

Storage Systems
Dr. K. Gopinath
Department of Computer Science and Engineering
Indian Institute of Science, Bangalore

Design Factors

Lecture - 31



Design of Large-Scale distributed Storage Systems _Part 2: How current Storage APIs like POSIX/NFS/S3/Zookeeper handle

Locking/Synchronization/Commit/Consensus Problems, ACID Vs. BASE Models, Commit & Consensus Problems defined, Relation between Atomic Commit & Consensus

(Refer Slide Time: 00:14)

Lecture Outline

in the previous lecture, we learnt that a Distributed Storage System may experience many types of failure, and, moreover a failure state is indistinguishable from a dead state. We also saw some impossibility results in the last lecture, viz., that in a general distributed system prone to failures, Distributed Consensus, Distributed Locking, Distributed Synchronization, Distributed Commit are impossible to achieve. However, since all real distributed storage systems are prone to some type of failure, the question arises as to how they handle the failures and what guarantees they provide in the event of failure. The first part of this lecture focuses on this topic. We study the APIs provided by some current storage filesystems like POSIX, NFS, Amazon S3 and Zookeeper to get an idea of aspects like the following: what reliability guarantees are provided by the API models? do they provide weak or strong guarantee? considering a write operation, what happens if there is a failure in the middle of the operation? in a write operation, are we always sure that what was eventually written to a device is the same as what we intended to write? If an O/P device returns success after a write, can we be sure the write really happened? what data structures are used by the models, what is the mechanics of operation of a given API? Next, we look at Consistency models of S3 and Zookeeper. S3 is based on an Eventual Consistency model which guarantees that all updates will be consistent after a sufficiently long period of time, i.e. the updated replicas are Basically Available, Soft-state and Eventually Consistent - this is called BASE. Zookeeper is based on a Time bound Consistency model which guarantees Sequential Consistency, Atomicity, Single System Image, Reliability and Timeliness. Next we look at ACID vs BASE models, which are models that define the properties that guarantee reliable processing of transactions/operations. Every domain - Realtime systems, Filesystems, Database systems etc. - has its own particularities and requirements, and, hence may make use of either ACID or BASE model depending on what is more important for it - strong Consistency, high Availability, Stale data is OK etc. The above topics form the first part of the lecture. The second part of the lecture is about the important abstract problems related to designing a Distributed System for Consistency viz., Commit problem, Consensus problem, Commit Protocols and Atomic Commit. The (informal) problem definition, axioms & conditions and validity checking are discussed, and, the relation (& differences) between the Atomic Commit and Consensus problems are highlighted in the lecture. The Professor concludes by saying that the Commit/Consensus problems are important for various parties & communities - Storage Systems community, Database Systems community, Web community; but each may employ a slightly different model (Stronger/Weaker Consistency/Availability).




Welcome again to the NPTEL course on storage systems. In the previous class, we were trying to understand why it is not easy to get the kind of consistency in storage systems, basically because we have to solve some coordination problems and, in this coordination, problems have inherent difficulty in the context of failures. So, we looked at some of the issues for example, we briefly looked at the 2 general problems which is looked at the

(Refer Slide Time: 00:58)

Why are failures difficult in asynch env?

- (simpler) The 2 generals coord problem: need to coord to defeat enemy in between (who can seize, dupl, corrupt any msg sent)
 - *No protocol exists!*
- The FLP (Fischer, Lynch, Patterson) result: impossibility of distributed consensus with 1 faulty processor (JACM'85)
- Fekete, Lynch, Mansour, Spinelli: impossibility of reliable communication in the face of crashes (JACM'93)
- No reliable data link layer can exist in CAML model (JACM'00)
 - *crashes, asynchronous, memoryless, lossy model*

System can be driven by a sequence of crashes to any global state where each node is in a state reached in some (possibly diff) run, and each link has an arbitrary mixture of packets sent in (possibly diff) runs (Jayaram/Varghese JACM'00)





Kind of problems that will be there with. I just mentioned something about the FLP result and then the fact that you cannot even have a reliable communication in the face of crashes. So, various interesting results are there which basically show

(Refer Slide Time: 01:14)

Brewer's CAP Theorem

- Three important properties
 - Consistency
 - Availability
 - tolerance to Partitions due to breakdowns in communications in the system

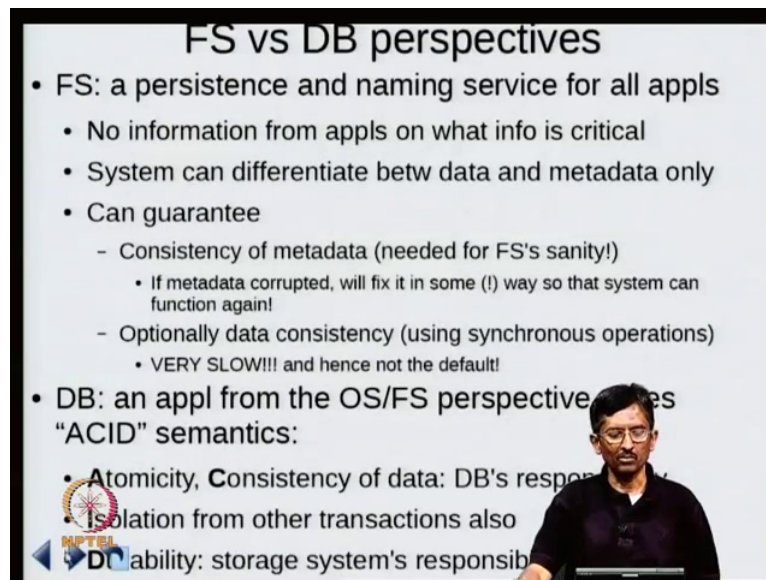
cannot all be guaranteed at the same time
according to a theorem in distributed systems theory (proved by Gilbert & Lynch '02)



That there are some problems. We also looked at briefly at the cap theorem which basically tells you that certain things are not possible that is,

For example, you cannot have consistency availability and tolerance to partitions at the same time.

(Refer Slide Time: 01:29)



FS vs DB perspectives

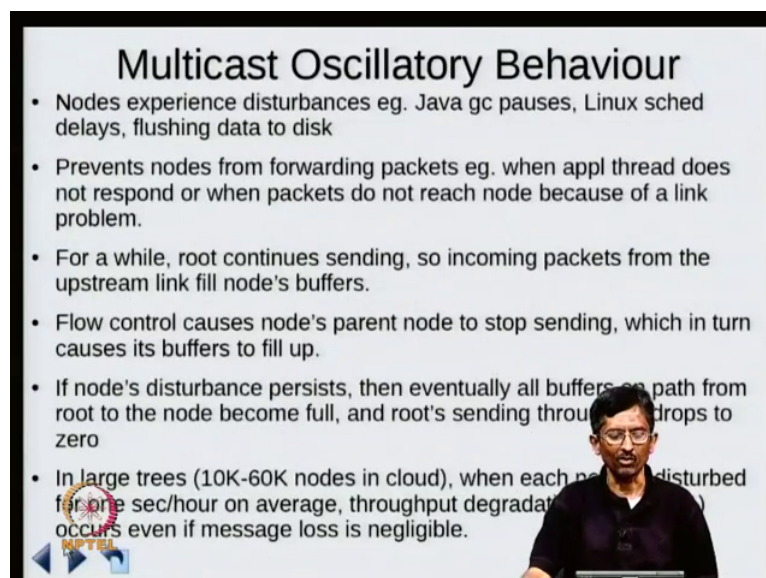
- FS: a persistence and naming service for all appls
 - No information from appls on what info is critical
 - System can differentiate betw data and metadata only
 - Can guarantee
 - Consistency of metadata (needed for FS's sanity!)
 - If metadata corrupted, will fix it in some (!) way so that system can function again!
 - Optionally data consistency (using synchronous operations)
 - VERY SLOW!!! and hence not the default!
- DB: an appl from the OS/FS perspective
 - "ACID" semantics:
 - Atomicity, Consistency of data: DB's responsibility
 - Isolation from other transactions also
 - Durability: storage system's responsibility

NPTEL

So, I also briefly mentioned about the kind of differences that exists between file systems and databases, because files systems usually do not have application level information. So, they can only attempt to be consistent at their own in their own data structures and their database typically has much more information application.

Therefore, they can do something better. So, I think we also try to briefly look at some issues about why in practice also this is very difficult we gave you some I gave some examples of some interesting problems that we come across in real life.

(Refer Slide Time: 02:09)



Multicast Oscillatory Behaviour

- Nodes experience disturbances eg. Java gc pauses, Linux sched delays, flushing data to disk
- Prevents nodes from forwarding packets eg. when appl thread does not respond or when packets do not reach node because of a link problem.
- For a while, root continues sending, so incoming packets from the upstream link fill node's buffers.
- Flow control causes node's parent node to stop sending, which in turn causes its buffers to fill up.
- If node's disturbance persists, then eventually all buffers on path from root to the node become full, and root's sending throughput drops to zero
- In large trees (10K-60K nodes in cloud), when each node is disturbed for one sec/hour on average, throughput degradation occurs even if message loss is negligible.

NPTEL

So, basically the issue is that detecting when some system is dead or alive itself is a difficult problem. So, it might be that the system is extremely slow and therefore, it is almost indistinguishable from a failure.

The question is do you wait for that system to show signs of life or you decide that it is not possible and move on. Now this is a very tricky issue to be handled depending on how you handle this issue different kinds of models is possible. For example, if your system varies about consistency very carefully because it involves lot of real world implications. For examples it involves money etcetera, then you may want to wait for the system.


To correct itself there are some of the situations where either it is not important therefore, it is to proceed, or you are under a real time pressure you have do something right, you do not have all the information, but you still have to do something right for example, you have an aeroplane tracking system, air controller kind of systems if some part of the system has failed it does not stop moving from having to act do some acts. So, that aeroplane still take off and land smoothly.

Even if there are failure we still have to proceed somehow. So, because of this different kinds

(Refer Slide Time: 04:01)

FLP/CAP Related Problems

- **Distributed Consensus: FLP Impossibility result**
 - Distributed Locking/Synchronization
 - Distr Commit in clustered/distr fs and db
 - Slightly similar: Waitfree synchronization
- **Let us consider the state of art in current large scale storage systems:**
 - Distr locking, synch, commit problems exist
 - How are they being handled? What guarantees are being given?



Of situation we will find different kinds of models right. So, we will try to look at some of these issues in some detail. And so, today I will talk about the straightly more what I say detailed way with respect to these problems. First thing I think as we mentioned before

distributed consensus is not possible. That is the basically there is a fisher (Refer Time: 04:33) patters and result and basically shows that distributed consensus is not possible even if one faulty processor. And therefore, if distributed consensus is not possible, distributed locking also is not possible, because what is locking basically it is a you have k number of parties who would one of them has to doable to enter the critical section or the area which is a shared region which has to be modified.

So, distributed consensus is not possible; that means, there is no way to say where one party can be in that shaded region is not possible. Similarly, distributed synchronization is also not possible in many cases what happens is that multiple parties have to step through a certain steps they have to go through several steps in a particular order you want to synchronize there the way they execute.

So, distributed synchronization may also be not possible because we can always reduce it to the problem of consensus, that is I want to know whether I am in stage 5 or stage 8 across all the parties were involved. So, this also is considered to be impossible. So, by the same reason it turns out that if I am trying to do consensus sorry consistency, which is the issue for clustered or distributed file system and databases.

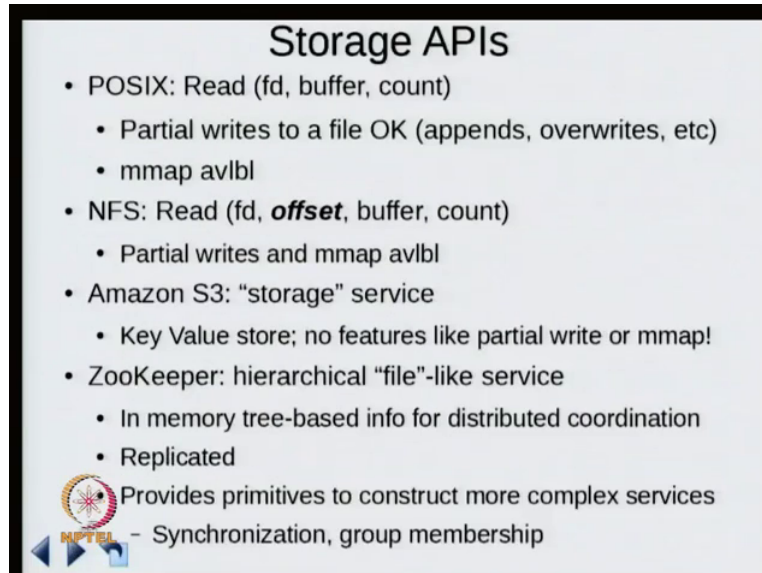
They essentially have to agree on whose value has to be used to update the most recent version of the of the some piece of data; that means, they all have to agree therefore, in some sense the commit problem also is connected with this issue. There are some interesting connections with some other types of work what is called wait free synchronization, where the idea is to synchronize and, in a way, which does not prevent you from making progressive and in spite of failures. They are somewhat similar actually this result also has some bearing on this particular area.

But we will not get into this one. So, what we are what I am trying to say is that once we have this FLP impossible result there are lot of things which get affected by this particular result. So, you do have to look at engineering solutions which actually take care of this problem because in theory it is impossible, but in practice we have to find somewhere to solve this problem. It will be certainly fail in some situation not only engineering due to.

But that is the fact of life we have to live with it. Now just look at you just quickly look at the kinds of systems we might come across and what kind a how they what kind of guarantees they give for example, it might be distributed locking or synchronization that is stepping

through pieces of the program or the computation or agreeing on a particular value to be committed etcetera. So, let us just look at how they are doing and why.


(Refer Slide Time: 07:41)



Storage APIs

- POSIX: Read (fd, buffer, count)
 - Partial writes to a file OK (appends, overwrites, etc)
 - mmap avlbl
- NFS: Read (fd, **offset**, buffer, count)
 - Partial writes and mmap avlbl
- Amazon S3: "storage" service
 - Key Value store; no features like partial write or mmap!
- ZooKeeper: hierarchical "file"-like service
 - In memory tree-based info for distributed coordination
 - Replicated

Provides primitives to construct more complex services
- Synchronization, group membership



So, before we proceed I will just go over the slide which we saw some time back. So, basically there are various types of models there is POSIX model the NFS kind of model that help model that 1980's people came up with the amazon model which is a more recent one.

And ZooKeeper also is some recent one, yeah, I am just giving you for examples, of the trends of properties they get POSIX has certain it tries to give you some interesting guarantees and of course, the guarantees are finally, based on what the devices can give it for example, in POSIX you can write something and unless you verify that what has been written is the same as what you wrote.

We are essentially trusting the device to have done it now this is next interval, but it turns out that devices do malfunction. You can ask it to write it on a particular track it is possible for you to write on some other track this what is called off track writes it is possible. It is also possible we are going to write it, it tells you that it has done written successfully, but actually it is not written it that also is possible it is going to happen because if you are talking about disk systems the disk work on the basis of what is called Bernoulli effect.

That is the disk heads are hovering over the right surface about some few microns above actually fractions of a micron above and if there is any let us say small dust particle etcetera

then essentially it goes over it. And we are writing of that point it may not succeed in writing with the magnetic strength that is required to write, because it is no longer at that same few fraction of micron height it may be at much higher one before the write might not be properly done that also is possible.

So, the only solution for if you want reliability in POSIX kind of models. After writing it you try to read it back if you read it back and if it same as what you then you know that it is bent up this is basically what is called write verify actually there is a scsi command which does this and this is very important because in some situations especially when it very cold it turns out that sometimes there are some problems with the disk and they can actually start malfunctioning, or if you end up going to some place like tibet where the air is quite thin it turns out disks not work there properly for the same reason.

So, if you are thinking that POSIX will give you some guarantee it is actually dependent on the kind of guarantees that storage the disk device is giving you. So, that is kind of one kind of device determinant kind of guarantees. We look at NFS this basically, because of reasons of throughput or latencies we do caching and because of the caching purposes it turns out that the value that if we look at the attributes it may have a stale value and we might still proceed because it has not timed out.

So, issues of that kind that are there at NFS. If you look at Amazon S3 the problems that are here are because of what you discussed the previous class, because of FLP or cap related cap basically you cannot have consistency, availability and resilience to partitions at the same time; that is because so this Amazon S3 can give you much weaker type of consistency model and that is basically reason why you have to be you have to be careful about the model that they provide.

Similarly, this is ZooKeeper which is hierarchical file like service and this one also has a slight different model we will actually take quick look at exactly what it does. So, what is S3? S3 is basically a key value stored whereas, ZooKeeper is a if you are familiar with the proc file system in Linux it is something similar to that. So, the idea here is we want to provide certain information in a tree like fashion, and this is memory based and various clients can actually get information about some particular state of the system from this file like service.

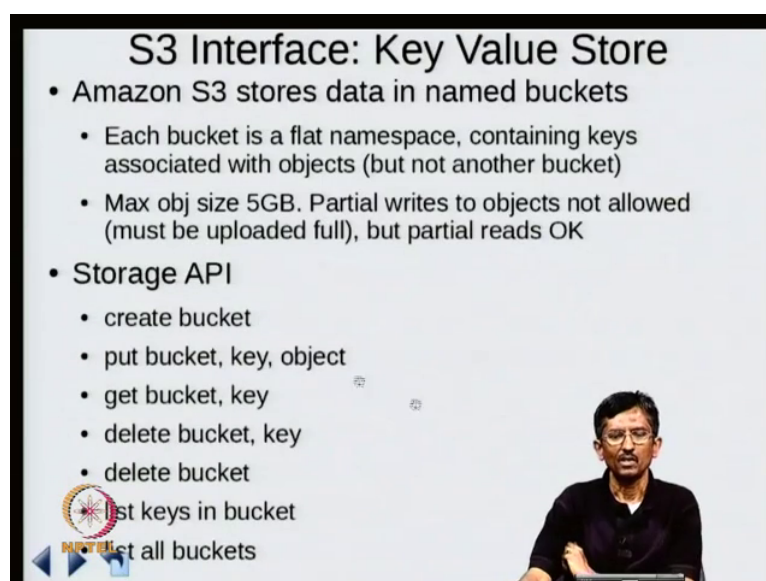
So, basically in some sense we want to coordinate between multiple users and they all put that information into a hierarchical kind of a model. And it is a memory-based system. It is

replicated and it provides primitives to construct more complex services synchronization I mentioned synchronization basically as I mentioned earlier. You in a distributed problem it is possible that you want to sequence multiple users through the same steps that is all of them have to have begin step one then only you can go to step 2 then all of them have to finish whatever they are doing in step 2 then go to step 3.

So, we want to synchronize those things or we can have issues like group membership often times when you are a trying to make some value the same in all places for example, consistency you might want to send messages across when you are sending messages across it is important that all the messages be received in all the places in the same order sometimes is very important. If things if there are 2 sets of messages one modifying a and one modifying b and some messages that go to certain clients of a that intersposts with messages of for updating b sometimes you can have very peculiar results.

So, and this becomes even more complicated when things fail. So, there are whole model called group communication systems which guarantee how messages are sent and the order in between messages as received by various clients. So, these are very tricky area and it turns out to have some connections with the problems we discussed before that is FLP result etcetera. So, this also turns out to be impossible if you make the most general assumptions, but engineering wise you can settle do something about this also.

(Refer Slide Time: 15:01)



S3 Interface: Key Value Store

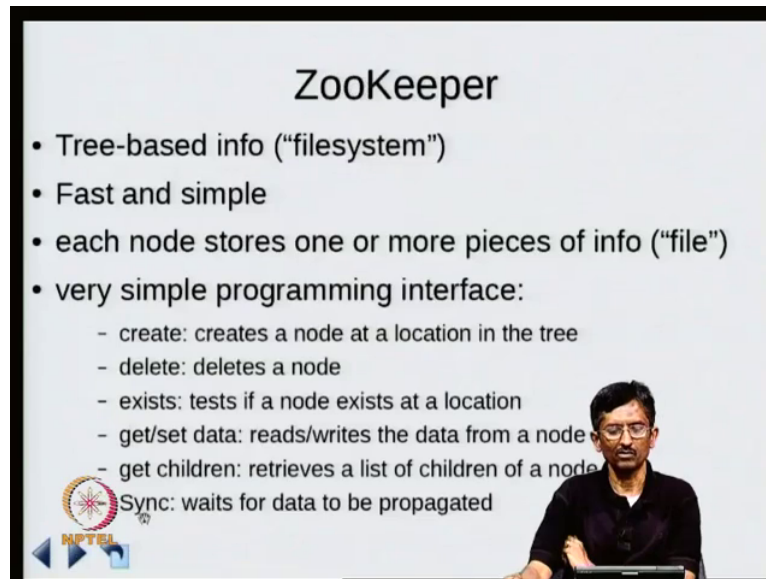
- Amazon S3 stores data in named buckets
 - Each bucket is a flat namespace, containing keys associated with objects (but not another bucket)
 - Max obj size 5GB. Partial writes to objects not allowed (must be uploaded full), but partial reads OK
- Storage API
 - create bucket
 - put bucket, key, object
 - get bucket, key
 - delete bucket, key
 - delete bucket
 - list keys in bucket
 - list all buckets

The slide features a presenter in the bottom right corner and a navigation bar at the bottom left with icons for back, forward, and search.

So, as we discussed before I am not going to go into this before, as we discussed it before this S3 basically stores the data ending buckets we have some API.

So, we have a bucket or something like a directory an object is accessed through a key. So, we have seen the this particular API before. So, this is one type of storage.

(Refer Slide Time: 15:26)



ZooKeeper

- Tree-based info ("filesystem")
- Fast and simple
- each node stores one or more pieces of info ("file")
- very simple programming interface:
 - create: creates a node at a location in the tree
 - delete: deletes a node
 - exists: tests if a node exists at a location
 - get/set data: reads/writes the data from a node
 - get children: retrieves a list of children of a node
 - Sync: waits for data to be propagated

The slide also features the NPTEL logo in the bottom left corner and a small inset image of a man with glasses speaking at a podium on the right side.

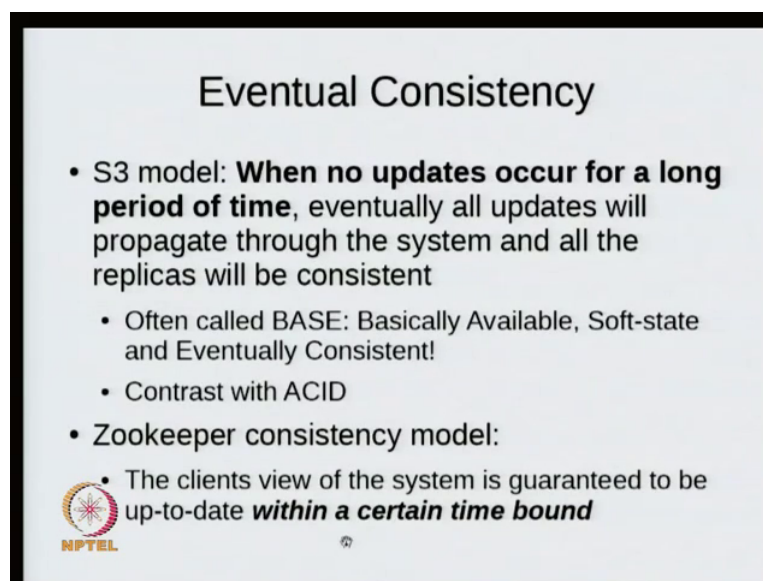
The ZooKeeper as I mentioned earlier is a tree-based information model. It is it is like a pseudo file system just like slash proc in Linux. It is fast and simple it is based on memory and each node stores one or more piece of information for example, just like in Linux if you go to slash proc CPU info it gives you some information about the CPU. So, this is also ZooKeeper also does the same thing except that this is a coordination device. It is basically used so that, multiple parties can pick up information and update information and ZooKeeper does that.

We can thing is to make sure it is highly available. So, it has to be repeated and if multiple parties tried to attempt to update the same to same at the same time then there has to be ordering happens. So, this has a very simple programming interface you can create a node at the location at tree. For example, if you take again the Linux model then it has some information called slash proc slash sub x I can create a node which says slash proc x slash y for example, I can delete a node. I can check if a particular node exists at a location. I can get the value or set value. I can figure out all the things that are there at for example, slash proc x

that there is something called y z etcetera and or I can I also have a model say sync what does it mean it means that; some changes are taken place I would like to wait till it all stabilizes


So, essentially it is like the model could be something similar to what you have in regular file systems there you can do what is called f sync and idea is that you want to ensure that the values get updated on the disk. So, there is a similar procedure that provided to make sure that we can wait for till it propagated and you will get signal saying that it is done then only you can proceed.

(Refer Slide Time: 17:25)



Eventual Consistency

- S3 model: **When no updates occur for a long period of time**, eventually all updates will propagate through the system and all the replicas will be consistent
 - Often called BASE: Basically Available, Soft-state and Eventually Consistent!
 - Contrast with ACID
- Zookeeper consistency model:
 - The clients view of the system is guaranteed to be up-to-date ***within a certain time bound***

 NPTEL

Now what are the things that they provide? S3 actually has some model of called eventual consistency. What does it stand for?

It stands for, when no updates occur for a long period of time eventually all updates will propagate through system and all the replicas will be consistent. What is that? Listen to that how long it is, if they say is long enough. Now this this of course, is a problematic thing. And so, essentially it is very closely connected with the kind of system you are, thinking about it could be wide area network system it could be local area network and this is long period of time depends on that particular aspect. Actually, ideally it should be this long period of time should be adaptive it should be based on a current load on the system.

If the load is very high probably should be longer time, if the load is very light it should be smaller number of time. If you look at the time disynchronous model that I talked about

earlier it actually attempts to be adaptive, but eventual consistency only gives you this model. No updates occur for a long period of time. Then everything will be consistent. This has been often called BASE, it is transfer basically available it is a made up acronyms as per I know it is stand for basically available soft state and eventually consistent it is a long name some cooked up name and there is reason why it is called basis, because there is a different model of ACID we looked we talk we talked about earlier which is used in databases and storage systems and architecting systems and this is basically ACID.

So, what is I will come to this earlier we discussed a bit about this we come to we will talk a bit more about it. So, basically because this model is ACID this model called is called base and this cooked up name called. So, in ACID what is the issue here? We are talking about multiple transactions and we want to ensure that either the transactions go through or the transactions does not go through. Let us look at the one example suppose I want to do I want to go someplace, I want get the railway ticket booking plus I want to get a hotel booking and probably it a ticket for some event that I am interested in.

So, I am interested in doing it only if I get the railway ticket slash the hotel plus the ticket. So, I am talking of the transactions in which you try to do all of them if any all of them does not work out them I am going back out of it. So, the issue that I am interested is either the transaction completes or does not complete I do not have intermediate state where I have a hotel booking, but I do not have a train booking slash the event booking that does not work out. So, I want it to be either all of them to happen or not going to happen.

So, some updates are making to some system all updates should have to happen or none of them should happen that is basically atomicity that is at this model atomicity is somewhat different from what different operating systems good quality in atomicity atmospheric temperature in general we understand it includes there are 2 operations a and b nobody comes in between that is usually the model that is what they usually call as atomicity whereas, databases is a slight different model because we are talking about applications here.

So, here what they are interested is not the fact that somebody has come in between what they want to make sure is that either I did something or it is the same steps before that is what atomicity is. Consistency what is consistency here? It basically says that there is some model of how data in my system there is some invariant that has to be satisfied that invariant always has to be satisfied.

So, whatever any transaction does it has to keep the invariant intact if you are not able to keep the invariant intact then you are not consistent. Now again if we talk about a file system nobody has told it how you should touch it is data how you should handle it is data there is no model therefore, file system does not have a model of consistency, whereas a database has a model of consistency because there is some schema and there is various other things right which tells you how what has to be done. So, the database has a consistency model.

So, it is not really as a systems person suppose we build in a system this C is actually not part of a system definition that is why a file system does not worry about this at all. The only thing it worries about is its own model of it is metadata it has to be consistent what is internal to it that is how it bothers about. What is I? I is isolation. Isolation basically means that if you have multiple concurrent transactions it should be even if you are executing it concurrently even if each of the transactions are reading and writing things concurrently, whatever be the effect of execution of all these concurrent transactions it should be equivalent to some serial execution that is what isolation is; that means, that each transaction it executes as if it was isolated from other transactions, it should have that the equivalent behavior that what isolation is and this is required also for file systems and database all these things are required for this.

So, the D part of it is the durability part. So, D what does it mean it means that if I write something it should in spite of failures crashes panics in the system whatever right? When the system comes up again whatever a rate before the system went down it should persist. So, the D is the durable part of it of course, the D is the main concern in storage systems. So, this is an important thing to be looked at. So, essentially when we look at some of the systems database special database systems specially they have ACID model, but this ACID model is too sometimes extreme or too difficult to handle and therefore, it is a weaker model.

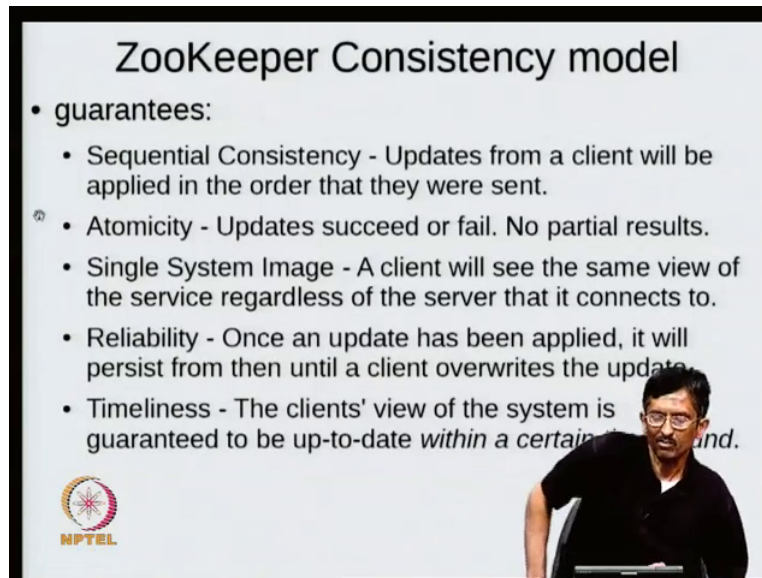
As I mentioned earlier there are real time systems where ACID does not make any sense, because as I mentioned earlier your air traffic controller systems where if some part of the system dies you cannot just say that I am going to ensure that the system is completely consistent across all the parties. I need to take some real time actions and I will tolerate some of it is failures and still proceed or do basically what might be called as best effort delivery of the service that I have that I can do.

So, ACID model is not meaningful in these circumstances or it circum circumstances in which a consistency notion is not itself clear to the party who is supposed to provide a model. As I mentioned earlier a file system has no idea about consistency of the data nobody has told it. So, there is no way it can be ACID the way it is mentioned here, but it can be ACID with its own data structures, because it knows something about its own metadata and it can do all these things. So, in the sense you might see the file systems actually implements an equivalent ACID concept for its own data, but not for the application database that is the difference between a file system and database.

Now, this ACID has originally was for example, proposed by Lamson and it was picked up by Grey et cetera (Refer Time: 25:33) they became famous there at source. So, the opposite class it is basically this. So, whereas, where you do not really guarantee a consistency until all the updates propagate through a system and then only you can say that replicas will be consistent. So, let us look at what ZooKeeper does ZooKeeper also says something similar to the clients where the system is guaranteed to be up to date within a certain time bound. So, for example, in the ZooKeeper model you might keep an information about which machines are up which machines are down. Now it is possible that within that time bound, if you wait long enough then it will all be consistent, but in the if it is much before that it may be that some machine is thought to be down and it is there and vice versa it is possible.

And this can create lots of complexities in dual systems and that is why as I mentioned earlier it is something called group committing systems which attends to solve this problem we will take a look at it because some storage systems use group committing systems as a way to make the programming job in the kernel easier, because when you run systems in the in the certain parts of systems are in state number of state it is very difficult to program in that context. So, you need some kind of infrastructure which it shields you from this and it can be kernel infrastructure which can be used. So, that it can this is possible.

(Refer Slide Time: 27:10)



ZooKeeper Consistency model

- guarantees:
 - Sequential Consistency - Updates from a client will be applied in the order that they were sent.
 - Atomicity - Updates succeed or fail. No partial results.
 - Single System Image - A client will see the same view of the service regardless of the server that it connects to.
 - Reliability - Once an update has been applied, it will persist from then until a client overwrites the update.
 - Timeliness - The clients' view of the system is guaranteed to be up-to-date *within a certain bound*.

NPTEL

So, we just looked at in some detail with ZooKeeper consistency model. This ZooKeeper consistency model guarantees the following sequential consistency, updates from a client will be applied in order that they were sent it is not the cases that it will go in different orders. As I mentioned earlier web does not have this property that is why cricket scores, the cricket score may not be monotonic it will show 365 then it suddenly it shows 360 meter it is possible atomicity this is just what we discussed update succeed of fail no partial results.

Single system image the client will see the same view of the service regardless of the server that it connects to basically as I mentioned to you earlier ZooKeeper is replicated. So, that it can scale with respect to number of clients and also to make sure that if there is any failure it will crash of the systems some systems, then without losing all the information. So, it is important to the client will see the same view of the service regardless of the server that it connects to. Again, this is also guaranteed by group communication systems group communication systems they go through what is called views and they ensure of course, it cannot be an absolutely because I mentioned FLP results are prevented from it,

But they try to ensure as far as possible that each client has the same view of the system and all the messages that are sent and received they go in a particular order. And so, good communication system also do give you some they attempt to do the same thing reliability. Once an update has been applied it will persists from them until a client over accept it update; that means, what they are trying to say is that if you write something the bits will no rote it is

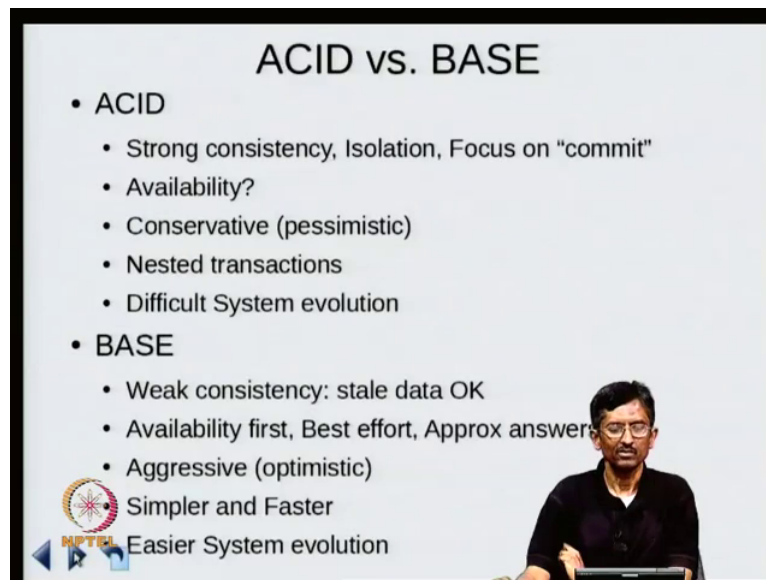
possible right for example, sometimes what happens is that on a disk you write something and you read it after about 3 years there is a small fewer chance that

It will develop a that is called latent sector here. I think this is happens for all the cd roms it happens also for floppy drives which probably nobody knows now not nowadays, but happens in all these situations in case of cd roms it turns out that it happens because it uses certain organic polymers and organic polymers degrade our product time. So, because the degrade of period of time. It can turn out that what has happened (Refer Time: 29:50) let us says once one or 0 can be indeterminate.

It cannot be indeterminate basically the CPU's are picked up when the river cd rom reader tries to read it does not give a clue a 0 or one it is possible. So, in the sense if you want the reliability; that means, that we have to be able to overcome all these kinds of issues; that means, somebody is actively scrubbing the information. What is scrubbing? It means that somebody is attempting to read it every. So now, and then. So, that this kind of a bit rot does not happen. Somebody has to reading it every. So, often and writing it to some other place. And so, keep the data alive in some sense.

The level which we are interested about consistence is in this part timely timeliness the client's view of the system is guaranteed to be up to date within a certain time bound. This is a the critical thing. So, again this is basically what is called eventual consistency model, eventual consistency model. Only big issue for us is the time bound is not mentioned it depends on different you have to know the system you are working with and come up with that particular time bound and then work with it there is no automatic (Refer Time: 31:06) figure on what it is.

(Refer Slide Time: 31:08)



ACID vs. BASE

- **ACID**
 - Strong consistency, Isolation, Focus on "commit"
 - Availability?
 - Conservative (pessimistic)
 - Nested transactions
 - Difficult System evolution
- **BASE**
 - Weak consistency: stale data OK
 - Availability first, Best effort, Approx answers
 - Aggressive (optimistic)
 - Simpler and Faster
 - Easier System evolution

So, let us just summarize again some of the discussion so far. So, there are 2 models the ACID versus base models.

As I mention to you earlier the reason why we have different models is because they have different domains in which we work it could be the real time systems domain it could be the file system domain database domain and our (Refer Time: 31:34) each of them they have their own particularities and issues they have to solve. So, they will do it in different ways

So, in ACID the issue is strong consistency isolation, but availability is not seriously looked at, because it can because availability has to be thrown out approxly because if you want to have strong consistency the (Refer Time: 31:59) results comes into comes into way therefore, we cannot have strong availability in consistency. It turns out to be conservative; that means, because of this it is not consistency it will block it will wait for some situations to become ordered before it will proceed. So, it is pessimistic it has seems the worst has taken place and it will keep waiting till things everything seems then we are going to proceed.

Whether it can also have nested transactions basically. So, the top-level transaction depends on what has happened to the, let us say sub transactions. And so, there is some issues of the correctness of it based on the correctness of the sub transactions. And typically, this basically typically this system evolution any system based on ACID model has a slighter difficult system evolution slightly because you have to specify about the system consistency model is.

So, once you specify the consistency what does it mean it means that you have to give it some new variants and they have to be carefully specified.

So, anytime I had something I will take out something you have to again revisit consistency. Is this I added this piece of stuff does it again is my invariant still valid or invalid I have to keep thinking about all these things and turns out to be non-trivial. So, whereas, in the case of base it is weak consistency it is to provide stale data once in a while, it is not catastrophic. Or it may be that for reasons of progress in the system we do not mind certain parties not responding in time we just proceed. So, the most important part is availability again as I mentioned earlier this is required for some critical systems which need to make progress. We have to make progress we have to just drop some issues consistency being one of them.

We always best effort and approximate answers are again you think about it if your car has crashed sorry if your car has stopped moving you would like to somehow get it going somewhere or other we do not really care about the most elegant way to do it or the most correct way to do it or if I am most way I am doing it just not keep it going till you get to some place where it can get to do it. So, same thing here. So, base is basically that model it is optimistic perceives that because you are not you are being weakly consistent there are certainly going to be problems, but it is optimistic sense says that you hope that somehow it can be pitched can fixed later it is not catastrophic.

Things divergent values some are able to reconstruct them later that is why it is optimistic. It is aggressive in terms of moving forward that is what it means. So, it has got an easier system evolution because it does not have various strong notions of invariant properties, and that is quite true for large systems. If I am talking about large systems there is no real nobody comes and says that this is the ingredient that has to be followed.

Suppose we take a look at the case of some major problems that we face in the world right now, right? There is question of some parties which want to blow off some places right there is no such says that you cannot that all buildings have to be for some systems right? So, there is no such invariant available that has to be satisfied.

So, therefore, it is possible because there is no invariant you can add things and remove things without worrying about (Refer Time: 31:36) invariants of course, it may not function the way you want it, but it allows you to make progress locally. So, it is may not be the best, but it always progress that is most important thing.

(Refer Slide Time: 35:48)

Commit Protocols

Abstract problem related to consistency: commit or consensus protocols

- Atomic Commitment (AC) and Consensus: both require fault tolerant agreement among processes

AC:

- AC1: No two processes reach different decisions.
- AC2: Commit is decided only if all votes are Yes.
- AC3: If there are no failures and all votes are Yes, then all processes decide to Commit.
- AC4: If all existing failures are repaired and no new failures occur for a sufficiently long period of time, then all processes will reach a decision.
- ~~No Blocking~~: All correct processes reach a decision:
Unrealizable! (General's paradox)

So, we will try to see this in connection with let us try to understand this problem with a bit more in some certain depth. So, we look at an abstract problem related to consistency this is basically the commit or consistence protocols. So, basically there is this problem called atomic commitment called let us call it AC and consensus I will come to the differences soon first let us look at atomic commitment.

So, what is it how is it defined? Basically, you have set of processes and they either have to vote yes or no and before they decide yes or no there could be in some undefined state, but they have to move to yes or no. Now what is the thing that we are thinking about they all agree on basically this is a simpler problem what we are saying is we are just talking about 2 decisions yes or no. You can from this you can actually construct other kinds of problems like for example, deciding on a particular value for example, should the value be 7,8 or 100 whatever and, but right now we are going to talk about binary it is yes or no.

So, what are the issues or what are the kinds of properties that you may want to have for atomic commitment. The first obvious thing is no 2 processes reach different decisions either everybody is agreed to yes or everybody will be against to no one of these things. So, this is fairly a straight forward commit is decided only if all votes are yes. So, this is one you might call accent commit is decided only if all votes are yes, what is it I am not saying commit is decided if and only if all votes are yes that is the reverse direction is not interesting or useful. If there are no failures and all votes are yes then all processes decide to commit.

Then there is another one AC 4 if all existing failures are repaired and no new failures occur for a sufficiently long period of time then all processes will reach a destination. We can see that all of these things are independent you can have any combination of these things. For example, what we are saying is that we are not saying if not what happens if some of the votes are not yes, we leave that decision open we do not say anything about that whereas, if it is yes; that means, everybody has agreed. So, other thing we are not saying here is

Or the process is correct, or the process is can fail in some strange ways what is called byzantine ways; that means, they are saying yes, but actually they mean no; etcetera, this can we are not talking about failures of that type. So, the only thing we are talking about at the best or if the word called.

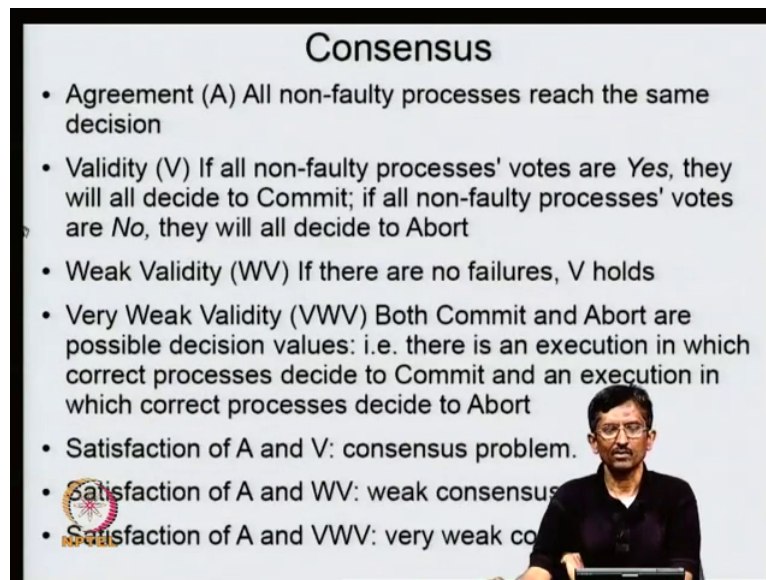
Student: staff.

Staff failures; that means, that if they fail they have accumulated some state which can be looked at then we come up again that is what we are talking about there is no byzantine failures are or what are called benign failures now ideally, I want this no blocking property all correct processes reach a decision. Now it turns out as we discussed earlier that 2 general paradox that problem it makes this particular problem unrealizable something that is the one cannot come up with this particular situation that is all correct processes with a decision,

Because it is always possible because I am saying that if we decide only if all votes are yes therefore, if some parties have decided yes, but before they can vote yes if they die right? Then you can not you are in indeterminate state. So, and this kind of models are appropriate for databases and file systems, because for file systems with respect to metadata consistency and for databases for application-based consistency, because they are something important and it has to be handled. So, using this kind of a atomic commitment protocol we can construct more complex ones which agree.

On a particular value or which file has to be, which file has to flush it is data to this disk or whatever we can construct all those things. So, so you can see that there is a there is assumptions made and this for example, no more failures occur for a sufficiently long period of time this is looks linear to the eventual consistent model we talked about.

(Refer Slide Time: 41:29)



Consensus

- Agreement (A) All non-faulty processes reach the same decision
- Validity (V) If all non-faulty processes' votes are Yes, they will all decide to Commit; if all non-faulty processes' votes are No, they will all decide to Abort
- Weak Validity (WV) If there are no failures, V holds
- Very Weak Validity (VWV) Both Commit and Abort are possible decision values: i.e. there is an execution in which correct processes decide to Commit and an execution in which correct processes decide to Abort
- Satisfaction of A and V: consensus problem.
- Satisfaction of A and WV: weak consensus
- Satisfaction of A and VWV: very weak consensus

The slide also features a small inset image of a man with glasses and a dark shirt, likely the speaker, positioned in the bottom right corner of the slide area.

There is also this something called consistency problem. So, in the previous case here we are not saying anything about what the faulty processes are doing the faulty process is also have to say yes which is bit of a peculiar thing whereas, in the consensus problem.

We can exclude the faulty process. So, basically in some sense the assumption is that the faulty process is they basically crash and then come up again somehow that is the assumption made in this model. Whereas in the case of consensus faulty processes can do arbitrary things they can be Byzantine. So, that is why we are carefully dealing it in how we make up our minds on consensus. All non-faulty processes is a same decision again a special reason is non-faulty process whereas, in the case of the atomic commitment.

We do not talk about non-faulty process it is also a validity issue, if all non-faulty processes votes are yes, they will all decide to commit if all non-faulty processes votes are no they will all decide to abort. So, this is validity. Now one thing that one should mention already is that there are some fundamental problems here itself it turns out to detect if the process is faulty or not is itself is not feasible even that comes back to FLP again.

So, these things are impossible infeasible a and b together are impossible you need to have you need to have some weaker models one weak model is if there are no failures V holds; that means, that of course, when no failures therefore, you do not think about the non-faulty processes therefore, this is appropriate then there is also Very Weak Validity both commit and abort are possible decision values there is an execution in which correct processes is decide to

commit and execution in which correct processes are decide to abort. So, there is lot more latitude here.

Here we mention that if all non-faulty processes are yes then they will have to decide to commit and here it is possible that they also in spite of it they can do an abort (Refer Time: 43:54) the only thing that is been said here is all agree or commit abort that is always said where as this one is stronger it turns out that you can come up with varieties of weaker kinds of problems starting with consensus problem this is the strong one weak consensus and very weak consensus very weak consensus for example, is basically agreement and they are very weak validity.

So, as I mentioned earlier this consensus problem is used in the real time systems community this air traffic controller kind of situations they actually try to follow this kind of model. The idea here is that when some part of the system dies you do not you are not going to wait for it to come up and in the cases of databases or file systems usually when they do some operations they have some persistent storage which has to be looked at when they come when they come up again. That is when they boot reboot again there is already some state that has been accommodated in the previous; in the previous life you might say and that information has to be used to guide the new booting system. So, therefore, it can take much longer period of time, and so, it makes sense for you to go for this model atomic commit kind of models for databases and file systems whereas, for those real time kind of systems consensus is lot more sensible because you are not going to be you are going to be (Refer Time: 45:40) to something quickly and you cannot wait for systems to reboot and use the word old state persistent state to guide what has to happen next.

(Refer Slide Time: 45:51)

Relation Betw AC and Consensus

Differences betw AC and diff versions of consensus concern


- the decisions reached by faulty processes; and
- the strength of the conditions required

AC attainable only under the assumption that process failures are benign

Can prove

- AC 2,3,4 *imply* WV but not the converse
- With “no-catastrophe” axiom (NC): all failures repaired and no new failures for a sufficient period of time, then AC1, AC4 and NC *imply* A

AC conditions stronger than WV, assuming NC



So, if you look at atomic commitment in consensus we will see that there are the differences are, because what happens with respect to faulty processes as I mentioned earlier in the case of atomic commitment even faulty processes have to say yes for commitment; whereas, that is not really necessary for consensus.

So, it turns out that atomic commitment as I mentioned earlier is attainable only under assumption that processes failures are benign whereas, it is not valid for it is not valid for byzantine failures. We can quickly look at and convince ourselves that certain things imply something else it turns out that atomic commitment has conditions much stronger than weak validity and various results theoretical results have been shown about the weak validity model for consensus and essentially what you can say is that everything that have been proved in that context is actually going to limit commitment also. So, let us just look at one or 2 things quickly and see what the issue is. Now if I look at the conditions 2 3 4 imply weak validity what is 2 3 4? Commit is decided only if all votes are yes if there are no failures and all votes are yes then all processes decide to commit.

So, if these things if all existing failures are repaired and no new failures occurs for a sufficiently long period of time then all processes will reach a decision you can see that this is connected with a weak validity. What is it saying? It is saying that there are no failures then V holds. And so, if there are no failures then we can say that if all process is right now all of

them are correct all votes are yes, they will all decide to commit and if all processes which are correct now because there are no failures or no they will all decide to abort.

So now the reason why 4 is included is because if you have some failures, but no new failures occur for a long period of time then again, this issue is similar to what is there in the condition here therefore, you can show that 2 3 4 essentially imply weak validity. The reverse is not true because as I said earlier in the case of here even faulty processes have to agree on yes which is not required in the case of consensus. There is also another axiom you can put. So, that example I had this condition in the case of commit protocols about this part and it is not there for consensus. So, if I want to make both of them similar then basically I can add this part if I have this part then I can essentially get the agreement part of it because then at this so. In fact, that things stabilize right? It is available now for it is consensus also therefore, it is possible for me to now say that AC one AC4.

And the fact that thing stabilizes after some period of time they imply agreement. So, basically the attainable theory basically show that there are various conditions various communities are assuming some condition stronger some conditions are weaker as I mentioned earlier the storage systems or database community they have much stronger requirements compared to the other parties who are assuming weaker model because for them progress is far more important than. Again, if you look at the web community same situation for them progress or availability is far more important than complete consistency.

So, they have much more weaker models. So, that explains why you can use the web this is highly available, but it can give you some incorrect information and it should try to make it consistent then we have situation where it is not possible. So, what we will do next is to look at in the next class we look at some commit protocols what that is called the true phase commit protocols and then we look at it is natural progression to 3 phase commit protocol and then we will go to either models like taxons and then get on to how some of these things impact a design of large case storage systems.

Thank you.