit if it is writing a value 0:29:04.345 that is not in this other table. And, for those familiar with 0:29:08.968 databases, relation databases, there are these two tables 0:29:12.906 called T1 and T2. This might be a primary

databases, relation databases, there are these two tables called T1 and T2. This might be a primary key. 0:29:16.07 Department ID might be a primary key of T2 defined as 0:29:19.726 what is called a foreign key in T1, which means that you are not 0:29:24.156 actually allowed to add something to a foreign key if it 0:29:28.023 is not already in the other table where that same column is 0:29:32.101 a primary key. So there are rules like this in 0:29:35.778 most relational database systems and there are a variety of rules 0:29:39.45 like this that all have to do with maintaining the integrity 0:29:42.836 of the data that you add here. Now, this has nothing to do 0:29:46.106 with isolation. It has to do with atomicity 0:29:48.516 because these rules are typically checked at the commit 0:29:51.614 point, because until then anything could happen. 0:29:54.311 So, right before you commit, there are these invariants on 0:29:57.581 the data that are application-specific that you 0:30:00.221 need to check. But it has nothing to do with 0:30:03.551 locks. It has nothing to do with 0:30:05.209 anything. It sort of presumes atomicity, 0:30:07.296 and after that it checks these application-specific rules. 0:30:10.345 And you can get quite sophisticated. 0:30:12.218 Some of these things about primary keys and secondary keys 0:30:15.267 are things that are checked by most transaction processing 0:30:18.316 systems, but you could get quite sophisticated about these rules. 0:30:21.74 For example, you could have rules. 0:30:23.506 Let's say you have a database storing employees and their 0:30:26.502 salaries. You could have rules that say 0:30:29.596 any time an employee gets a raise then everybody else in the 0:30:32.845 same peer group also gets some kind of raise. 0:30:35.268 And so you wouldn't allow any transaction to commit that did 0:30:38.516 not insure that invariant to hold. 0:30:40.333 And checking these things could be quite difficult, 0:30:43.087 and most systems do not actually do a really good job of 0:30:46.115 checking these things. The sets of rules they allow 0:30:48.868 you to write is quite limited because checking it is quite 0:30:52.007 hard, because when you are trying to commit a transaction 0:30:55.09 now you might have to check a large number of rules. 0:30:59 And some of them could be both time-consuming and complicated. 0:31:03.492 But the main point here is that these rules are 0:31:06.881 application-specific. And that is what defines 0:31:10.195 consistency of the data that you have. 0:31:12.92 The more interesting case for consistency and the thing that 0:31:17.266 is going to occupy us for the rest of today and tomorrow is 0:31:21.538 consistency in distributed systems. 0:31:25 0:31:30 In particular, when the same data gets 0:31:32.452 distributed, typically for fault-tolerance and for 0:31:35.7 availability, to insure that the data is 0:31:38.285 available at different locations, you end up with 0:31:41.467 consistency problems. And we have already seen a few 0:31:44.848 examples of this. One example of this is in the 0:31:47.897 "Domain Name System" which maintains mapping between domain 0:31:51.742 names and IP addresses. And, if you remember, 0:31:54.659 in order to achieve availability and good 0:31:57.31 performance, these mappings between DNS names and IP 0:32:00.691 addresses where cached essentially on demand. 0:32:05 Whenever a name server on the Internet made an access to that 0:32:09.7 name it address cached the mapping results. 0:32:12.99 And so now you have to worry about whether the data that is 0:32:17.534 cached somewhere out on the Internet is, in fact, 0:32:21.294 the correct data where correct is defined as the data that is 0:32:25.995 being maintained by the primary name server. 0:32:29.364 And if you think about DNS did, it actually used a mechanism of 0:32:34.221 expiration times to keep this cache consistent. 0:32:39 And what that means is that the only time you are guaranteed 0:32:43.433 that the data in a cache is, in fact, the data that is 0:32:47.416 stored at the primary name server for that name is when 0:32:51.473 this expiration time finishes. And the first access after the 0:32:55.982 expiration time requires the name server to go to the 0:32:59.89 original primary name server and do a look up of the name. 0:33:05 So the rest of the time you cannot actually be guaranteed 0:33:10.195 that the data is consistent. And, in other words, 0:33:14.649 you are not getting what is considered strong consistency. 0:33:19.938 What is strong consistency? 0:33:23 0:33:28 One way to define the semantics of what it means for data to be 0:33:31.932 consistent in a distributed system is it is sort of a 0:33:35.231 natural definition which is to see that any time you do a read 0:33:39.1 anywhere, any node does a read of some data, 0:33:41.828 read returns the result of the late write. 0:33:45 0:33:55 That is one notion of consistency. 0:33:57.479 And a system provides strong consistency if you can insure 0:34:01.763 that every read returns the result of the last write that 0:34:05.971 was done on the data. And this is really hard to 0:34:09.611 provide because what it typically means is that the data 0:34:13.019 is widely replicated or cached. Any time anybody changes the 0:34:16.676 data you have to make sure that all of the copies get that 0:34:20.208 change. And, even if you work really 0:34:22.377 hard to invalidate all the entries and make changes to it, 0:34:25.909 there are these small windows of vulnerability where -- 0:34:30 In fact, in DNS, for example, 0:34:31.473 even the first access that you make the server after the 0:34:34.368 expiration time may not guaranty that when the response returns, 0:34:37.684 the response is, in fact, the newest response 0:34:40 because the primary name server could send a response. 0:34:42.789 And, while it is coming back to the person who made the query, 0:34:46 the data could get changed at the primary name server so it is 0:34:49.21 really hard to guaranty this, at all points in time, 0:34:51.894 in a distributed system. And it gets much harder when 0:34:56.134 there are failures making certain copies unavailable or 0:35:00.252 making access to a primary in the DNS case unavailable. 0:35:04.369 In practice, in most systems, 0:35:06.504 the kind of consistency that people try to get is eventual 0:35:10.85 consistency or they try to approximate strong consistency 0:35:15.12 in some other way. And eventually consistency just 0:35:18.856 -- It is a little bit of a loser 0:35:21.905 notion, but what it says is that there might be periods of time 0:35:25.98 where things are consistent or that the system is doing work in 0:35:30.055 the background to make sure that all of the copies of a given 0:35:33.998 data item are, in fact, the same and are the 0:35:36.824 result of the last write to that data. 0:35:39.255 Again, the notion of eventual consistency depends a lot on the 0:35:43.264 application. So, really, to specify this 0:35:45.827 precisely you have to look at in the context of the application. 0:35:49.968 Different applications you have different notions of consistency 0:35:54.108 and eventual consistency. So we looked at DNS as an 0:35:59.131 example. Another example to look at is 0:36:02.286 something you might be familiar with which is "Web caches". 0:36:07.232 Web caches, for example, your browser has a cache in it. 0:36:11.922 And there might be Web caches located elsewhere in the network 0:36:17.124 that capture your requests. And people use Web caches to 0:36:21.813 save latency or to prevent slamming a Web server that

requests. And people use Web caches to either have lower latency or to prevent slamming a Web server that might 0:36:26.759 otherwise get overloaded. The semantics here are usually 0:36:32.028 that you do not just return stale data. 0:36:34.598 If the data has changed on the Web server, it might be that you 0:36:38.79 actually want to return good data to the client. 0:36:41.969 The way this is normally done is for the client or for any 0:36:45.823 cache to first check with the Web server to see if the data 0:36:49.745 has been changed since the last cached version. 0:36:52.856 Let's say that the cache went to the Web server at 9:00 in the 0:36:56.981 morning and had to go there because it did not have the data 0:37:00.971 in the cache. And it got some data back. 0:37:04.471 The data has a timestamp on it. Then the next time somebody 0:37:07.754 makes a request to the cache, the cache does not just return 0:37:11.094 the data immediately. What the cache usually does is 0:37:13.981 to go to the Web server and ask the Web server if the data has 0:37:17.433 changed since 9:00 in the morning. 0:37:19.301 If the data has changed since 9:00 in the morning you might 0:37:22.584 retrieve the data. You would retrieve the data for 0:37:25.358 the server. If not then go ahead and return 0:37:27.735 the data to the client. This is also called 0:37:31.465 "If-Modified-Since" because what you are saying is the cache is 0:37:36.246 telling the server send me the data if it has been modified 0:37:40.719 since the last time I know the version of the data that I have. 0:37:45.501 And a convenient way to represent that is as a 0:37:48.971 timestamp. It's just a version of the 0:37:51.748 data. So you can see that this 0:37:53.984 actually provides a more stronger consistency semantics 0:37:58.149 than DNS. Because in DNS the data could 0:38:02.453 have changed and your cache just has outdated data. 0:38:06.684 But for the application that DNS is used for it is perfectly 0:38:11.676 OK for that to be the case. Now, in general, 0:38:15.315 in distributed systems there is a tradeoff between the 0:38:19.8 consistency of data at the different replicas and 0:38:23.861 availability. Availability just means that 0:38:27.33 clients wanting data should get some copy of the data. 0:38:33 Now, if the system is strongly consistent then the copy of data 0:38:37.203 that you get is, in fact, the result of the last 0:38:40.389 write. But the tradeoff occurs between 0:38:42.898 availability and consistency because in many distributed 0:38:46.627 systems your networks are not reliable or nodes themselves are 0:38:50.762 not reliable and they might fail. 0:38:52.932 So in the presence of failures, say network partitions or 0:38:56.728 failures of nodes, it turns out to be really hard 0:38:59.983 to guaranty both high availability and strong 0:39:02.966 consistency. As sort of a trivial existent 0:39:06.75 example of this, if you have three copies of the 0:39:09.588 data and you were not very careful about figuring out your 0:39:13.029 write protocol. Let's say that your write 0:39:15.444 protocol was to sort of write to one version and then your read 0:39:19.187 protocol was to just read from some other version and for some 0:39:22.869 process in the background to transfer the replica from the 0:39:26.311 first version that the client wrote to, to all of the other 0:39:29.812 copies, then there would be periods of time of the network 0:39:33.253 where partitioned you could end up in a situation where the 0:39:36.755 version that a given client is reading is not actually the last 0:39:40.498 version of the data that was written. 0:39:44 In fact, if you started thinking about DP2, 0:39:47.38 Design Project 2, really, one part of it gets at 0:39:51.164 how you manage replicated data. For example, 0:39:54.625 when the utility that does the archiving publishes data, 0:39:59.052 one approach it might take is to publish the data that it 0:40:03.56 wants to archive to all of the copies, to all of the replica 0:40:08.309 machines. And the read protocol might be 0:40:12.09 to read from one of them. Now, if you insure that the 0:40:15.714 write protocol finishes and succeeds only when all of the 0:40:19.616 replica machines are updated then you can try to get at a 0:40:23.519 decent version of consistency. But you need to be able to do 0:40:27.63 that when failures occur. The network might fail or nodes 0:40:32.047 might fail, and you need to figure out how to do that. 0:40:35.547 But you might decide that writing to end copies and 0:40:38.849 reading from one copy is difficult or has high overhead 0:40:42.415 so you might think about ways of writing to certain subsets, 0:40:46.311 writing to a subset of the machines and reading from a 0:40:49.811 subset of the machines to try to see whether you could come up 0:40:53.839 with ways to get a consistent version of the data. 0:40:58 Or you might decide that the right way to solve the problem 0:41:01.186 is not to try to achieve really strong consistency in all 0:41:04.263 situations but to relax the kind of consistency you want and 0:41:07.505 maybe a different version of semantics. 0:41:09.593 As long as you are precise about the semantics that your 0:41:12.615 system provides, it might be a different 0:41:14.758 solution or reasonable solution to the problem. 0:41:18 0:41:28 So one interesting way in which people achieve reasonable strong 0:41:33.055 consistency in tightly coupled distributed systems, 0:41:37.067 and distributed systems that are not across the Internet 0:41:41.481 where a network could arbitrarily fail, 0:41:44.53 but in more tightly coupled systems is in a multiprocessor. 0:41:49.185 If you have a computer that has many processors -- 0:41:54 0:42:04 And the abstraction here for this multiprocessor is that of 0:42:08.303 shared memory. You actually have memory 0:42:11.123 sitting outside here, and these processors are 0:42:14.462 reading and writing data to this memory. 0:42:17.356 The latency to get to memory and back is high. 0:42:20.696 So, as you know, processors have caches on them. 0:42:25 0:42:35 As long as the memory locations that are being written and read 0:42:38.945 are not shared between them these caches could function just 0:42:42.699 fine. And when there is an 0:42:44.29 instruction running on one of these processors that wants to 0:42:48.045 access some memory location, you could just read and write 0:42:51.672 from the cache so things would just work out. 0:42:54.472 The problem arises when there is a memory location being read 0:42:58.29 here that actually was previously written by this 0:43:01.345 processor. And, if you read it here, 0:43:04.666 then you might get an old version of the data. 0:43:07.666 And if you think of just memory as the basic abstraction, 0:43:11.4 virtual memory then this is bad semantics because your programs 0:43:15.533 wouldn't function the same way as they did when you just had 0:43:19.466 one processor or when you didn't have the caches at all and you 0:43:23.599 just went directly to memory from multiple processors. 0:43:27.133 The question is how do you know whether the data in a cache is 0:43:31.199 good or bad? Now, like in the Web caches 0:43:34.804 case, checking on every access whether the data has changed is 0:43:38.881 not going to be useful here because the amount of work it 0:43:42.623 takes to check something is about the same as the amount of 0:43:46.5 work it takes to read or write something because you have taken 0:43:50.643 the latency hit for that. So that approach is not going 0:43:54.252 to work. The solution that is followed 0:43:56.725 in many systems is to

use two ideas. 0:44:00 The first idea is that of a "Write-Thru Cache". 0:44:03.788 What a write-thru cache says is if there is a write that happens 0:44:08.976 here, or store instruction, the cache gets updated. 0:44:13.094 But, in addition to the cache getting updated, 0:44:16.8 the data also gets written through on the bus to the memory 0:44:21.576 location here. So that is the first idea, 0:44:24.87 to use a write-thru cache. 0:44:28 0:44:38 The second is because this is a bus all of these nodes can 0:44:41.007 actually snoop on this bus and see what activity there is on 0:44:44.119 the bus because it is a shared bus. 0:44:45.913 It is a very special kind of network, as I said. 0:44:48.393 You cannot apply this idea in general. 0:44:50.345 It is a very special kind of network where because it is a 0:44:53.352 bus and nothing fails, or the assumption is nothing 0:44:55.99 fails, everybody can check to see what is going on on the bus. 0:45:00 And any time there is any activity on the bus that 0:45:03.259 corresponds to something that is stored in any node's cache you 0:45:07.383 can do two things. You can actually invalidate 0:45:10.376 that cache entry but you can actually also see what the 0:45:13.968 update is and go ahead and look at the change that was being 0:45:17.893 done and update your cache. And this idea is sometimes 0:45:21.419 called a "Snoopy Cache" because you have these caches that are 0:45:25.476 snooping on activity that is occurring in your system. 0:45:30 And this is one way in which you can achieve something that 0:45:33.501 resembles strong consistency. But it actually turns out, 0:45:36.82 if you think hard about it, a precise version of strong 0:45:40.08 consistency is really hard to achieve. 0:45:42.313 In fact, it is very, very hard to even define what 0:45:45.271 it means for any read to see the result of the last write because 0:45:49.134 when you have multiple people reading and writing things, 0:45:52.515 when you get down to the instruction level, 0:45:55.05 it turns out to be really hard to even define the right 0:45:58.309 semantics. A lot of people are working on 0:46:01.615 this kind of thing. But this is a little bit of a 0:46:04.199 special case because this kind of solution applies only in a 0:46:07.376 very tightly coupled system where you do not really have 0:46:10.338 failures and everybody can listen to everything else. 0:46:13.138 But it is interesting to note that there are cases when you 0:46:16.261 can achieve it and that is why this is interesting. 0:46:18.953 It is practically useful. 0:46:21 0:46:26 So the main thing about Design Project 2 that relates to the 0:47:03.072 consistency discussion is for you to try to, 0:47:30.092 I mean at least one part of it, in case it was not clear from 0:48:07.793 the description of the project is for you to think about what 0:48:45.494 kind of consistency you want and come up with ways to manage 0:49:22.567 these different replicas. We are going to stop here. 0:49:54.613 Next week we will talk about multi-site atomicity. 0:50:25.402 Tomorrow's recitation is on a system called Unison which also 0:51:03.104 looks at consistency when you have mobile computers that are 0:51:40.177 trying to synchronize data with servers.