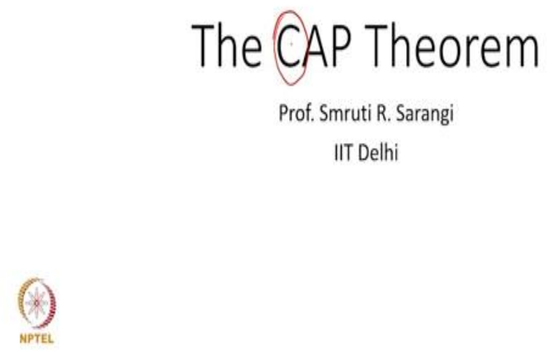
Advanced Distributed Systems
Professor Smruti R.Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology, Delhi
Lecture 26

CAP Theorem

(Refer Slide Time: 0:17)



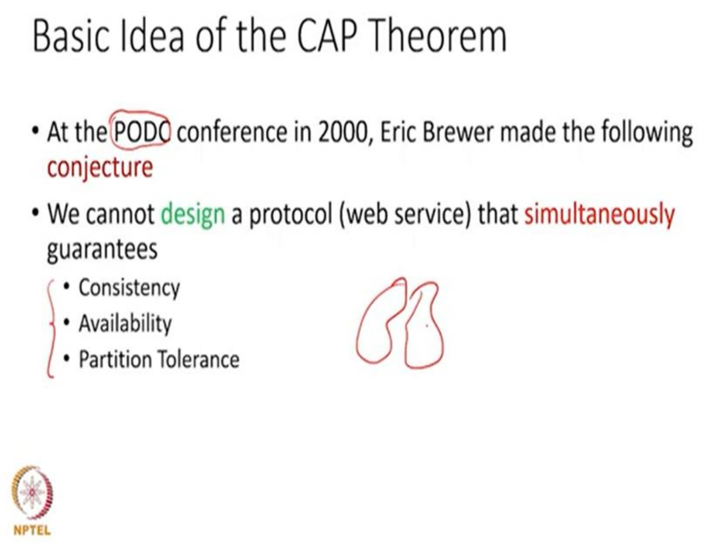
The CAP Theorem

Prof. Smruti R. Sarangi

IIT Delhi

Welcome to the lecture on the CAP theorem. So, the CAP theorem is a very important theorem in the design of distributed systems and distributed databases. So, in this lecture, we will discuss many other aspects related to the C of the CAP theorem, which is consistency. So, think of this lecture as a lecture where we discuss issues related with consistency and issues related with the CAP theorem.

(Refer Slide Time: 00:44)



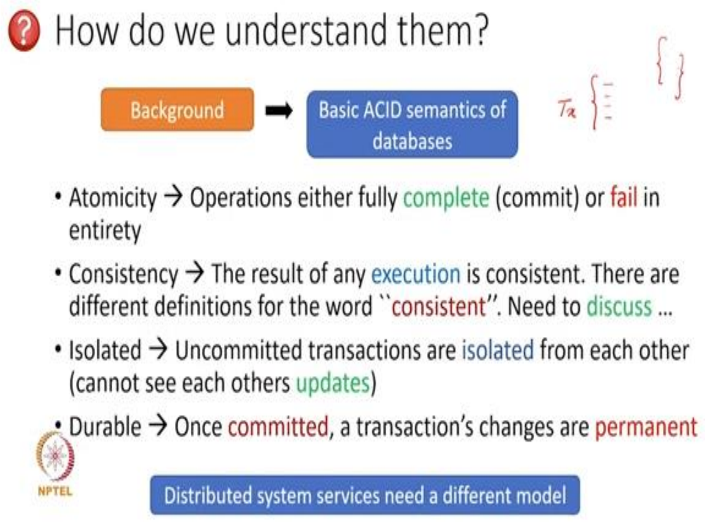
Basic Idea of the CAP Theorem

- At the PODC conference in 2000, Eric Brewer made the following conjecture
- We cannot design a protocol (web service) that simultaneously guarantees
  - Consistency
  - Availability
  - Partition Tolerance

So, the basic idea of the CAP theorem is like this. So, this theorem and an extension were initially kind of, were put forth to the community, basically, as a conjecture as a part of a talk it was put forth to the conjecture, and it is proved much later in some cases by the same person in some cases by others. So, the PODC conference, which is one of the top conferences in distributed systems, Eric Brewer made the following conjecture; that you cannot design a protocol or a web service that simultaneously at the same time guarantees the three properties which are as follows; consistency, availability, and partition tolerance.

So, we will discuss what these are? But essentially, these are desirable properties of any distributed system or any database where you want the data to be consistent, consistent with what? Consistent with specifications. What kind of specifications? Well, we will see. Availability means it should be available, in a sense every request gets a response. And partition tolerance means even if, the network kind of becomes two separate partitions, the system is still operational. So, he said you cannot guarantee all three together. So, this may sound obvious, but it was not or it still is not, if you actually rigorously apply it.

(Refer Slide Time: 02:05)


How do we understand them?

Background ➡ Basic ACID semantics of databases

- Atomicity → Operations either fully complete (commit) or fail in entirety
- Consistency → The result of any execution is consistent. There are different definitions for the word ``consistent''. Need to discuss ...
- Isolated → Uncommitted transactions are isolated from each other (cannot see each others updates)
- Durable → Once committed, a transaction's changes are permanent

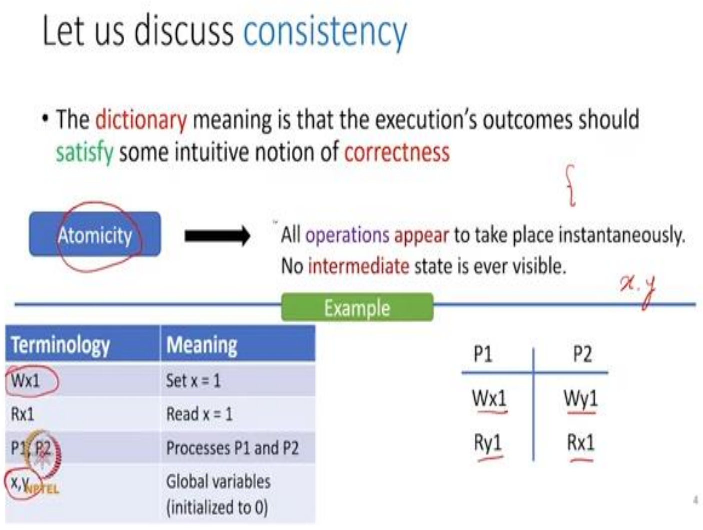
Distributed system services need a different model

So, how do we understand them? So, we need to understand that all of these theorems basically came out of the database literature. And in the database literature, we have the classic ACID semantics, which is something that needs to be discussed first, before we get into the CAP theorem. The operations are atomicity. So, the idea is that either a single operation or a bunch of operations known as a transaction, either all of it successfully completes, so, this is called a commit event or it fails in entirety, so, it appears as if the entire sequence of operations had never started in the first place.

The next is consistency is as I have discussed, consistency means the result is consistent with execution. The result of an execution is consistent with specifications. Of course, the specifications may themselves vary, but still there are some commonalities and different specifications which we will discuss. Isolated means it appears as if the transactions which are not committed isolated from each other. In the sense it appears that if let us say two transactions like two bunch of database transactions are happening at the same time, then they are happening in isolation, which means one cannot see the updates made by the other.

Final is durable, durable means once the transaction is committed. So, once we are done with an operation, the changes are permanent. So, they feel they are returned to permanent storage. So, distributed system services clearly need a different model the ACID model will not work for them. So, in this case, we adopt a different model which I will describe next.

(Refer Slide Time: 04:04)



Let us discuss consistency

- The dictionary meaning is that the execution's outcomes should satisfy some intuitive notion of correctness

Atomicity → All operations appear to take place instantaneously. No intermediate state is ever visible.

Example

Terminology	Meaning
Wx1	Set x = 1
Rx1	Read x = 1
P1, P2	Processes P1 and P2
x, y	Global variables (initialized to 0)

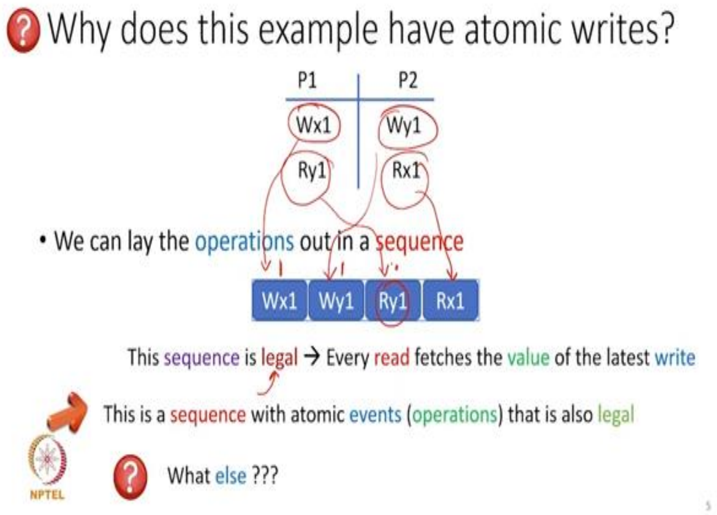
P1	P2
Wx1	Wy1
Ry1	Rx1

So, let us discuss consistency, so, the dictionary meaning as I said is that the outcomes of an execution should satisfy some written specifications or some intuitive notion of correctness as the dictionary meaning. So, if you break it down, so, we can break it into several components. The first component would be atomicity, which means that all operations appear to take place instantaneously. So, it is as if, given an operation or a bunch of operations, it appears to take place instantaneously in the sense you cannot see any intermediate state.

So, let us first define some terminology and then I will explain, select Wx1 mean, write 1 to the global variable x. Let Rx1 mean that I read the value of global variable x to be 1, let P1 and P2 be processes P1 and P2. And let x and y be global variables initialized to 0. So, what we have over here is P1 and P2 two processes, processing in a distributed system. So of course, whether you have shared memory or not, it does not matter.

But in this case, let us assume we have shared memory. In the sense they are operating on the same object. And objects are read-write objects, and they are of type x and y, both initialized to 0. So, let us look at this execution you write 1 = x, write 1 = y, read 1, read y = 1 and then you read x = 1, so, this is the execution.

(Refer Slide Time: 05:50)



Why does this example have atomic writes?

P1	P2
Wx1	Wy1
Ry1	Rx1

- We can lay the operations out in a sequence

Wx1	Wy1	Ry1	Rx1

This sequence is legal → Every read fetches the value of the latest write

This is a sequence with atomic events (operations) that is also legal

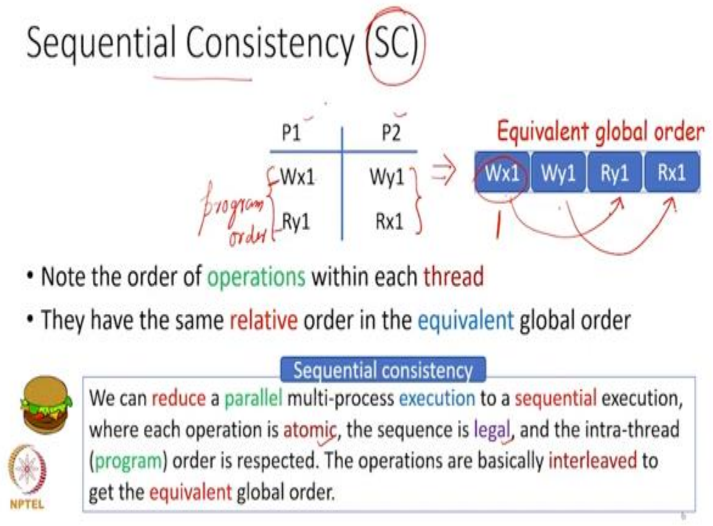
What else ???

So, what we want to do now is we want to see why does this example have atomic writes? That is because we can lay the operations out in a sequence where we will first have this Wx1, then we will have Wy1 which goes over here, we will read y to be 1, which again goes here and we will read x to the 1 which again goes here, so, Wx1, Wy1, Ry1 and Rx1. So, as you can see, we can lay them out.

And it appears that the write happened at 1 instant, then the next write happened at 1 instant, then the read again happened at 1 instance reads in general R atomic in all kinds of systems writes may not be, so, it read why as you wrote y 1 to y, so, you are reading 1, you are reading y = 1 and you wrote 1 to x and you are reading x = 1. So, you can lay them out, furthermore, the sequence is legal, which means if you look at every read, it fetches the value of the latest write.

Again, to the same address, it is fetching the value of the latest writes, R/1 is providing you Wy1 and Rx1 is providing with a value written by Wx1. So, this is clearly a sequence with atomic events, and this sequence is also legal. What else? What else is special about this sequence?

(Refer Slide Time: 07:21)



Sequential Consistency (SC)

- Note the order of operations within each thread
- They have the same relative order in the equivalent global order

Sequential consistency

We can reduce a parallel multi-process execution to a sequential execution, where each operation is atomic, the sequence is legal, and the intra-thread (program) order is respected. The operations are basically interleaved to get the equivalent global order.

So, let us look at this sequence once again, let us not look at the title of the slide yet. So, we can lay them out in an equivalent order. So, of course, these are two separate processes, but we can lay them out in a sequential order where it will appear that some master execution engine is choosing one operation from here then the next then the next and so on. So, you have Wx1, Wy1, Ry1 and Rx1.

You note the operations, order of operations within each thread. So, what we see here is that the within the threads, this is the first instruction, this is the second, so, clearly the intra trade order is Wx1 first and Ry1 after that, and we get to see that. Similarly, we see Wy1 first and Rx1 after that, so, we also get to see that. So, what we see is that if we create an equivalent global order number one it is atomic, because it appears even at this entire operation executed at one instant of time.

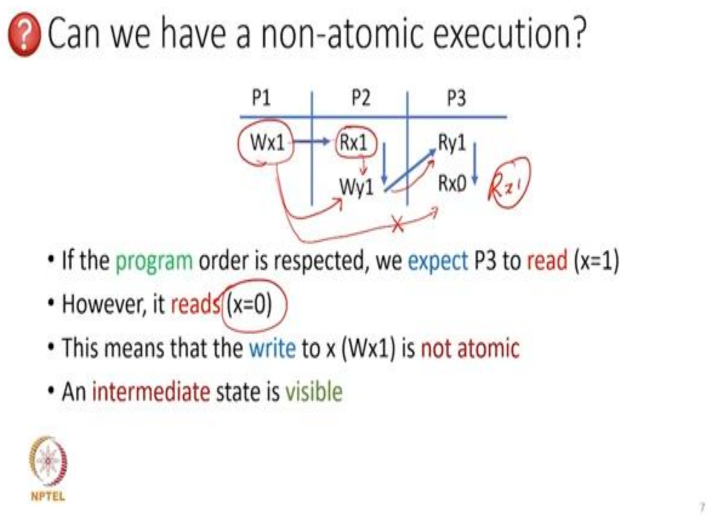
And this is true for the rest it is legal, as well as the order of operations within each thread is the same as the order in the equivalent global order. So, this is also known as a program order, so, the order within a thread is also known as the program order. So, this particular execution or its outcomes rather are such that so let us read this, so, we can reduce a parallel multi-process execution.

So, in this case, we have a parallel multi-process execution because we have two processes P1 and P2, we are kind of reducing it to a sequential execution, where three things hold each operation is atomic, the sequences legal and the intra-thread order or the program order is respected, so, all three things are happening at the same time. So, it appears as if you take the operations of the different processes. And you can interleave them the same way you take your fingers and you can interleave them, you can interleave the operations, you will have different global orders.

But the idea is that the execution is consistent with at least one global order. Fair three things are there atomicity, legality and the program order is respected. See, any search execution is known as a sequentially consistent execution you can clearly see where the name comes from; the name comes from the fact that a parallel execution is being equated with a sequential execution. So, you say that the execution is sequentially consistent.

So, SC is by far the most popular consistency algorithm in I would say all of distributed systems even though many a time enforcing SC itself can be quite impractical. But still the idea is that SC is quite popular it was published or proposed by Leslie Lamport way back in the seventies. But henceforth, it is kind of regarded as a gold standard of consistency in distributed systems of all kinds.

(Refer Slide Time: 10:44)


Can we have a non-atomic execution?

- If the program order is respected, we expect P3 to read (x=1)
- However, it reads (x=0)
- This means that the write to x (Wx1) is not atomic
- An intermediate state is visible

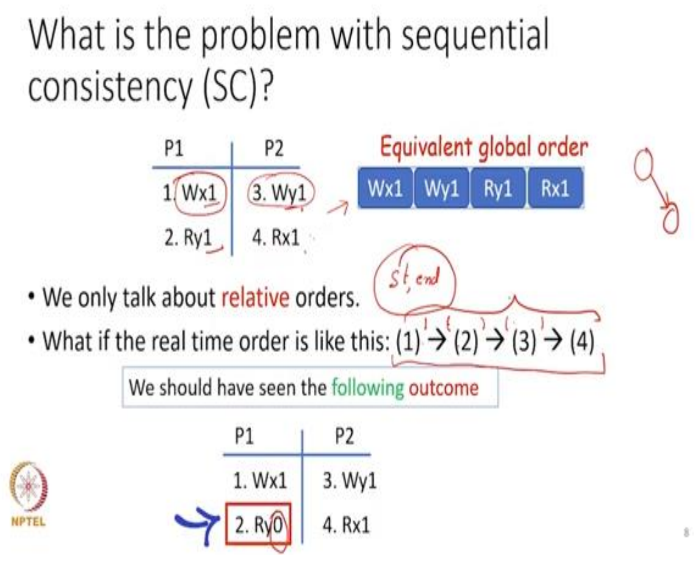
So, can we have a non-atomic execution? Well, we can. So, let us assume we have three processes, and you write 1 to x. So, process two reads x and it sees that the value is 1, so, this is fine, you can see that there has been a data transfer like that, and then assume that you respect program order in the sense you finish this operation and then you do this.

So, after reading x = 1 you write y and you write the value of 1 to y, then again you read y = 1 and finally, you read x, what do you expect? You expect that since there is a causality, since there is a chain of dependence over here, if you have written 1 to x you will also read x to be 1. So, you would expect to see Rx1 over here, but let us say you do not see it, if you do not see it, what does this mean? This means that this write has gone to P2 first and this has gone to P3 much-much later that is what you have seen that the write has gone to P3 much-much later and it has at this particular point it has not reached P3 but it has reached P2.

So, this means that if the program order is respected, we expect P3 to read x equal to 1, but in this case it has not, it has read x equal to 0 which essentially means that the writes are not atomic in the sense of the write Wx1 over here has not happened or has not appeared to happen at a single instant of time. Instead, it is appearing that it is like happening at different instants of time to different processes. So, in the first case it is appearing to have occurred before Rx1, in the second case it is appearing to happen after Rx0.

So, in this case, the writes are not atomic and in many distributed systems writes are not atomic mainly because to make a write atomic what you have to do? You have to broadcast the write to everybody and ensure that a process can only read a write when the update has been propagated to the rest of the processes. So, this is an atomic write problem and of course, there are algorithms to ensure atomic writes, but a naive implementation would lead to a non-atomic execution. So, in other words, in this case, you can see an intermediate state where the write has gone to one process, but not gone to another process.

(Refer Slide Time: 13:21)



What is the problem with sequential consistency (SC)?

P1	P2
1. Wx1	3. Wy1
2. Ry1	4. Rx1

Equivalent global order: Wx1 | Wy1 | Ry1 | Rx1

- We only talk about relative orders.
- What if the real time order is like this: (1) → (2) → (3) → (4)

We should have seen the following outcome

P1	P2
1. Wx1	3. Wy1
2. Ry0	4. Rx1

st, end
NPTEL

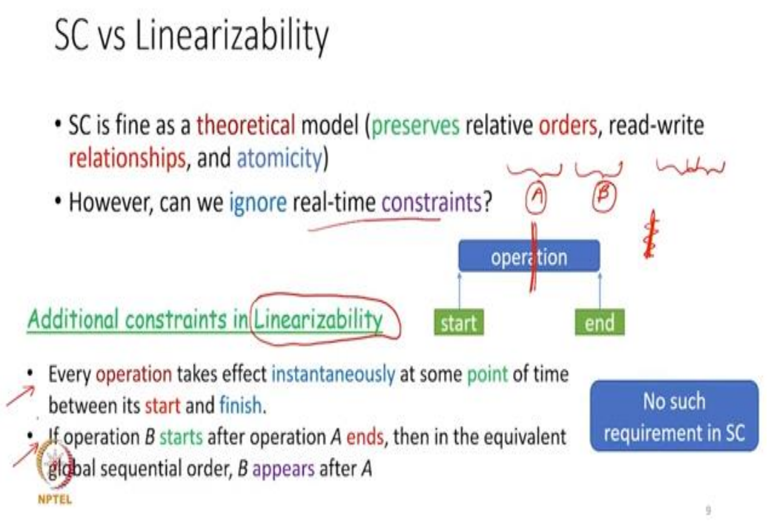
So, what is the problem with sequential consistency? Well, if we come back to the same example and the equivalent global order, the idea is you only talk about relative orders in the sense you say that look first, this operation happened then this, then this and then this. Fair enough, so, this you are guessing from the outcome, but the point is you are totally ignoring the real time the absolute time at which these operations happen. So, maybe it is possible that in terms of absolute time operation one happened first, operation two happened after that and then operation three happened after that.

So, in this case what would happen? In this case operation two will not read y to be 1, but it will read y to be 0 then operation three happened and then four, so the rest are fine. So, the issue is that if this was the real order of operations happening and let us say with each operation you can have a start time and an end time.

So, it is very well possible that after operation one ended operation two began, after operation two ended operation three began so on and so forth. So, in this case, what you will see is that even though this execution is sequentially consistent, it is not really consistent with the wall clock order, because, here we are only talking of relative orders first, this operation then this it has nothing to do with real time.

But the moment you bring in real time then and if let us say this was indeed the real time order fair operation one stopped operation two began, operation two stopped, operation three began so on and so forth, you will see that the outcome will not be this instead the outcome will change, and the outcome will become Ry0. So, if the outcome becomes Ry0 then, clearly in this case sequential consistency would give you a different answer and adhering to the real time order you will get a different answer.

(Refer Slide Time: 15:36)



SC vs Linearizability

- SC is fine as a theoretical model (preserves relative orders, read-write relationships, and atomicity)
- However, can we ignore real-time constraints?

Additional constraints in Linearizability

- Every operation takes effect instantaneously at some point of time between its start and finish.
- If operation *B* starts after operation *A* ends, then in the equivalent global sequential order, *B* appears after *A*

No such requirement in SC

So, whenever we are fine with just a relative order we will accept any outcome that sequential consistency gives. But the issue is that given the fact that SC is more of a theoretical model, because it preserves relative orders, and it preserves read-write relationships and atomicity. We call it a theoretical model, primarily because it ignores real-time values and constraints, so that is the reason we ignore it. So, if we put in additional constraints to SC, we come up with another consistency criterion. This consistency criterion is called linearizability.

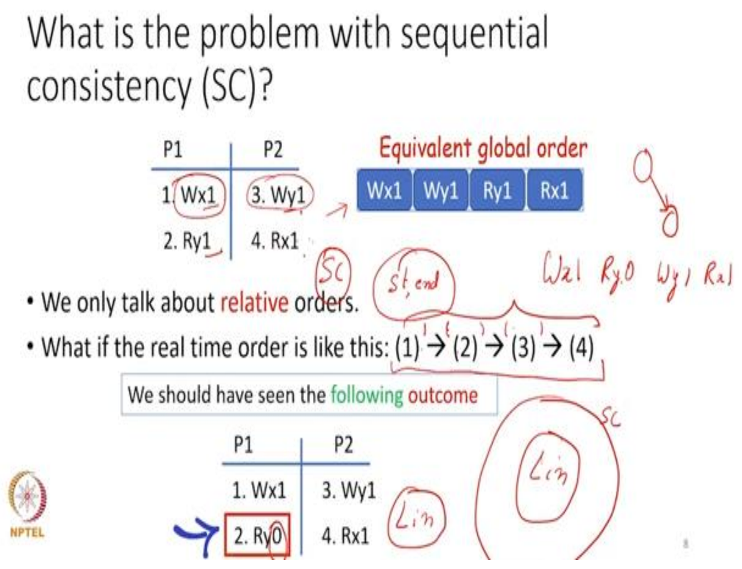
So, linearizability is something which is more powerful than SC. And the reason that it has been added is basically because there were issues with SC, and the issues were pointed out in the previous slide. So, in this case, the two additional caveats that we add (are like that) are like this, that every operation of course, has a start and end time. So, atomicity basically says that an operation needs to appear as if it is executing at a single instant of time. But this single instant of time could be any time in the sense it could be right over here after the operation has ended.

So, in this case, the results of the execution will not be consistent with the real-time order of operations, so, linearizability does not allow this it says that the time at which an operation appears to complete instantaneously, it still has to complete instantaneously has to be between start and end, that is the first point. So, then another thing it says is that assume operation B starts after operation A ends. So, this is operation A and this is operation B, then in the equivalent global sequential order that we create, so, we created that in the case of SC as well. SC is sequential consistency, B needs to appear after A, so, that is clear.

So, why is that the case because clearly these two are not concurrent operations? That is the reason A appears first and B needs to appear after A. Of course, if there are concurrent executions, like if this is the case, then linearizability does not say who needs to appear after whom because there is an overlap.

But again, since they appear to execute instantaneously, all threads should see either A appearing before B or B appearing before A in the equivalent sequential order. So, these are the two extra conditions, extra caveats that are added in the case of linearizability which SC did not have, so, they make it like more in line with real-time constraints. So, this is something that SC did not have.

(Refer Slide Time: 18:45)



What is the problem with sequential consistency (SC)?

P1	P2
1. Wx1	3. Wy1
2. Ry1	4. Rx1

Equivalent global order: Wx1 Wy1 Ry1 Rx1

- We only talk about relative orders.
- What if the real time order is like this: (1) → (2) → (3) → (4)

We should have seen the following outcome

P1	P2
1. Wx1	3. Wy1
2. Ry0	4. Rx1

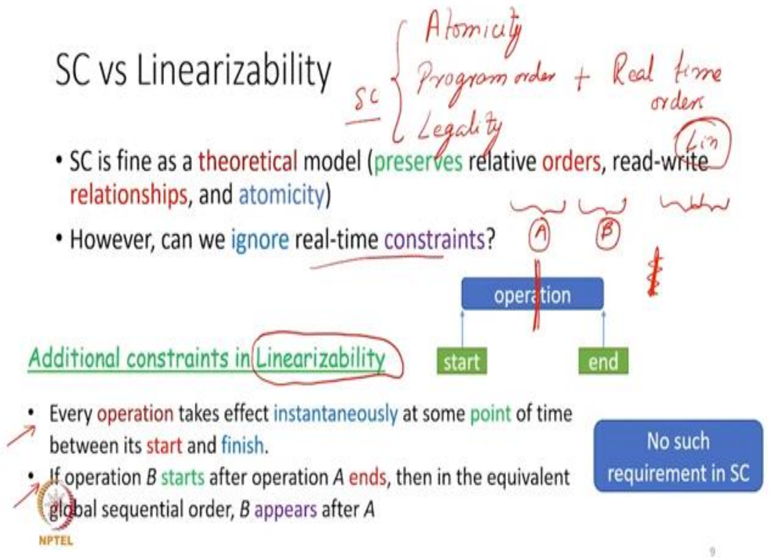
SC
st end
Wx1 Ry0 Wy1 Rx1
Lin
Lin
SC
NPTEL

So, that is why we had a little bit of an odd behavior as you could see over here because we expected Ry0, this was the real time order. But this was the execution results that we got, which is fine from an SC standpoint. But this is not fine from a linearizability standpoint, whereas if you take a look at this execution, this execution is linearizable, it is also in SC, but this execution the first one is an SC but it is not linearizable. So, linearizability a linearizable executions are essentially a subset of SC.

So, basically it sees a superset of linearizable executions say anything which is linearizable is an SC but not the other way around as we saw just over here, that this is an SC but it is not linearizable but this is linearizable as well as an SC, well, why? Because we can create a sequential order like this Wx1 then Ry0 then Wy1 and then Rx1.

So, as you can see this order is legal, this order respects program order and it is atomic, so, atomicity is there. So, given the fact that we have discussed atomicity how atomicity is added mixed with reliability and program orders to create SC? And how we add real time ordering to create linearizability? We can now move forward.

(Refer Slide Time: 20:22)



SC vs Linearizability

- SC is fine as a theoretical model (preserves relative orders, read-write relationships, and atomicity)
- However, can we ignore real-time constraints?

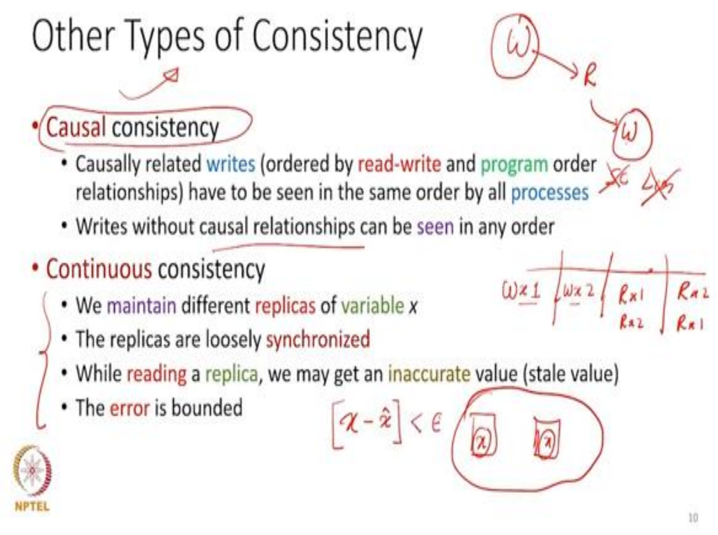
Additional constraints in Linearizability

- Every operation takes effect instantaneously at some point of time between its start and finish.
- If operation B starts after operation A ends, then in the equivalent global sequential order, B appears after A

No such requirement in SC

So, let me nonetheless just summarize with the little bit of extra space that I have over here and I will do that. So, the properties we have our atomicity then we have program order, legality, so, these were properties of SC, fair basically you have program order atomicity and every read needs to return the value of the latest write, if with this we add real time orders then what we get is linearizable. So, we have discussed all of that for the sake of, for the purpose of discussion of the CAP theorem, we will only use the only consistency guarantee that we will use will actually be atomicity. So, that is what we will use.

(Refer Slide Time: 21:26)



Other Types of Consistency

- Causal consistency
  - Causally related writes (ordered by read-write and program order relationships) have to be seen in the same order by all processes
  - Writes without causal relationships can be seen in any order
- Continuous consistency
  - We maintain different replicas of variable x
  - The replicas are loosely synchronized
  - While reading a replica, we may get an inaccurate value (stale value)
  - The error is bounded

And we will show that, that itself is sufficient to almost prove everything. But since, we have the opportunity; let us discuss other types of consistency as well. So, you can have causal consistency, which means, let us say if there is a write, and then there is a read. And after that, let us say there is a write, so, there is a program order relationship between this read and write. So, then everybody should see this write happening first and this write happening later. It should never be the case that you do a read after this, you see the results of this, but you do not see the results of this.

That is because two operations are related with cause effect relationships it is causal. So, you wrote and I read what you wrote, so, we have a cause effect relationship I executed, that is the reason you executed again, we have a cause effect relationship. So, as long as there is some sort of a cause effect or a causal relationship, we will expect them to be seen in the same order by all processes.

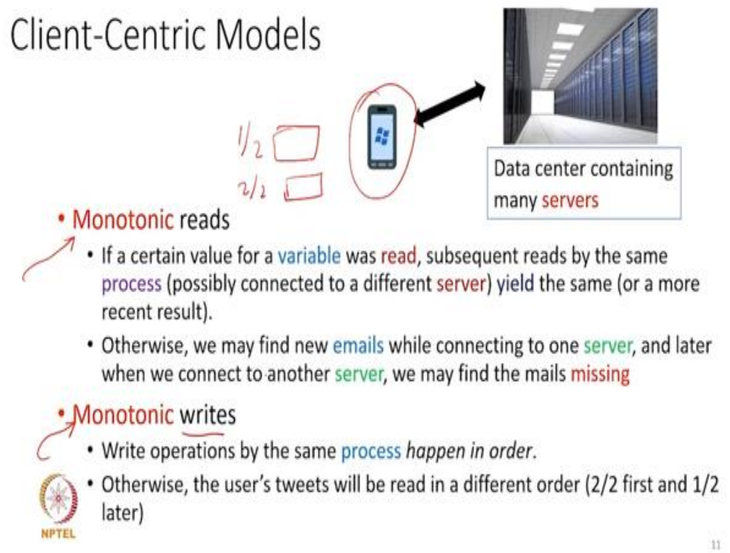
But, if you do not have causal relationships, they can be seen in any order. So, a wonderful example can be given to illustrate the point, let us say we are writing x = 1 and then we write x = 2, one thread reads X so, it sees first 1 and then 2, well, that is fine. So then, as far as we are concerned, if atomicity were to hold, then x would be written first with 1 and then with 2, but what if another thread observes Rx2 and Rx1. So clearly, in this case, the atomicity of x is not holding.

So, it appears that we are updating x in a different order. So, both the threads are not agreeing with respect to the order in which x was updated and as you can see there is no causal relationship between this write and this write. So, causal consistency at this point basically says you can see them in any order which the threads are doing. So, this is clearly not sequentially consistent this is clearly not linearizable but, it is causal consistent, causally consistent.

So, causal consistency is weaker, the same way that SC was weaker than linearizability, causal consistency is weaker than sequential consistency, so, we can again define this in a different way, let us say that we are maintaining many replicas of a variable. So, we have multiple servers, and they are maintaining many replicas of a variable x, you want to read it or write it. So, the replicas themselves could be loosely synchronized but let us see, if you read a value from the set of servers, you may get a replica which is stale, in the sense its value is old, it is inaccurate.

But in certain conditions, or let us say in this continuous consistency model, it is guaranteed that whatever is the value of x that you get, and whatever is the real value of x, which you should have gotten in ideal circumstances, the error is bounded. So, again, this is not a very generic model it is true only for some cases where you are dealing with things like clocks or counters and so on. But again, this is a consistency model where you look at the difference in the values and you try to bound that, you try to place a bound on that.

(Refer Slide Time: 24:58)



Client-Centric Models

1/2
2/1
Data center containing many servers

- Monotonic reads
  - If a certain value for a variable was read, subsequent reads by the same process (possibly connected to a different server) yield the same (or a more recent result).
  - Otherwise, we may find new emails while connecting to one server, and later when we connect to another server, we may find the mails missing
- Monotonic writes
  - Write operations by the same process *happen in order*.
  - Otherwise, the user's tweets will be read in a different order (2/2 first and 1/2 later)

So, we will now discuss a few more kinds of consistency models from the point of view of a client, so, who is a client? So assume that you are accessing Gmail. So, Gmail is a server, my mobile phone is the client, or you are accessing an office network that has lots and lots of documents. And I may be accessing it using my mobile phone or laptop or desktops in that case, my mobile phone or laptop is a client. So, from the perspective of the clients, up till now, we are looking at it from the perspective of the system. But from the perspective of the client, we could have many consistency models so, let me discuss a few.

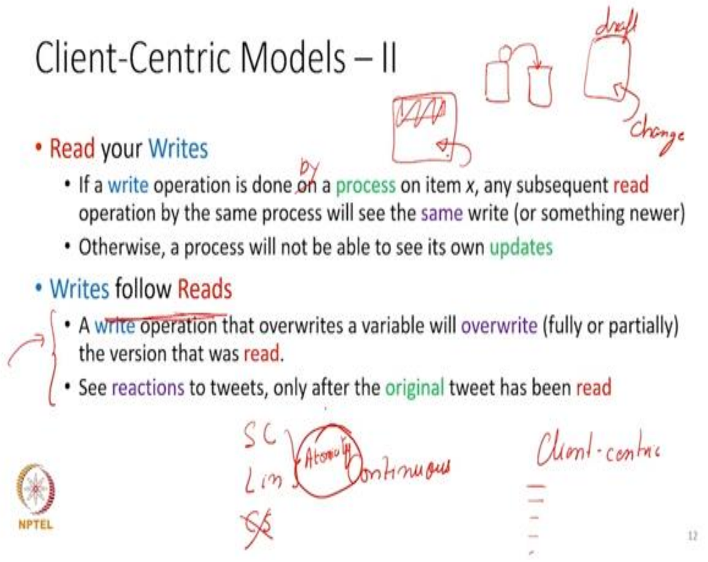
So, let us look at monotonic reads, so, if a certain value for a variable was read, so, in the sense, let us say, you read an email, subsequent reads by the same process, the same process means the same mobile phone, where it may connect to the same server or a different server will yield the same value or more recent result. So, which basically means that if let us say, connect to Gmail, and I am able to see an email, half an hour later, I connect to Gmail again, but internally Gmail connects me to a different internal server, I should still see that mail, it should not go away.

If this does not happen, what will happen is that I may see a mail now and sometime later, I may not see the mail. And later, when we connect to another server, we may find the mails to be missing, so, then there will be an issue. So, this is the monotonic reads, which is purely from the perspective of the client, we are not looking at it from everyone's perspective, as we were doing earlier.

We are gonna monotonic writes, which means write operations made by the same process happen in order. What would that be? So, let us say, in on Twitter, you would have seen many of these tweets, 1/2, 2/2, and so on, which is like the first part of the tweet and second part of the tweet. So, this I do from my Twitter client, but what if Twitter shows the second part of the tweet first and the first part of the tweet? Along with being incorrect, it can lead to quite embarrassing situations, this is definitely not what you expect and that is why this consistency model from the point of view of a client is called a monotonic write.

So, clearly, you can see that if atomicity is there, monotonic reads are not a problem, because once you have read, you will always read it. Monotonic writes are not a problem, because once you have written you have written, so, it is appeared to happen at an instant, so, the moment the first tweet is done, you can do the second and it will remain that way.

(Refer Slide Time: 27:52)


draft
change
SC
Lin
Atomicity
Continuous
Client-centric

Client-Centric Models – II

- Read your Writes
  - If a write operation is done ~~on~~ by a process on item *x*, any subsequent read operation by the same process will see the same write (or something newer)
  - Otherwise, a process will not be able to see its own updates
- Writes follow Reads
  - A write operation that overwrites a variable will overwrite (fully or partially) the version that was read.
  - See reactions to tweets, only after the original tweet has been read

12

So, I can extend this to have two more models read your writes and writes follow reads. So, read your writes is like this, that if a write operation is done on a process by a process and item x, any subsequent read operation by the same process will read will at least fetch the write that in its own write, or something newer of course, so, why am I saying that? Well, the reason I am saying that is let us say you compose a mail and you store it in drafts and later on, you read the drafts folder you are supposed to see the mail. If you do not see the mail, you will sort of see the mail vanish which again is something that atomicity will not allow.

That a fundamental level, if atomicity holds, you will never see that. But if let us say you have different email servers, and one email server holds the drafts folder, and it has not broadcasted the updates to the other this indeed can happen. Then we have writes follow reads. A write operation that overwrites a variable will overwrite fully or partially the version that was read, so, that is correct. So, let us say I read something I read and, I read a document or I read a draft email, and then I make a change. So, what am I reading? The object that I am reading is a draft, and then maybe I make a change on it I change a sentence.

So, I am doing write, so, what will happen is that if I have read something, the write operation will only overwrite the draft that I have read, it is not that the draft is going to change. For example, what I could have done is that for the same draft? I could have saved it twice, a smaller version and a bigger version, so, of course, at the end; the bigger version is the more recent one, so that is the one that should stay. So, what should happen is that the next time I open my drafts folder, I should not see the smaller version, I should see the bigger one there I make a change.

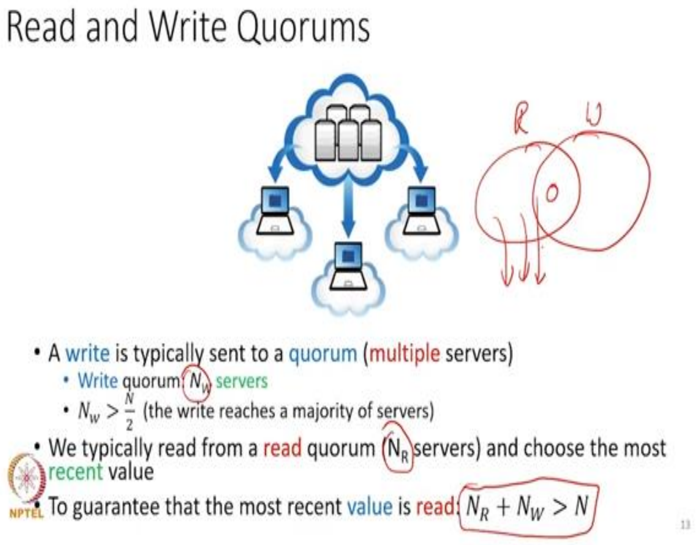
But what happens if while I am writing, the write goes to the smaller one and not the bigger one? Again, it can happen if I have multiple servers in these versions or these replicas of the draft mail are there in different servers and that could cause an issue, but in a good distributed system it should not happen. Of course, if you have atomicity, this will always happen, this condition will always hold.

Again, one more thing you have reactions to tweets where reaction is a write but that should only happen after the original tweet has been read so, it should not be the case. You read a tweet, you react on it. But then you see that the reaction is there, but the original tweet is gone so, that again, should not happen and atomicity would guarantee that this does not happen.

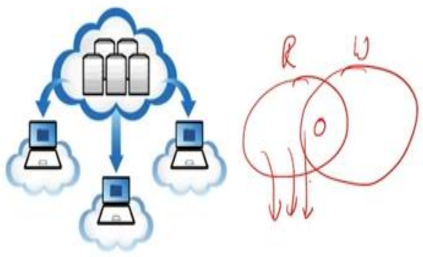
So, what is the crux of our idea? The crux of the idea is like this, that regardless of your consistency SC linearizable causally consistent, we also discussed continuous consistency, where we looked at errors, but ideally, the error should be zero. Then we discuss client centric models, different client centric models.

So clearly, in the first two atomicity was common, causally consistent we did not see atomicity that is why we did not like the results that it gave. Continuous again, if you have atomicity and not have an error, so, atomicity is definitely desirable. And client-centric, if you have atomicity, then you will not have any problem. So, that is why in the CAP theorem they have taken atomicity as the definition of consistency, by and large.

(Refer Slide Time: 31:46)



Read and Write Quorums



- A write is typically sent to a quorum (multiple servers)
  - Write quorum: $\boxed{N_W}$ servers
  - $N_w > \frac{N}{2}$ (the write reaches a majority of servers)
- We typically read from a read quorum ($\boxed{N_R}$ servers) and choose the most recent value
- To guarantee that the most recent value is read: $\boxed{N_R + N_W > N}$

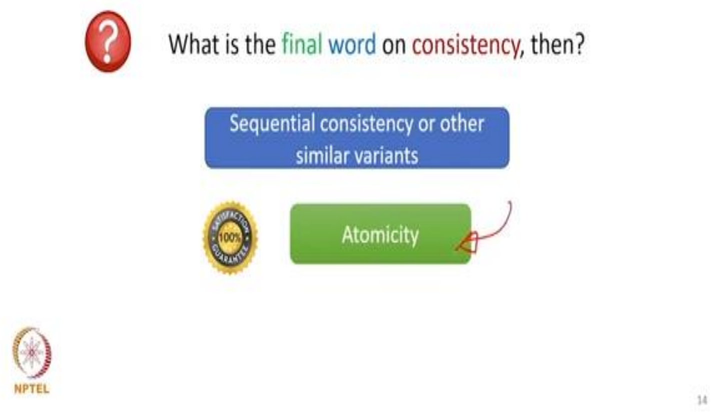
So then, after we have discussed the consistency models, we should cap this discussion with a discussion on quorum. So what is a quorum? A quorum, let us say quorum in a meeting is what? So quorum is basically the number of the minimum number of people who need to be present in a meeting to make a decision. So, we can do the same if you have a bunch of servers, what we can do is that we can have a write quorum of N_W servers. So, the write quorum is more than a majority so, you will know that at least a majority have the latest update.

And so basically, number one that insulates you against failures and we will link this with a read quorum, where a read quorum is a set of N_R servers that choose the most recent value. So, to guarantee that the most recent value is read, the only condition that needs to hold is $N_R + N_W > N$ which basically will then ensure that the read quorum and the write quorum have an overlap. And the overlap will provide the latest value of the write.

So, of course, the implicit assumption in the definition of a quorum and these quorum based writes is that when you write you actually broadcast it to the entire quorum all of them update it, when you read, you actually read from the entire quorum and every write is associated with a timestamp, where higher the timestamp no more recent write is and the value that is ultimately returned is the latest value or the value with the latest and greatest timestamp. So, this does increase the overhead within the server form because write now has to be broadcasted, or multicast it to the quorum.

And a read is also required from everybody from the read quorum such that you can get the value of the latest write, but this is a price to pay if we are using a set of servers for both performance as well as redundancy. And of course, if we have more servers it will make the data more available but ensuring consistency becomes difficult. So, you can clearly see a consistency and availability trade-off over here.

(Refer Slide Time: 34:08)



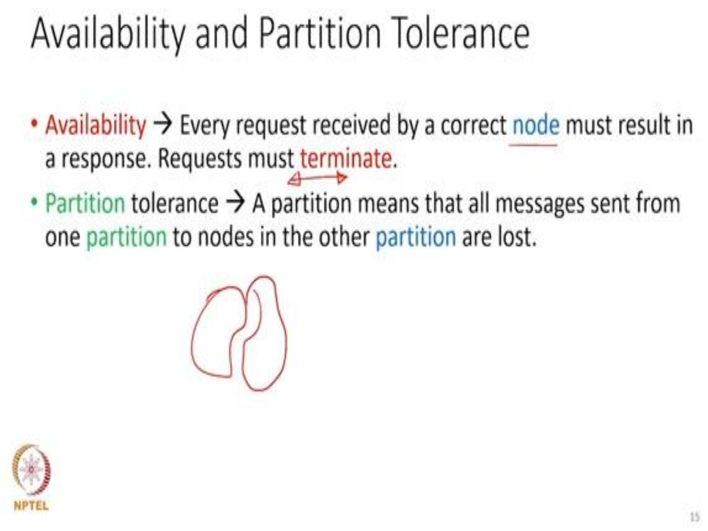
What is the final word on consistency, then?

Sequential consistency or other similar variants

Atomicity

So, the CAP theorem is all about trade-offs, so, but before that, what is the final word on consistency here? It is that sequential consistency or other similar variants and client-centric models everybody rely on atomicity. So, we will also primarily focus on atomicity as they see in consistency.

(Refer Slide Time: 34:30)

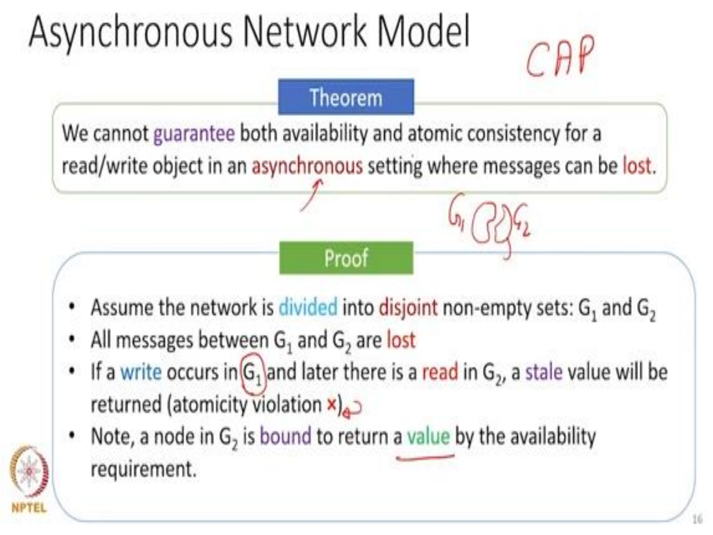


Availability and Partition Tolerance

- Availability → Every request received by a correct node must result in a response. Requests must terminate.
- Partition tolerance → A partition means that all messages sent from one partition to nodes in the other partition are lost.

The rest two are easy, so, rest we have discussed that is not a big deal. The first is availability, features that every request received by a correct node must result in a response, which means requests must terminate. So, writes are requested termination is guaranteed in the sense that every request has to result in a response. The second is partition tolerance, where a partition means that all messages sent from one partition to nodes in another partition are lost. And in spite of that the system needs to work, as well and as nicely and as correctly as possible.

(Refer Slide Time: 35:12)



Asynchronous Network Model

CAP

Theorem

We cannot guarantee both availability and atomic consistency for a read/write object in an asynchronous setting where messages can be lost.

Proof

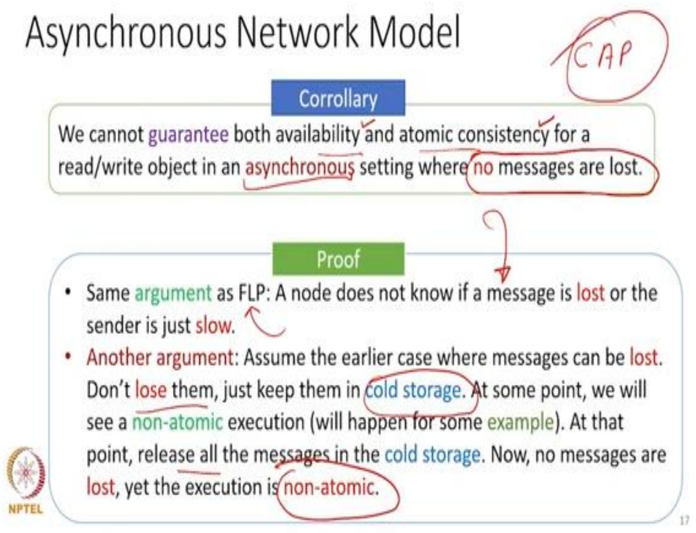
- Assume the network is divided into disjoint non-empty sets: G_1 and G_2
- All messages between G_1 and G_2 are lost
- If a write occurs in G_1 and later there is a read in G_2, a stale value will be returned (atomicity violation ×)
- Note, a node in G_2 is bound to return a value by the availability requirement.

So, what is the theorem? The key theorem is we cannot guarantee both availability as well as atomic consistency for a read-write object. In an asynchronous setting, where messages can be lost, in any asynchronous setting, where messages can be lost, or a partition can be created, we cannot guarantee availability and atomic consistence okay, why not? So, assume that you know, messages are lost in a partition is created, we call one partition G_1 and the other G_2. So, all messages between G_1 and G_2 are lost.

So, if a write occurs in G_1 and later, for the same variable, there is a read in G_2, clearly a stale value will be returned, because you are guaranteeing availability, you do not have the option of sitting on the request, you have to return whatever you have and what you have will be a stale value because you never got a chance to get the most up to date value. So, this would clearly violate atomicity, because it does not appear that the write occurred at an instant, because the write occurred and much later, you made a read from G_2 and what you got was a stale value.

So, the issue is that in this case, you cannot guarantee both availability as well as atomic consistency, and the reason being that it is not possible to propagate the write to G_2, and that is why a write will not appear to complete in an instant. And given the fact that nodes in G_2 are bound to return a value by the availability requirement, they have to return a stale value, which is going to break atomicity. So, what we can see is ensuring cap consistency, availability and partition tolerance. In this setting where messages can be lost and we have an asynchronous setting is not possible.

(Refer Slide Time: 37:16)



Asynchronous Network Model

Corrollary

We cannot guarantee both availability and atomic consistency for a read/write object in an asynchronous setting where no messages are lost.

Proof

- Same argument as FLP: A node does not know if a message is lost or the sender is just slow.
- Another argument: Assume the earlier case where messages can be lost. Don't lose them, just keep them in cold storage. At some point, we will see a non-atomic execution (will happen for some example). At that point, release all the messages in the cold storage. Now, no messages are lost, yet the execution is non-atomic.

Let us have a corollary. So, we cannot again guarantee both availability and atomic consistency for a read-write object in an asynchronous setting where no messages are lost, so this is much stronger, this says that even if no messages are lost, we still cannot guarantee both of them. The argument is the same as the FLP theorem. Given the fact that we have an asynchronous setting, a node does not know if a message is lost, or the sender is just slow. So, this was the key aspect of the FLP result.

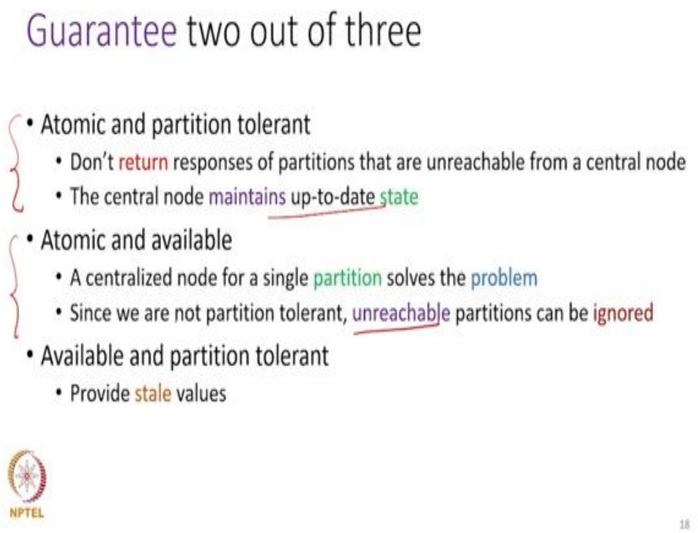
So, if you go back to that, where we argued that in an asynchronous setting, you simply do not know, if the node is still replying, or if a message is genuinely lost, or the node has developed a fault the node is dead, it is not possible to find that, given the pause, it is not possible to find it. This situation is no different from the earlier situation where messages are lost, because you cannot find out anyway, whether messages are lost or nothing is lost, given the fact that it is not possible to find out. This situation is exactly similar to the previous one. In so far, as the perspective of the node is concerned and in the previous one, we could not guarantee availability and atomic consistency, so, we cannot guarantee it over here as well. Another argument, assume the earlier case where messages can be lost.

No problem, so, do one thing, do not lose them whichever messages you intend to lose, so, we are looking at a master oracle which has control over the network just keep them in a cold storage, do not deliver them keep them in a cold storage, so nodes will perceive that these messages are lost, but they are actually not there in a cold storage. At some point from the proof we will see a non-atomic execution. For some example, it will we will see a break in atomicity, at that point release all the messages in the cold storage.

So, as far as we are concerned at this particular point, when all the messages have been released and also delivered, no messages are lost because the cold store is empty and every message is accounted for but we have still seen an atom non-atomic execution. So, no messages are lost yet the execution has still been seen to be non-atomic. So, this means that even if there is no message loss, we can still construct an example.

Fair if availability is guaranteed atomicity has to be compromised, so, there is a trade-off between them or rather both cannot be guaranteed. So, which essentially means in asynchronous setting regardless of the reliability of the network, so, whether your partition tolerant or not availability and atomic consistency both cannot be guaranteed or if we extend this argument in the CAP theorem all three cannot be guaranteed.

(Refer Slide Time: 40:24)



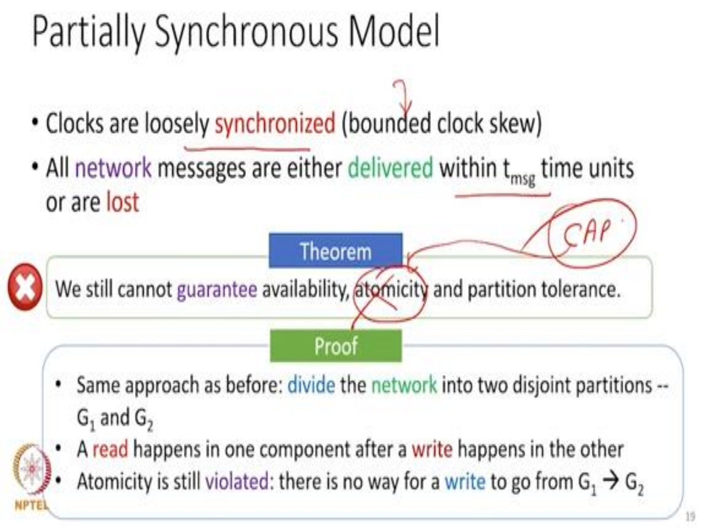
Guarantee two out of three

- Atomic and partition tolerant
  - Don't return responses of partitions that are unreachable from a central node
  - The central node maintains up-to-date state
- Atomic and available
  - A centralized node for a single partition solves the problem
  - Since we are not partition tolerant, unreachable partitions can be ignored
- Available and partition tolerant
  - Provide stale values

Can we guarantee two out of three? Well we very easily can. Atomic and partition tolerant. Well have a central node and ensure that all the updates are there with the central node it maintains up to date state if you can reach the central node return a response which is guaranteed to be correct in the sense that will be atomic. If you cannot reach the central node from a partition, well do not return anything given the fact that availability is not required, we are fine. So, basically if the option is there or not returning anything, then of course, correctness will always be guaranteed because we will never return something which is false, which is wrong.

Atomic and available. Well, same idea have a centralized node for a single partition that will solve the problem you will both be atomic and available do not care about the rest of the partitions because we do not have to be partition tolerant. So, other unreachable partitions can be ignored. Available and partition tolerant. Well, in this case there is no consistency at all. So, for that matter any value can be provided any stale value can be provided given the fact that atomicity is not being considered here there is no problem.

(Refer Slide Time: 41:48)



Partially Synchronous Model

- Clocks are loosely synchronized (bounded clock skew)
- All network messages are either delivered within t_{msg} time units or are lost

Theorem

We still cannot guarantee availability, atomicity and partition tolerance.

Proof

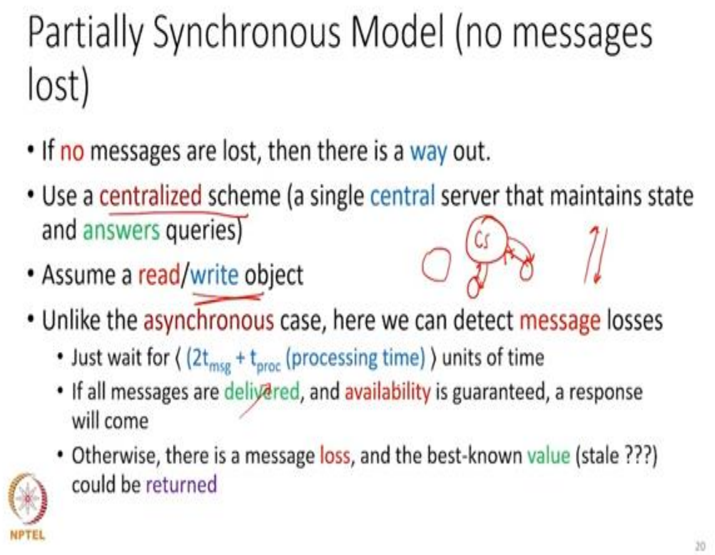
- Same approach as before: divide the network into two disjoint partitions -- G_1 and G_2
- A read happens in one component after a write happens in the other
- Atomicity is still violated: there is no way for a write to go from $G_1 \rightarrow G_2$

Now, let us look at a different model. So, we have been looking at an asynchronous model up till now, let us look at a partially synchronous model, which we will also discuss in other papers like stellar and so on, where clocks are loosely synchronized, there is a bounded clock skew, all network messages are either delivered within T message time units or they are lost. So, the network is also reliable, clocks are also reliable.

So, the theorem is we still cannot guarantee CAP, so, the all three cannot still be guaranteed. The idea is the same; we divide the network into two disjoint partitions G_1 and G_2. A read will happen in one component or write in the other as we have seen. Given the fact that the right cannot propagate to the other component, let us say from G_1 to G_2 the write cannot go, we will never be able to provide the up to date values atomicity will not hold. If atomicity one does not hold you the C does not hold over here, then CAP cannot be guaranteed.

So, in this case, regardless of whether you are asynchronous or partially synchronous, we still cannot guarantee all three. If you are let us say, even if the clocks are synchronized and the network is reliable to the extent that it tells you within T message time units either delivers a message or loses it forever, so, there is no cold storage here. Or it will be incorrect to say no cold storage, or rather; we cannot use the cold storage theoretical trick to prove something in this case.

(Refer Slide Time: 43:21)



Partially Synchronous Model (no messages lost)

- If no messages are lost, then there is a way out.
- Use a centralized scheme (a single central server that maintains state and answers queries)
- Assume a read/write object
- Unlike the asynchronous case, here we can detect message losses
  - Just wait for ⟨ ($2t_{msg} + t_{proc}$ (processing time) ⟩ units of time
  - If all messages are delivered, and availability is guaranteed, a response will come
  - Otherwise, there is a message loss, and the best-known value (stale ???) could be returned

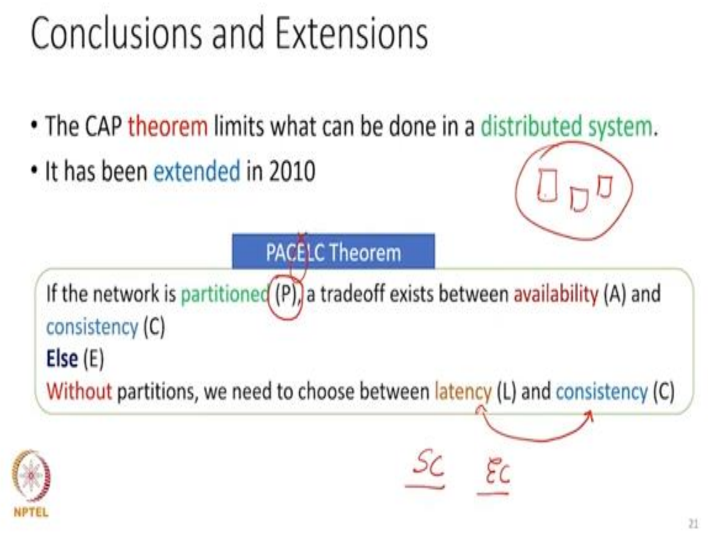
Now, let us look at the corollary in terms of the partially synchronous model. Assume that no messages are lost. If no messages are lost, is there a way out? Well, it turns out to be yes, use a centralized scheme, which is a single central server that maintains the overall state and it also answers queries and consider a basic read-write object. So, a single central server here multiple client machines, all of them issue read-write request to the central server and then, return the result.

So, unlike the asynchronous case, here, we can detect message losses in the sense if I send a message and expect an acknowledgment and a wait for $2t_{msg}$ + t_{proc} to process my original message. I will get to know some things, I will get to know whether you know the message was sent and act was received otherwise I can again send the message again. So, let us see if all messages are delivered, if all messages are delivered, I will always get the act back.

So, let us not consider the case where either the message is lost or the act is lost. But let us consider cases where all the messages are delivered in that case and get acknowledgment back there is no message loss. If that is the case and availability is guaranteed a response will come. So, in this case, the response will come, and the response that will come will be correct.

So, we will know if there is no message lost, we will definitely expect a response within a finite amount of time and if you still use the central server base scheme, this is going to work. Otherwise, a best-known value the stale, best known stale value could be returned. But again, in this case, if there is no message loss, then this is a different case, because in this case, message loss can be detected. And given the fact that it is a different case, and if we have availability responses will come and the responses will also have the correct value.

(Refer Slide Time: 45:53)



Conclusions and Extensions

- The CAP theorem limits what can be done in a distributed system.
- It has been extended in 2010

PACELC Theorem

If the network is partitioned (P), a tradeoff exists between availability (A) and consistency (C)
Else (E)
Without partitions, we need to choose between latency (L) and consistency (C)

SC
EC
NPTEL
21

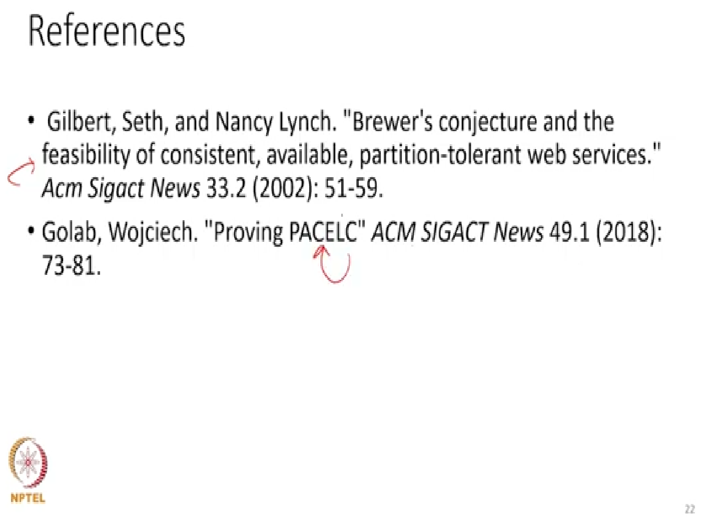
So, conclusions and an extension, so, the CAP theorem limits what can be done in a distributed system, it really cannot have C, A and P all together. So, this has been extended in 2010 to the PACELC theorem. So, this actually says something more, so, it says that look at the network is partitioned. A trade-off exists between availability and consistency and this we have seen, this we saw in our discussion, that even if a network is partitioned, then of course, there is a trade-off.

If it is not partitioned, so, if let us say that, all nodes are reachable without partitions, then there is a trade-off between else so, this E is for else, then there is a trade-off between latency and consistence, you can always have a system with no consistency that is going to be very fast. For example, if you have a set of servers, and instead of all of them maintaining a replica, you could just say that, you know your query some server and return a stale value then the latency will be very low. And the consistency will also be quite weak.

But if I made the consistency quite strong in the sense, if I make this linearizable then of course a lot of messages etcetera, have to be sent between the servers to ensure that we are genuinely getting the latest value. So, in that case, the latency will shoot up, so, basically what is required in a distributed system, it could be SC, so in SC you will get a solid correctness model.

We have not discussed eventual consistency, but there is not a bad place to discuss it. So, eventual consistency means that writes are ultimately visible in the sense that if I am writing to something, I do not care about the order of writes and other aspects but writes are ultimately visible. So, you have an eventually consistent system where rights are ultimately propagated by latency could be quite fast. So, as you can see, there is a trade-off between L and C particularly when we do not have partitions. So, this is a PACELC theorem.

(Refer Slide Time: 48:06)



References

- Gilbert, Seth, and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services." *Acm Sigact News* 33.2 (2002): 51-59.
- Golab, Wojciech. "Proving PACELC" *ACM SIGACT News* 49.1 (2018): 73-81.

22

And this also was formally proven as you can see, so, the first paper by Seth and Lynch was the one that proved the CAP theorem and the PACELC theorem the one that you see over here was proven actually in 2018 and you can take a look at the proof.