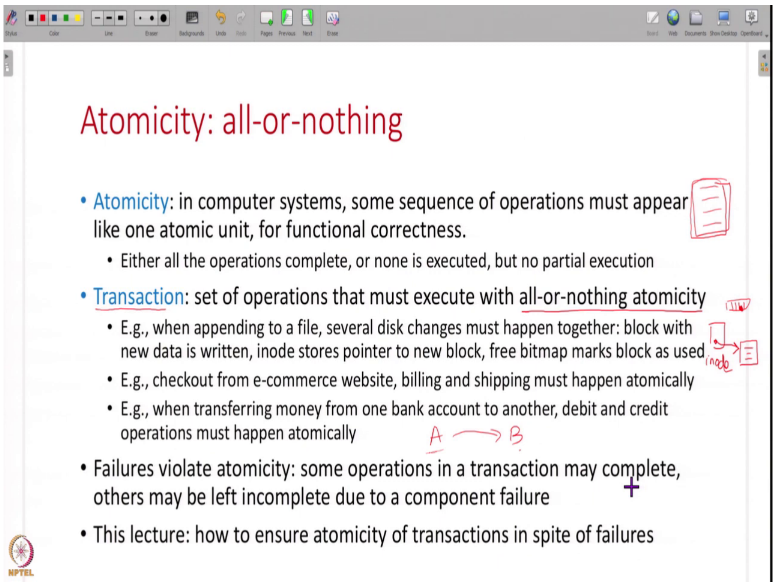
Design and Engineering of Computer Systems
Professor Mythili Vutukuru,
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 53
Atomicity

Hello everyone, welcome to the 38th lecture in the course Design and Engineering of Computer Systems. In this lecture, we are going to study more about the topic of Atomicity. So, what is Atomicity? We have discussed this couple of times before in the course.

(Refer Slide Time: 00:32)



Atomicity: all-or-nothing

- Atomicity: in computer systems, some sequence of operations must appear like one atomic unit, for functional correctness.
  - Either all the operations complete, or none is executed, but no partial execution
- Transaction: set of operations that must execute with all-or-nothing atomicity
  - E.g., when appending to a file, several disk changes must happen together: block with new data is written, inode stores pointer to new block, free bitmap marks block as used
  - E.g., checkout from e-commerce website, billing and shipping must happen atomically
  - E.g., when transferring money from one bank account to another, debit and credit operations must happen atomically
- Failures violate atomicity: some operations in a transaction may complete, others may be left incomplete due to a component failure
- This lecture: how to ensure atomicity of transactions in spite of failures

So, atomicity is a property where if there is a sequence of operations, then sometimes in computer systems, you require that all of these operations, they do not appear as different different steps, but they appear as one atomic unit, only then the system will work correctly. That is you either complete all these operations or you do not do any operation at all, but for this atomic unit, you cannot execute half of this atomic unit and leave the other half, no partial execution should happen.

So, such set of operations are called transactions, and all the operations in a transaction must execute or none of them should execute, if you execute a transaction partially, then you will have all sorts of errors and bugs in your system, your system will not have correctness.

So, that is atomicity, the ability to execute all the operations within a transaction or if there is a failure do not execute anything at all. So, that is called all or nothing atomicity. So, what are some examples? We have seen this before, when we were studying file systems, for a file system, system call, if you are implementing, there could be many changes being made too many disk blocks, and all of these should happen together.

But you cannot do half of them and leave the other half safe, some failure happens in the middle, you cannot just do half and half, for example, the sequence of operations are if you are writing some data to a file, so there is a new block with the file data, then there is your inode, in which you are storing a pointer to this new block, you are storing the block number in your inode, and there is a bitmap where you are marking this block as occupied, all of these changes should happen together. If you do some and you do not do some, you will run into issues.

For example, if you add this block number to the inode, but you do not fill in this data in the block, then what will happen, when you are reading this file, you read this block; you will find garbage in the file. If you allocate this block to this file, but you do not mark the bitmap, then what will happen, the bitmap will think that this block is free allocated to some other file and your data will get overwritten, all sorts of bad things can happen. So, if a failure happens, you skip all these changes do not do any of these changes at all. But if you do them do all of them together, that is atomicity.

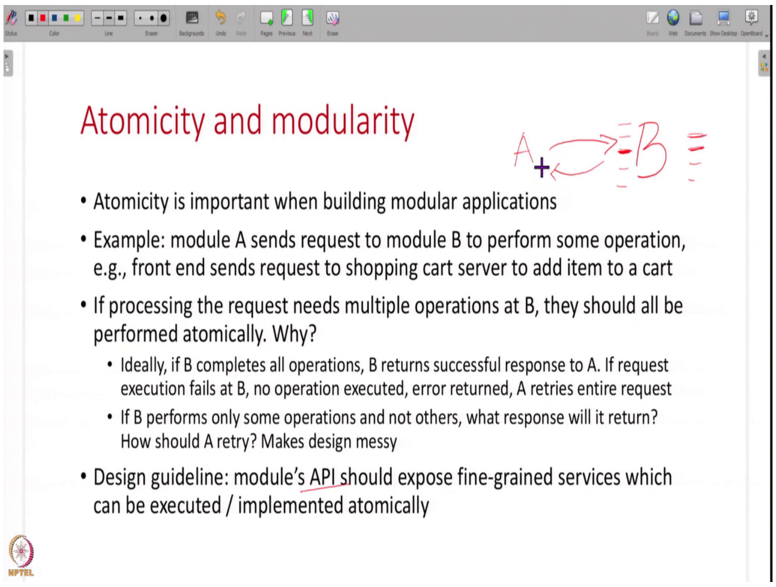
There are other examples also that we have seen, if you are doing a checkout from an e-commerce website, the billing and shipping should happen atomically, you cannot say that I have charged your credit card, but there was a failure, I did not ship the product that is not acceptable. Similarly, a common example is transferring money, if you are transferring money from one account to the other you deduct the money here, you add the money here; you cannot deduct the money here and forget to add the money here that is not acceptable.

So, why will atomicity be violated, of course people are not out there to violate atomicity on purpose, but due to bugs or most common reason is failures, you are in the middle of a sequence of steps and a failure has happened and therefore, some were executed and some steps could not be executed, and whatever you executed were maybe lost some changes were lost due to memory contents being flushed out things like that due to failures is when atomicity is usually violated in computer systems.

And in such cases, when failures happen, of course, we cannot avoid failures, but when any failures or faults happen what we want to do is either you complete the operation or you do not do anything at all. That is also okay, you say a failure happened this account transfer did not take place this checkout did not take place okay. But you cannot say failure happened I did half of it I did not do the other half of it.

So, in this lecture, we will see what the techniques to ensure atomicity of transactions. This is also a part of fault tolerance. So, we have studied this principle of modularity, applications are built as modular components and with modularity atomicity is very, very important.

(Refer Slide Time: 04:40)



Atomicity and modularity

- Atomicity is important when building modular applications
- Example: module A sends request to module B to perform some operation, e.g., front end sends request to shopping cart server to add item to a cart
- If processing the request needs multiple operations at B, they should all be performed atomically. Why?
  - Ideally, if B completes all operations, B returns successful response to A. If request execution fails at B, no operation executed, error returned, A retries entire request
  - If B performs only some operations and not others, what response will it return? How should A retry? Makes design messy
- Design guideline: module's API should expose fine-grained services which can be executed / implemented atomically

If you do not have atomicity, then it is very hard to build modular applications. Why is that? So, suppose you have module A that is sending some requests to module B, module B has some API it has a set of requests that it is exposing, you can ask B to do all of these different things and A is making a request to be asking it do a particular operation.

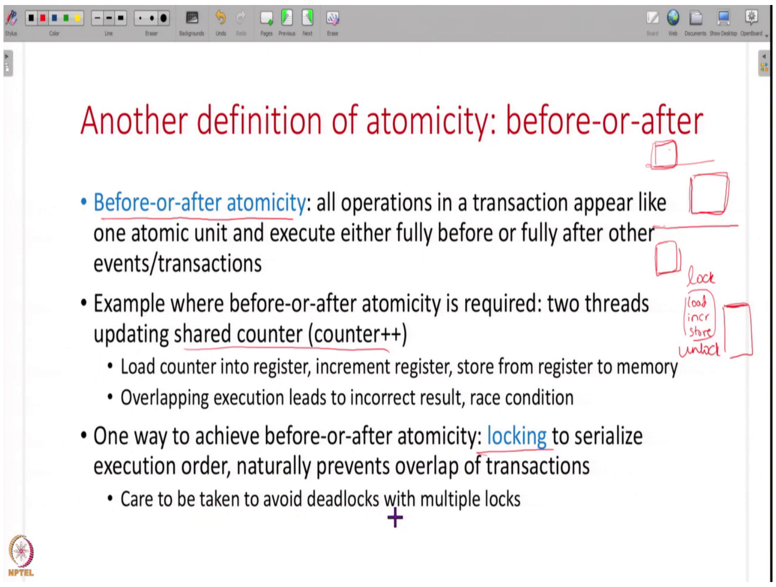
Now, one processing this request, suppose B have to do multiple steps to handle this request then if B completes all of these steps, then B will send back a positive reply to A saying your operation is done. Now, if B could not complete any of these steps, then B will send a failure or no response will be returned to A that is okay, then he will retry the entire request.

If B does all the steps we are good. If B does not do any of the steps, we are good. But what if B performed some of the operations I did these two steps, but I did not do the other two steps. In such cases, what is the response that B will return back to A it cannot return a positive response, it cannot return a negative response and how will A retry, there is no API request to say do half of the previous request, that does not exist.

So therefore, if you want correctly, clean modular applications, then you need atomicity for component is exposing an API and saying I will do this job for you, then you better do the job fully or do not do it at all, in which case, I will retry that is okay, if you do not do anything, I will retry if you do it well and good. But if you do half of it, then the other component is kind of left in a lurch. It does not know how to retry.

So therefore, when you design APIs of modules, always keep this in mind give your API's in such a way that you can ensure atomicity do not have an API where you are doing like 20 different things, and therefore, sometimes you do half of it, sometimes you will not do half of it, then your return values will all be messed up. So, therefore, when you design APIs make each service fine grained enough that you can execute it atomically.

(Refer Slide Time: 06:41)



Another definition of atomicity: before-or-after

- Before-or-after atomicity: all operations in a transaction appear like one atomic unit and execute either fully before or fully after other events/transactions
- Example where before-or-after atomicity is required: two threads updating shared counter (counter++)
  - Load counter into register, increment register, store from register to memory
  - Overlapping execution leads to incorrect result, race condition
- One way to achieve before-or-after atomicity: locking to serialize execution order, naturally prevents overlap of transactions
  - Care to be taken to avoid deadlocks with multiple locks

So, there is also another definition of atomicity, which we have used previously in the course, which is what is called before or after atomicity that is if you have a sequence of operations, these operations should appear like one unit, they should either occur completely before or completely after other operations, but there should not be overlap between these operations.

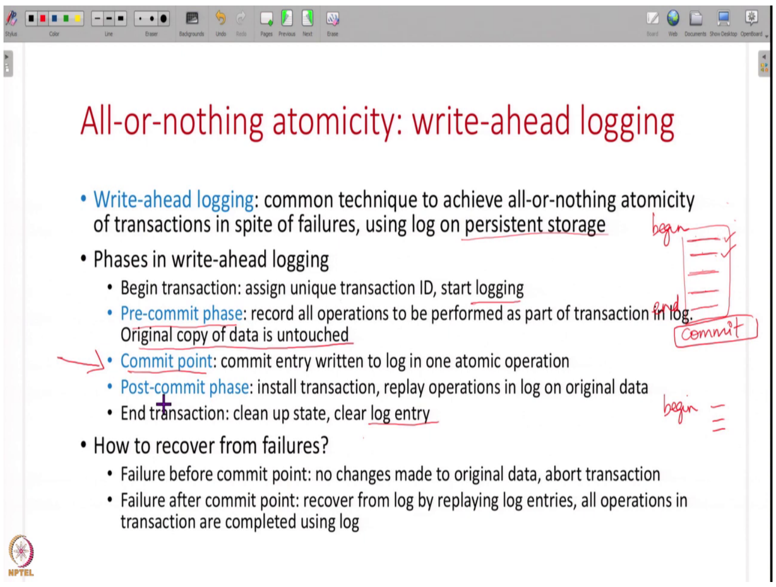
So sometimes you want transactions to be fully before or fully after other transactions. But you do not want this kind of an overlapping scenario, we have seen an example of this, when threads access shared data, for example, if they access a shared counter, then you have these three steps where you load a counter, then you increment the register value, then you store it back into memory.

And we have seen that if the sequence of operations overlap in a certain weird way, then you will have race conditions, and your counter is not updated correctly. We have seen all of this when we were discussing threads. So, in such cases, what we said was, we need atomicity, either one thread fully finishes, then the other thread starts, or after this thread finishes, the first thread starts, but you cannot have this overlap, that is also considered as atomicity.

But to differentiate between the two types of atomicity that is all or nothing atomicity this is before or after atomicity, a transaction appears like one unit, it will either fully be on or before another transaction or fully after another transaction. And we have seen this we have studied before, how do you do this, if you remember you use locks.

So, each thread will acquire a lock do its set of operations release the lock, therefore, this kind of overlap cannot happen, because there is a lock statement here and there is an unlock statement here, therefore, this thread cannot start executing in the middle. So, locking and of course, we have studied locking, deadlocks, how do you take care when you have multiple locks all of this we have studied before. So, this is another definition of atomicity which is called before or after atomicity but in this lecture, we will mostly focus on all or nothing atomicity. Now, how do you do all or nothing atomicity?

(Refer Slide Time: 08:49)



All-or-nothing atomicity: write-ahead logging

- Write-ahead logging: common technique to achieve all-or-nothing atomicity of transactions in spite of failures, using log on persistent storage
- Phases in write-ahead logging
  - Begin transaction: assign unique transaction ID, start logging
  - Pre-commit phase: record all operations to be performed as part of transaction in log. Original copy of data is untouched
  - Commit point: commit entry written to log in one atomic operation
  - Post-commit phase: install transaction, replay operations in log on original data
  - End transaction: clean up state, clear log entry
- How to recover from failures?
  - Failure before commit point: no changes made to original data, abort transaction
  - Failure after commit point: recover from log by replaying log entries, all operations in transaction are completed using log

The most common idea in computer systems is what is called write-ahead logging or simply called logging, which is all the steps that you want to do in your transaction you write them, you log them onto some persistent storage and then you execute them. This will guarantee atomicity. How is that possible? Let us see in more detail. So, whenever you are doing write-ahead logging you have a transaction, you have a series of actions that you want to perform atomically that is called a transaction. So, you do not just directly go start executing these operations one by one, then you will not have atomicity, what if you do half of it and then you crash.

So, instead you will execute them in the following manner, you will first start a new transaction assign a unique transaction ID then you will identify some log, log is nothing but some storage space in some persistent place like a disk or a data store, database something where it will stay safe even if failures occur, power failures occur.

So, for example, you cannot store your log in main memory because that can get wiped out after failure. So, you will store it in some persistent storage. So, you will first assign a unique ID for this transaction, you we will start logging and what you will do is in the pre-commit phase, so this logging, this write-ahead logging algorithm has multiple parts.

So, the first part is what is called the pre commit phase, in the pre commit phase what you will do is you will record all the operations that are part of this transaction that have to be performed as part of this transaction, you will not perform them yet, the original copy of the data you will not touch but you will only note down I will do this, I will do this, I will do this, I will do this in the pre commit phase.

Then after logging all of these operations then you will commit you will write an entry saying I am done, this is my list, this is the beginning of my set of operations, this is the end, you will write a commit point into your lock. After this then you will start installing your transactions, then you will start replaying or installing these transactions executing these operations one by one using your lock.

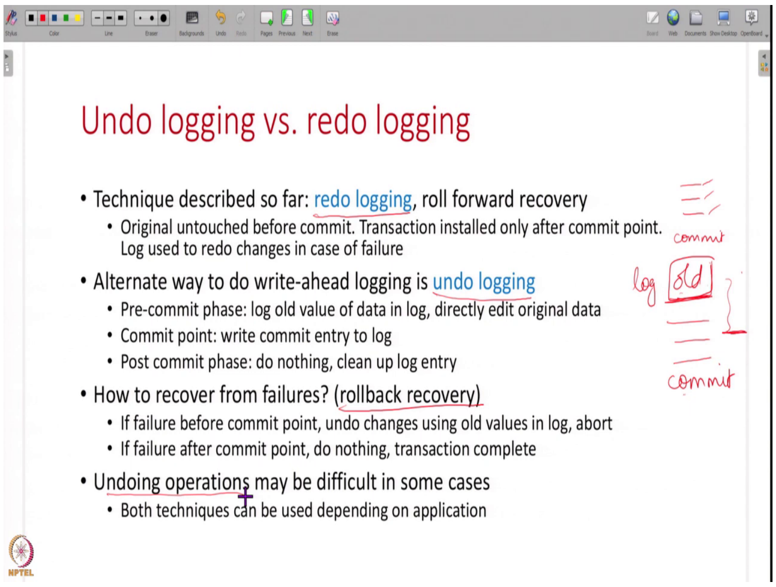
Now, you start modifying the original application data and once all of this is done of course, then your transaction is completed, all the sequence of steps have been done then you will clean up your state and you will clear this log entry that is the end of your transaction. So, how is this helping us? So, now let us see how this is helping us to recover from failures.

So, suppose you have a failure before this point, in this pre-commit phase, you have a failure, then how will your log look like? Your log will look like there is some begin and there is a bunch of transactions, but that is it there is no end, there is no commit nothing, you have failed as a power failure, you restart back again, then if you look at your log somewhere on disk, you will find something like this, then you know that the set of steps is not complete, therefore, you will not do anything to this transaction, you will abort this transaction.

And your original data has not yet been touched in the pre-commit phase. So therefore, it is as if nothing happened, no change has been made, and you are good. What happens if there is a failure after the commit point? You have still not completed all of these operations, you only done some of this, but there is a failure. But the good thing now is you have the list of operations in your log, when you restart, you see this log entry that it did not finish because if it would have finished, you would have cleared the log entry, therefore that it did not complete, it failed somewhere in between.

So, what will you do, you will start replaying the log, you will start executing all of these operations one by one, and you will get to the end of the list and then you will clear the log entry. And therefore, if a failure happens after the commit point, then all the operations in your transaction are complete. If it happens before the commit point, nothing is complete. So, either way, in no matter where the failure occurs, you have all or nothing semantics in your application. This is the key idea of write-ahead logging.

(Refer Slide Time: 12:59)



Undo logging vs. redo logging

- Technique described so far: redo logging, roll forward recovery
  - Original untouched before commit. Transaction installed only after commit point. Log used to redo changes in case of failure
- Alternate way to do write-ahead logging is undo logging
  - Pre-commit phase: log old value of data in log, directly edit original data
  - Commit point: write commit entry to log
  - Post commit phase: do nothing, clean up log entry
- How to recover from failures? (rollback recovery)
  - If failure before commit point, undo changes using old values in log, abort
  - If failure after commit point, do nothing, transaction complete
- Undoing operations may be difficult in some cases
  - Both techniques can be used depending on application

Now, there are a couple of ways in which you can do this write-ahead logging, what we have described so far is what is called redo logging. That is, you do not touch the original data initially, you will only log all the changes, and then you will say commit. And then you will proceed to install all of these changes.

So, during the recovery phase, it is called roll forward recovery. During the recovery if a failure happens in between a transaction that during the recovering, you are installing the transactions, you are making your changes to the original data, but pre-commit, no changes are made to the original data and any recovery is done by redoing the operations.

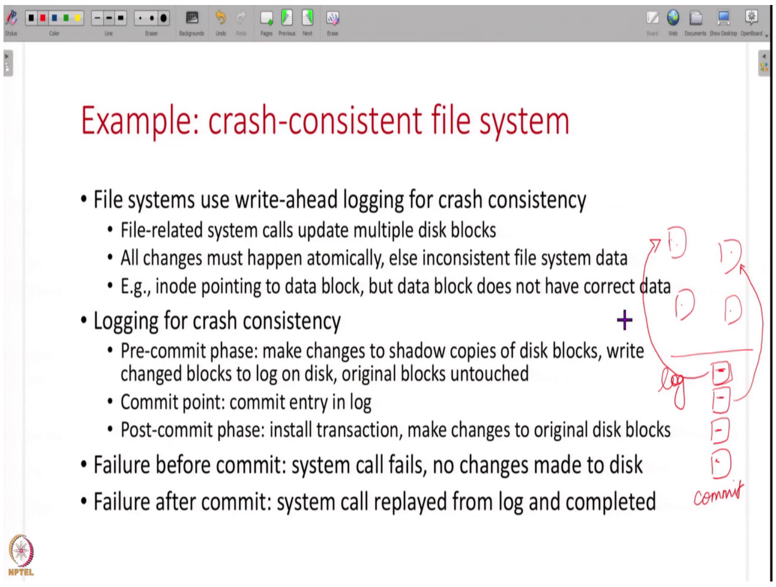
An alternate ways what is called undo logging that is in the pre commit phase, you will log the old value of data and you will directly make changes to the original data. So, your original copy is edited, but only after the old value is preserved, so now you have all the old value stored in your transaction and then you will write a commit entry. Why is this done?

Because if you have changed, made only some changes and you crashed in between you have not finished your transaction, then when you restart, you can see that some changes I have made but I have not made all the changes, I do not see a commit point therefore the transaction is not complete. Using these old values, you can undo the changes during recovery that is called roll back recovery, you can roll back If your transaction has five steps, you only did three steps you did not finish the last two. So, what you do, you will roll back you will undo these first three steps also so that you have your all or nothing semantics.

And how are you undoing? You are able to undo because you have the old value stored in your log, you are not just directly editing your first saving the old value then editing the original data. So, you can see that there is a slight difference in semantics between redo logging and undo logging but both of them achieve the same purpose in the end.

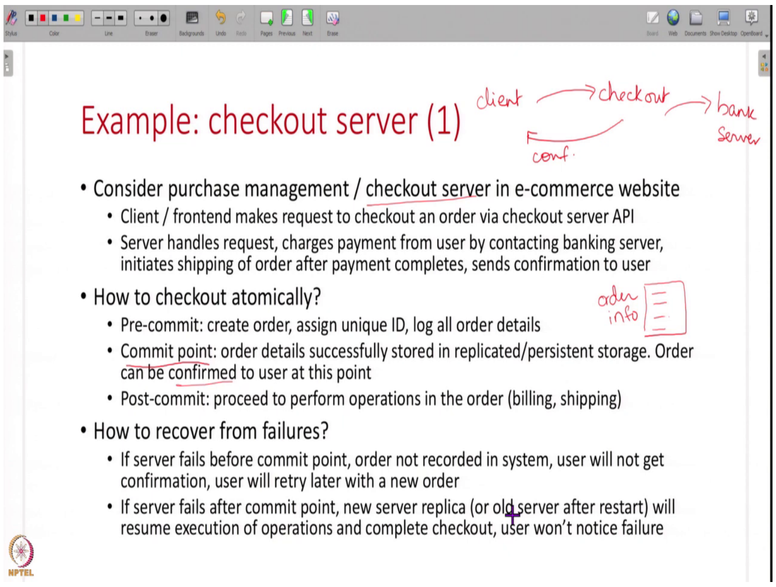
And you can choose which technique you want depending on your application, for example, undoing operations may not be easy in some applications, you cannot say you cannot add money into a bank account and later on say, sorry, I am undoing that it may not look good. So, depending on your application, you can either do redo-based write-ahead logging or undo based write-ahead logging.

(Refer Slide Time: 15:22)



Example: crash-consistent file system

- File systems use write-ahead logging for crash consistency
  - File-related system calls update multiple disk blocks
  - All changes must happen atomically, else inconsistent file system data
  - E.g., inode pointing to data block, but data block does not have correct data
- Logging for crash consistency
  - Pre-commit phase: make changes to shadow copies of disk blocks, write changed blocks to log on disk, original blocks untouched
  - Commit point: commit entry in log
  - Post-commit phase: install transaction, make changes to original disk blocks
- Failure before commit: system call fails, no changes made to disk
- Failure after commit: system call replayed from log and completed



Example: checkout server (1)

- Consider purchase management / checkout server in e-commerce website
  - Client / frontend makes request to checkout an order via checkout server API
  - Server handles request, charges payment from user by contacting banking server, initiates shipping of order after payment completes, sends confirmation to user
- How to checkout atomically?
  - Pre-commit: create order, assign unique ID, log all order details
  - Commit point: order details successfully stored in replicated/persistent storage. Order can be confirmed to user at this point
  - Post-commit: proceed to perform operations in the order (billing, shipping)
- How to recover from failures?
  - If server fails before commit point, order not recorded in system, user will not get confirmation, user will retry later with a new order
  - If server fails after commit point, new server replica (or old server after restart) will resume execution of operations and complete checkout, user won't notice failure

So, let us see some examples, so this crash consistent files system, this is what we have seen before, when we studied file systems. If you are using file systems, in a situation where you need highly consistent entries on your disk, then you will have to use some kind of logging so that even if crashes happen in between system call, your file system remains in a consistent state, it will never be the case that you have added a pointer to a data block to an inode you did not write data into the data block. So, your file will have garbage such scenarios will never occur if your file system is crash consistent.

So, how will you do that? Whenever your file system updates multiple disk blocks, what you will do is in the pre commit phase, here are all your disk blocks that are the inode, data block bitmap, all of these have to be updated as part of a system call, you will first make the changes on a separate copy and you will write all of these change disk blocks to your log, you will not change the original disk blocks, you will not change the original inode.

But you will make a copy of the inode and you will edit it here, you will edit the data block here edit the bitmap here, all of these changes you will make in the log, then you will write a commit entry. And now you will start installing, you will copy this disk block onto its original location, you will merge this copy with the original location and so on, you will start installing the transaction, so that if a failure happens in the pre-commit phase, then you have not touched the original values, you are good.

If a failure happens in the post commit phase, then you already have all the changes ready with you, you simply go and replay them from the log, in this way your system call will be completed. So, write-ahead logging is used to ensure that your file system is consistent in spite of a crash. Now, let us look at a more complicated example.

Suppose you have, again, going back to our standard example of an e-commerce system. Suppose you have a server that is managing all the checkout requests, user has added some items to a shopping cart and the user says checkout, then you have to bill the user's credit card for all of these purchases, and then you have to ship the order.

So, let us consider some such checkout server and there is an API, this checkout server has an API where you give it a shopping cart, it will check out that order and it will ship the order will do the billing and shipping for you. So, you have your front end or client whoever is making a request to this checkout server.

And this checkout server will do the billing and maybe to do billing, it will contact the bank or somebody else some other server it will contact to do the billing and then it will contact somebody else through the shipping it will finish all of these operations and it will send a confirmation back to the user this is your checkout server.

And now suppose let us take this example and let us see now this checkout server has to do two things atomically it has to do the billing and shipping atomically we have seen that, it cannot be this case that thay has bbbilled your credit card, but then a failure happened and it forgot do the shipping that should not happen. So, how will you do this checkout atomically?

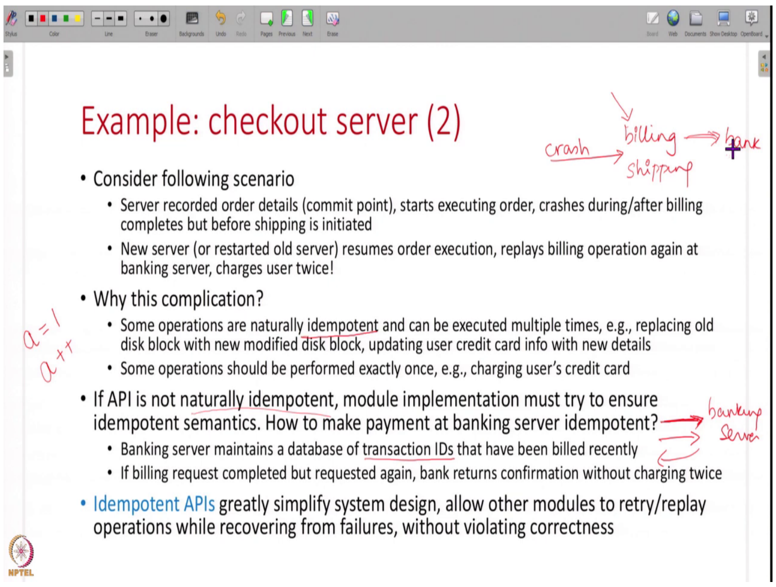
Again, you can use the same idea of write-ahead logging, instead of directly executing the operations the billing and shipping what this checkout server will do is it will create some entries, it will create order information where it will note down all the details of the order these are the items, this is the billing information, this is the shipping information, all this order details will be created and they will be stored somewhere in some database, in some persistent storage somewhere this order information will be stored, it will all be logged all the details.

At this point that transaction is committed and this can be a confirmation can be sent to the user. And now you start performing, all the information is there you start executing them one by one, billing, shipping everything you start executing, so that even if in all of these steps somewhere in between you fail, then when you restart, you can resume, you know all the information is there with you and you can resume.

If the server fails before the commit point, of course, the order is not even recorded in the system. There is no confirmation sent to the user, everything is lost, and the user will retry that is okay, the user will retry with a new order and the second time hopefully it will happen. And if the server fails after the commit point, the billing is done then all the information is there at the server.

Either a new replica, a new passive replica became active or the server only restarted, whoever is continuing this operation will have all the information in the log can resume execution of the operation. And the user will not notice a failure, in this way you can use write-ahead logging to first note down everything you have to do, and then do it. So, that in between somewhere, if you stop, you still have a record and you can do it again properly.

(Refer Slide Time: 20:29)



Example: checkout server (2)

- Consider following scenario
  - Server recorded order details (commit point), starts executing order, crashes during/after billing completes but before shipping is initiated
  - New server (or restarted old server) resumes order execution, replays billing operation again at banking server, charges user twice!
- Why this complication?
  - Some operations are naturally idempotent and can be executed multiple times, e.g., replacing old disk block with new modified disk block, updating user credit card info with new details
  - Some operations should be performed exactly once, e.g., charging user's credit card
- If API is not naturally idempotent, module implementation must try to ensure idempotent semantics. How to make payment at banking server idempotent?
  - Banking server maintains a database of transaction IDs that have been billed recently
  - If billing request completed but requested again, bank returns confirmation without charging twice
- Idempotent APIs greatly simplify system design, allow other modules to retry/replay operations while recovering from failures, without violating correctness

So here, but there is a small complication, if you are thinking through this carefully, you will notice a small complication. So, consider this scenario, the server recorded all the details of the order, and it starts executing the order, it has a sequence of steps, it has to do it, it has to do first billing, it contacted the bank to charge the credit card. At this point, it was just about to do the shipping, it did not do the shipping yet and at this point, the server crashed.

And then the new server replica came up or maybe the old server restarted, whatever, and then it look at its log, this order is there in my log, it will start executing the order, then it will start replaying the order again. And then it will start once again, I will say billing and it will once again bill, your credit card, I will talk to the bank charge you again. Is this acceptable?

Of course not, it is not acceptable, but you might be wondering what went wrong, we did the same thing with the file system, and it worked. And now why is it failing? So, here is a complication. So, there is this property of some operation being idempotent. What is idempotent operation is if you repeat the operation multiple times, also, no harm is done.

Whether you execute the operation once or you execute the operation 10 times the effect will be the same; such operations are called idempotent operations. For example, writing the value of one into a variable A, this is an idempotent operation, if I do it once, it will have the value of one, if I do it 10 times it will still have the value of one.

On the other hand, incrementing the value of A, this is not an idempotent operation. If I do it once the value will be incremented by one, if I do it 10 times the value will be incremented by 10. So, this is not an idempotent operation. So, replacing old values of the disk block with new modified disk block, where we have seen in the file system, you log all the changes and you install these changes one by one into the original disk blocks. That was an idempotent operation. Why, because you are taking the entire disk block overwriting the old disk block.

But charging a user's credit card that is not an idempotent operation, you charge 10 times 10 times the amount will be charged. So, some API's in your system are not naturally idempotent, and therefore, for such API's, this replaying the operations after a restart such mechanisms is redo logging, all of this will not work very well.

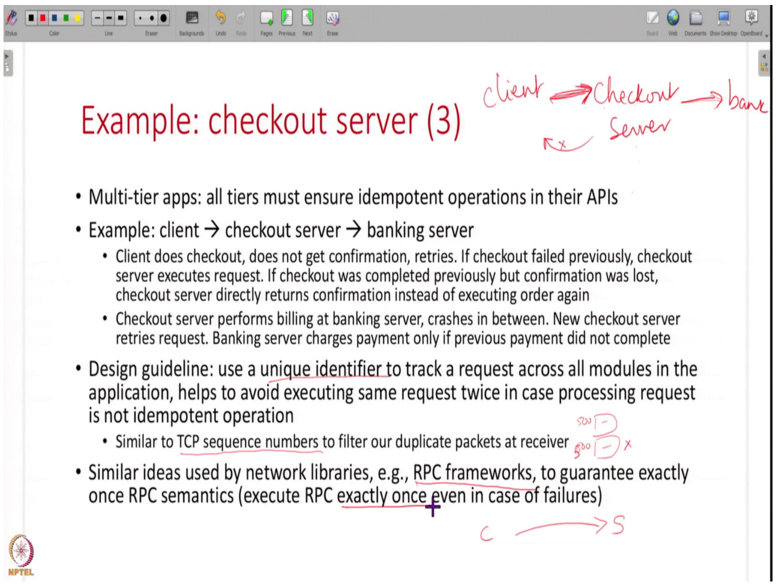
So, how do you fix this problem? The best way is your modules implementation must somehow try to ensure that any API that you expose is idempotent. For example, if your banking server is exposing some API, saying charge a credit card, it is exposing some API using which you can send a request to charge a credit card then the banking server should ideally implement this API in an idempotent manner.

For example, what it can do is it can maintain a database of all the transaction IDs that have been billed recently and, you created an order, you told the banking server to bill this order, charge this credit card. And now after 10 seconds, you come back and say bill the same order, then the banking server has to realize if it keeps a list of all the transactions, it will realize this particular order ID of this particular user it has been billed just now therefore, I will not repeat it again and it will simply send you a confirmation saying done.

In this way, the banking servers' implementation can make this credit card payment an idempotent operation. In which case if you come back to this scenario, the server did the billing or in the middle of billing, it crashed. Then when a new replica comes up, it will once again request billing but the bank will not charge twice.

Why? because this API to the bank is idempotent so the bank will say if the previous billing did not happen, then it will charge if the previous billing happened it will simply send back a confirmation saying your order has been billed it will not charge it again twice. So, if you have idempotent API's, if you are a module and you are implementing all your requests in an idempotent manner, then it is very easy to implement operations atomically, it is very easy to implement atomicity of transactions.

(Refer Slide Time: 24:53)



Example: checkout server (3)

- Multi-tier apps: all tiers must ensure idempotent operations in their APIs
- Example: client → checkout server → banking server
  - Client does checkout, does not get confirmation, retries. If checkout failed previously, checkout server executes request. If checkout was completed previously but confirmation was lost, checkout server directly returns confirmation instead of executing order again
  - Checkout server performs billing at banking server, crashes in between. New checkout server retries request. Banking server charges payment only if previous payment did not complete
- Design guideline: use a unique identifier to track a request across all modules in the application, helps to avoid executing same request twice in case processing request is not idempotent operation
  - Similar to TCP sequence numbers to filter our duplicate packets at receiver
- Similar ideas used by network libraries, e.g., RPC frameworks, to guarantee exactly once RPC semantics (execute RPC exactly once even in case of failures)

So, especially in a multi tier application, you have many different modules, you have your client or a front end that is talking to this checkout server and which is talking to some banking server for payment, some other shipping module for shipping, all of those you have a complicated multi tier system. At every level, you should ensure idempotent operations.

So, how do you do this? For example, your client has made a request to check out a shopping cart. If the checkout server finished all the operations, it sent a confirmation back, but somehow this confirmation was lost, for whatever reason. Now, the client will once again come back and say check out this order then, this checkout API should be idempotent, the server should remember that this particular shopping cart, I have just checked it out, I do not know why this guy is asking me again. So, it will not execute the order. Again, it will simply send back a confirmation saying done.

But of course, if the previous request failed, for whatever reason, and that is why the user is retrying, then the checkout server will execute the request correctly. Similarly, at the bank also, if the checkout server asked the bank to charge a credit card after and again, it makes the same request again, if the previous request succeeded, the bank will simply send a response back, it will not re executed, but if the previous request failed, then the bank will re execute.

So, every API in the system has to be idempotent, then it makes it easy to design your systems. And how do you do this? One way is you give some unique identifier, when the user checks out that order that transaction has a unique identifier, so that everywhere in the system, you are tracking that request using this unique identifier, so that when that request comes the checkout server will look, this identifier I have already processed it, I will not redo it again or I have not processed it I will execute it.

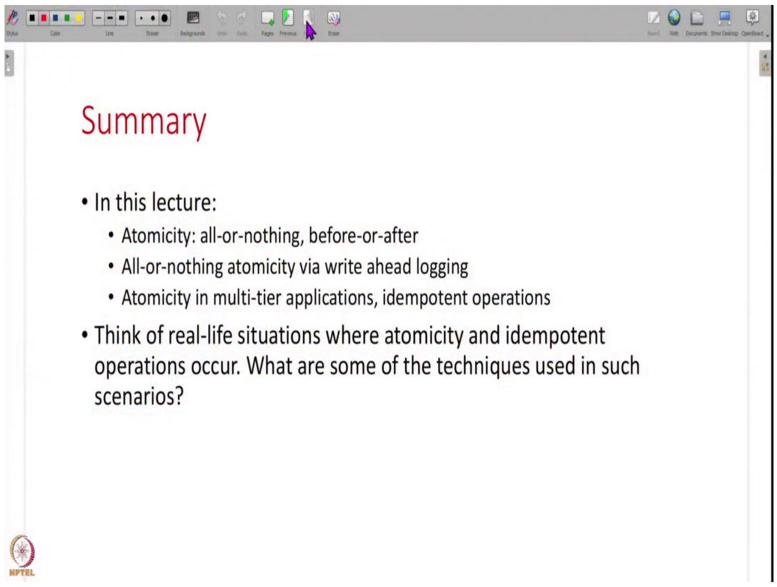
This is concept is similar to TCP sequence numbers, TCP, if there is a packet loss, and the same packet is retransmitted and somehow the receiver gets two copies of the data, it is not like in your file, that two copies of the data, some line in your text file or email will be repeated twice that will never happen.

Why? Because the receiver is using the sequence numbers, this is sequence number 500, another packet starting at sequence number 500, it will filter out the duplicates. And it will only use one copy in the final data it is given to the application. In this way, this concept of always filtering out duplicates, retrying, idempotent API's, these are all important if you are building a large system. And this really simplifies the design and implementation of your system.

And this idea is used in many places for example; if you have RPC frameworks some client is making some RPC to a server. And some RPC frameworks they guarantee exactly once RPC execution that is no matter what failures happen, we will execute it exactly once. How is this guaranteed again by you track this RPC request and if for some network failures, you repeat the request and the server will remember okay, this RPC request came just now I did it.

So, it will not redo it again, it will simply send a confirmation back saying I already done it, here is the final answer. In this way, several places in a complex system, the use of idempotent API's is very useful, so that you can do this write-ahead logging, you can replay your log, but still get correct results and the end.

(Refer Slide Time: 28:20)



Summary

- In this lecture:
  - Atomicity: all-or-nothing, before-or-after
  - All-or-nothing atomicity via write ahead logging
  - Atomicity in multi-tier applications, idempotent operations
- Think of real-life situations where atomicity and idempotent operations occur. What are some of the techniques used in such scenarios?

So, that is all we have for this lecture. In this lecture, we have discussed what is atomicity, all or nothing atomicity, before or after atomicity, then we have seen how you do all or nothing atomicity using write-ahead logging. And we have also gone into a little bit more detail. And we have seen that in real life systems, when in large multi tier systems if you are doing ahead logging and you are replaying your operations, then it is very important that all your API's are idempotent as much as possible so that even when you replay a request, it is executed exactly once within the module.

So, I wouldd like you to think a little bit more about these concepts of atomicity and idempotent operations in your real life in other systems which operations are idempotent? And how is this idempotent semantics guaranteed? What are the mechanisms used to make things atomic and to make things idempotent? So, think about these things a little more, not just from computer systems, but in other real-life interactions as well. So, that is all I have for this lecture. Thank you very much and let us continue our discussion in the next lecture. Thanks.