Storage Systems

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

Design Factors
Lecture – 30
Design of Large-Scale Distributed Storage Systems_Part 1: Design issues in real storage systems, Impossibility results, Brewer's CAP Theorem, Designing real storage systems for CAP. The Multicast Oscillatory Behaviour problem

(Refer Slide Time: 00:14)

Lecture Outline

This lecture is the first of 2 lectures (Lectures 27 & 28) on the design of large-scale Distributed Storage Systems. The focus of the present lecture is on inherent characteristics & complexities of real distributed storage systems, important desirable properties of such systems and ways & means of designing real systems so that they approach desirable properties. Real-life large-scale distributed storage systems are very complex systems which need to be distributed, replicated, provide support for QoS & security and solve co-ordination problems, consensus problems & many other problems arising from asynchrony. We try to understand how real systems actually behave and why they fall apart by studying live examples like "NFS: dir lockup problem", "lockd problem", "Recent 2.5hr Outage at Facebook", "Skype malfunctioning" etc. Next, we look at some important impossibility results for asynchronous distributed systems that are prone to failures. The next topic is Brewer's CAP Theorem, according to which all 3 properties - Consistency, Availability and Tolerance to Partitions - cannot be guaranteed simultaneously in a distributed system. This, however, is not a serious limitation since many real distributed systems like Cluster filesystems, Web caches, DNS Services and Distributed Lock Managers do not need all 3 properties to be satisfied at the same time. We try to understand the capabilities & differences between filesystems and databases from a design-for-consistency perspective: we look at the basis for designing Databases (the "ACID" semantics); we study some interesting aspects of designing Networks (like consistency), Cluster systems (like consistency & availability), Distributed filesystems (like availability), Large Storage Systems (like durability) and Protocols (like reliability, "reincarnation"). Achieving CAP is impossible, but are there ways & means of designing real storage systems so that they approach CAP requirements?. In the lecture, we consider some possibilities - can the frequencies of failures be reduced?, what if we make sure that Partitions are repaired within latency time?, can we relax assumptions like "Timed Asynchronous system"?, what if we adopt a policy such as "Dont do anything when partitions are present"?, what can we achieve by attacking root causes (suggested by Birman)? etc. The final topic is an interesting problem called "Oscillatory Behaviour" that is faced by large scale Storage systems which use multicasting during networking. The Professor concludes by saying that, as pointed out in the lecture, there are innumerable factors and variables present in large scale systems which make it impossible to accurately predict the happenings in real distributed storage systems, and, hence our ability to (even theoretically) provide gaurantees about the functioning of large-scale Storage systems is rather limited.

The NPTEL course of storage systems. So, in the previous class we looked at a network file system, and we mentioned that there are some interesting problems with respect to consistency management in nfs. So, we started with the nfs 2, then went to nfs 3, then went to nfs 4. And each of them has it is own model of consistency. And you can see that was increased to what you can expect from a particular implementation. Now goes slightly deeper into understanding why is it so difficult to give this consistency guarantees. It is not an easy thing it is not easy because there are issues like failures in the system, but create some complications.

We would like to understand this in some depth; that is what you are going to do this class. And hopefully it will give us an appreciation why is it so difficult to engineer good storage systems.

(Refer Slide Time: 01:41)

Design Issues in Large Storage Systems

- Large Storage Systems need to be distributed and replicated
  - For throughput
  - For resilience to failures
  - For disaster recovery
- May also need to solve "(w)holistic" problems such as RT/QoS and security

Usually, need to solve many *coordination problems*

- Many other imp aspects but difficult problem already!
- Difficulty arises from asynchronous nature and failures characteristic of real systems
  - Faced by Amazon, eBay, Skype, Gmail, Facebook...
  - (Synchronous systems typically small scale)

Now, if you look at the some of the issues in a large storage system, what are the issues? First of all, you have to have if it is a very large system it has to develop a network. Without networking it is not possible to give you a large storage system; that means, it has to be distributed, and because of failures if some particular node dies it may take away it is portion of the storage.

So, if there is something to be avoided you may want to replicate information. So, therefore, when one can say without hesitation; that large storage systems need to be both distributed. And replicated in some form it has to be some redundancy it has to be distributed. Now it for multiple reasons, I just give you one reason for it that was for resilience to failures. But it can be due to also for throughput reasons. It could be also be due to disaster recovery. Given the kinds of let us say large scale disasters that could be afflicted on storage systems, or the computing systems this also is an important issue.

But disaster recovery for example, banks there already regulations which basically says that you need to have at least 2 sites which are 30 kilometers apart; that means, you need to have the information in a bank should be available at least 30 kilometers away. I am sorry, it should be yeah, I think it is between 30 and 40 kilometers. It can not be too close you can not keep one another site about one kilometer. That considered reasonable.

So, valid reasons you need to do distribute and replicate information. And there is also other reasons why you need to do such things. For example, for security it turns out to be

a holistic problem; that means, it is even a small little problem somewhere can actually be magnified into a huge security incident. I think you might have heard about the flower board, a weak link being a problem right in a chain. But that means, that every important somehow. Similarly, if you are talking about real time or quality of service. Every small thing actually counts.

If one part of it is not doing it is job well it can affect the guarantees you can give. So, fundamentally we can see that you need to solve many coordination problems. So, let us to me this is the most important problem. You have to solve the other issues, but this coordination problem if you can solve in you can take care of. And generally, this difficulty arises from asynchronous nature of the systems and failures a characters with characteristics of many real systems. If you would look at synchronous systems; that means, that somehow you have times time synchronization somewhere and typically time synchronization does not work on large scale it is very difficult.

So, if you are talking about synchronous systems these systems have to be small scale. It can not really scale beyond certain size; that means, that any large system has to be asynchronous; that means, that there is no the timing across these systems cannot be mandatory or centralized. Let us say timing device or whatever it is just not possible. So, it is a fundamental problem again to repeat is because of asynchronous nature and the failures.

(Refer Slide Time: 05:22)

NFS: dir lockup

Consider a "slow op" on a (locked) file due to NFS congestion

- LOOKUP on that file results in a lock on its dir that cannot be released until "slow op" finishes =>
  - cascade of locks upto root that hangs the system till "slow op" finishes
- lockd: similar problem but worse as lockd a user process

Because of this there are a lot of difficulties in designing a good high-quality storage system is can guarantee many properties if you look for. I will give you some simple examples of the kind of problems, that we can get into.

Let us take the case of we looked at nfs last time. We were look at how nfs can get into some peculiar situations. So, let us say I am there can be what is called a directly lock up in nfs. What kind of problem is this? Suppose there is a file which is locked. And this is taking a long time for some reason. I am not we are not going to discuss why it is taking a long time. It is taking a long time if suppose somebody does a lookup on that file, another independent lookup.

Now, it turns out if you do a lookup on that file, the important thing is you are supposed to give some information about the file, which is let us say should be the information you are going to give should be something that is picked up atomically. That is why we are looking at it, it can not be changing in a sense; that means, only we can do it is by ensuring that it is parent directory, you take a lock on it to prevent anybody else operating on the file. So, what will happen is that if you do a lookup on that file which is actually doing some slow operation some other independent lookup.

The directory actually is going to be also you take a lock on it directly first before you start thinking of doing something on the file. So now, this lock on the directory is going to be held until the slow operation on that file original operation itself completes. Now you can imagine a situation the other request come in, we do the same thing on the directory also. So, it turns out that you can come up with a cascade of locks all the way up to root and everybody is locked up now.

So, this is now hostage to the slow operation that has to finish. Only that finishes the whole system actually makes forward progress. While this is going on, the system looks highly unresponsive or dead, we do not know what it is. We do not know whether it is unresponsive which is dead, we have no way of finding out. So, this is not an artificial problem it happens all the time. Basically, if you take your slash user war etcetera, right this is a system directories if there is some for some reason in one of the directories there is some slow operation. It can travel all the way to root very quick which is 3 or 4 steps, and your whole system looks as though it is dead.

Those of you have used nfs oftentimes will find that the system suddenly seems to be dead. And then suddenly moves forward, right. It suddenly starts suddenly seems to be alive again. It is because of this kind of issues. But the important thing to notice is that, you cannot be sure whether system is dead or it is actually just painfully slow. There is no way to distinguish between this. That is a critical thing that you have to remember. Of course, if you use some other types of a locking mechanisms, it there are similar problems. For example, I mentioned that in nfs it uses some additional protocols services like lock demeans. And you should take those things we also have a similar problem there.

And this can be on worse because lockd is a user process; that means, that it has to be scheduled in for it would be any useful thing.

(Refer Slide Time: 08:53)

Recent (Sep 23, '10) 2.5 hr Facebook Outage

- Caused by an automated system to check for invalid configuration values in cache and replace them with updated values from the persistent store
  - works well for a transient problem with cache, but it doesn't work when the persistent store is invalid
- Somebody made an "invalid change" to persistent config values
- Each alert client attempted fix of invalid cache value=> has to query cluster db that cannot scale (=>1000's queries per sec)
  - Also deleted cache key
    - Now queries do not succeed in the cache *during and just after* the fix!
    - Each new request has to go to non-scalable db again!
    - Positive feedback cycle with more requests to db
- Had to stop all reqs to cluster db to recover; *site down*

Now, let us take a slightly more interesting example, which is the sometime recently, facebook had a fairly long outage. I am just summarizing what I understand from the blog entry that facebook provided, about why their system failed. Let us try to give you some idea about the their real systems actually behave, and the way they fall apart.

Now, in the system, we will not discuss some of the exact reasons why something exists in the system, but we will just take it for granted. The there is some configuration values in a cache. And basically, this system already has been provided with lots of capabilities of making sure that that configuration values in the cache can get corrupted, and

somebody has to check it automatically. And this cached values come from a persistent store; that means, that even if you reboot a system that value that persistent store keeps track of it. A cache is basically a volatile value, but it is coming from a persistent store.

Now, there could be a problem either in the cache, or it can be a problem in the persistent store. Now this particular automated system that was designed, works well for a transient problems with cash, but does not really work very well with when the persistent store is invalid. For some reason somebody made an invalid change in the persistent configuration value. It is always possible, somebody makes a mistake. Now if there is a invalid change to persistent configuration value, then what will happen? If your system has been written well, the alert client will notice, that something in the cached value does not work, right. We can detect it by because for example, things like crc or a hash or some such thing we can keep track of whether some configuration value is valid or not. There is some way some simple way to check; that means, each alert client took some cache value configuration value, and discovered that those on anomaly.

So, there is some simple procedure for doing it. Also mentioning it can be something simple like checksum. So now, if it finds a invalid value, then it has to go and talk to a database, which can give the correct configuration values. Now this database is not get for high transaction throughput. Because by definition database tries to be we will discuss it later. It tries to be atomic and consistent, and it provides certain kind of guarantees. So, it turns out databases usually are not able to scale for very large number of requests. Because of this it turns out that every client which noticed this problem, it will try to because it can not figure out who else is also seeing the same problem.

So, everybody who makes the request, along the same time. And because there is a problem with the cache, it turns out that any new queries, they cannot be made to succeed in the cache during and just after the fix. Because they do not succeed. So, basically; that means, that all the new requests also have to go to the database. So now, what we have is a database, which has been configured for some reasonable performance. But now all these guys who have detected invalid values they are going and starting to hit on the database. And any new values which cannot go through the cache anyway they also start it in the database. Because there is no other way to get the information.

Now, this database actually is now saturated. So, in a sense what is happening is that the queues and the database become very, very long and oftentimes what also happens is that if some value is not responded within a particular time; these systems can also assume that there is a network problem, they will retransmit again. That means that in instead of waiting for some period of time like good citizens, they think that something bad has happened to make sure that they get their response quickly enough. They go and again retrieve the request again one more time.

So, in a sense what happens is that in a short time you can get so many requests to the non-scalable part of your system. But finally, what happens is that there is no way to recover from it. So, in this case for example, when this happened in this facebook situation. They discovered this problem they tried to see if that was going to resolve itself, it didn't resolve they waited for some time finally, decided there is no way to take care of it. That essentially disconnect the cluster database which was supplying the cached values, which was supplying the configuration values. And then once that was done essentially the site became unreachable.

Similar problems exist in case of skype also. If you look at there are incidents of skype also malfunctioning for multiple hours, and must not multiple hour sometimes even days. And so, this is not something uncommon. It keeps happening luckily the systems are engineered well enough that is happens not all the time it happens once in a while.

(Refer Slide Time: 14:06)

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)
  - ***c**rashes, **a**synchronous, **m**emoryless, **l**ossy model*

System can be driven by a sequence of crashes to ... *obal state where each node is in a state reached in so* ... *ssibly diff) run, and each link has an arbitrary mixture of p* ... *ts sent in (possibly diff) runs* (Jayaram/Varghese JAC...

NPTEL

So, let us see why failures are so difficult. The first simpler problem is the following, is something called the 2 generals coordination problem.

Now, in this case, what we have is, there are 2 generals who are divided by an env territory, and the 2 generals only if they work together that it can defeat the enemy. But the problem with the 2 generals have is that they can communicate only through the env tertiary. When the you send any message to the env territory, the messages can be seized, it can be corrupted, it can be dropped etcetera. So now, you have to solve the problem of coordination across these 2 generals, who can only send messages through the env territory.

It turns out there is no protocol that can solve the problem. Given this conditions. You can see easily a solution by this should be the case with a the following kind of argument. Suppose, you say that there is some way to this 2 parties 2 generals to agree on when to attack the enemy. Because when they attack together, they succeed if they attack singly they cannot succeed against the enemy. If they, if you say that there is a protocol that solves the problem, then I will say the following.

Make sure first you remove all non-essential steps in the protocol; that means, finally, you are left with only essential steps. So, there is a protocol, first of all I do I do is take off non-all the take out all the non-essential steps. Because I take out non-essential steps, everything has to be everything every step is non-essential. But I already said that if I send something through the env tertiary, the protocol step can be seized duplicated corrupted etcetera; that means, I just drop the last one. For example, that means, that the 2 parties cannot really conclude the protocol. There is no way for them to conclude it.

So, whatever argument is say, I basically will tell you that if they are succeed in agreeing it the last step should be the one it is leads to agreement, I just drop that one. Therefore, there is no way for you to say that they can agree on it. So, this is a some kind of a simplified or high-level argument, why no protocol. So, say the protocol can exist. So, in some sense in under these conditions coordination is impossible. That is first problem. There is a another problem which is called FLP result, Fischer lynch Patterson result; it stands for impossibility of distributed consensus with one faulty processor.

They show that in a distributed system, if you want everybody to agree to a particular value, if even one of these particular entities can be faulty then the distributed consensus

where all of them agree on a particular value cannot be satisfied. It is not possible to do it. Not just that you can go with some other results also. There is a result called impossibility of reliable communication in the face of crashes. If you have machines which crash, the result is basically shows that; there is no way to have reliable communication.

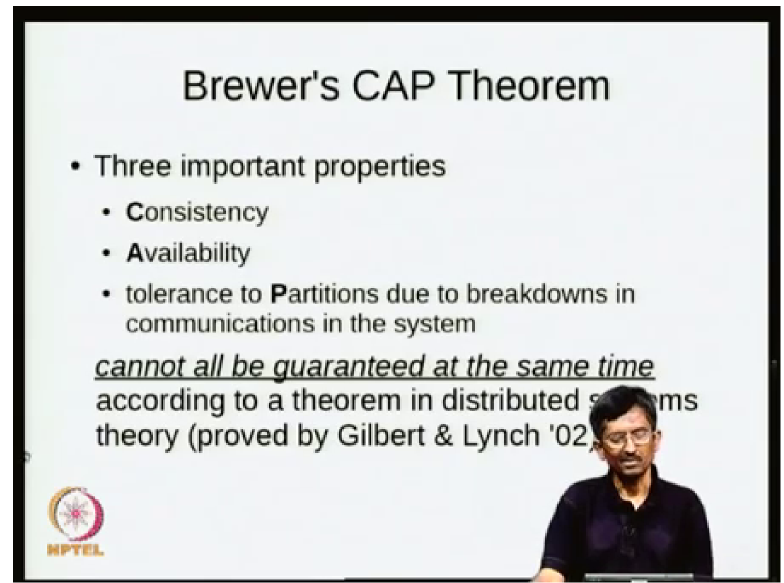
What do you mean by reliable communication? Either a message goes or it should go and exactly once. It cannot have duplicates in whatnot. So, this also has been shown to be impossible. You can even go further down, some other results have been derived, which basically says that let us this take the case of data link layer in a network protocol. If you assume that nodes can crash, it is asynchronous it is memoryless; that is when the do when the nodes for example, do not remember what they were doing previously. That is there is no stable storage by which they can figure out what protocol step they were doing previously.

If you have memoryless, and lossy. lossy means that I send some packet it can be dropped. It turns out that there is no reliable data link layer that can exist in this model. So, the foundation that we assume in a iso stack with a physical layer on top of data on top of fitted data link layer, it basically says that this is also impossible. This is of course, they are all let us say, extreme kind of results you might say. But once you assume that machines can crash that there is a synchronous there is no time synchronization memoryless; that means, the notes do not remember what they were doing before they crashed.

And if you assume that packets can be dropped, then this also is not possible. Actually, it turns out the situation is even more horrifying, actually this paper they show that a system can be driven by sequence of crashes to any global state where each node is in a state reached in some possibly different run, and each link has an arbitrary mixture of packets sent in possibly different runs. I will suggest to look at this paper to get an idea what all this means. But this is a pretty awful result. Basically, what it means is that if I am doing something for crash, I might get the packets when I come up again I might get the packets I might send my packets to somebody else, or somebody else the packets can come to me etcetera.

All kind of things like that are possible. So, luckily when we engineer our systems we sure make sure that this kind of horrifying things going to happen too often. And that is where if it is potent, but fundamentally it is a serious problem it is not something that we can wish it away. Because they come from the basic issues the once we assume for example, that messages can be corrupted or dropped or the notes can crash.

(Refer Slide Time: 20:13)

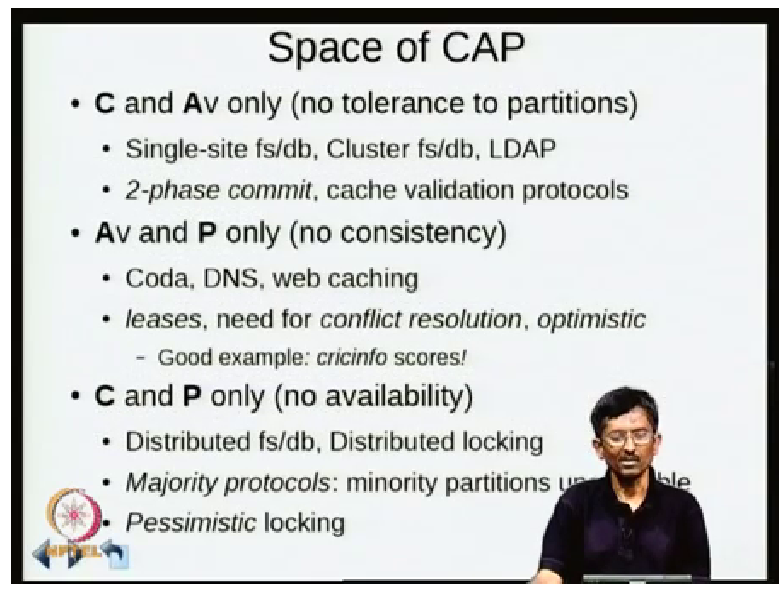


So, following on this, there is another result, which is called a brewers cap theorem. Which tells you the following. That if you are looking for consistence in the system; that is, the data that you are that is present in the system is somehow consistent according to some model. And the data that is there in a system is available for you, availability I want you able to use the data. For example, I have a file on a system I want to able to access it availability. And then tolerance to partitions. Basically if it is a distributed system, I want to have the nice situation where in spite of failures in a system partitions in the systems, I still I am able to do useful work; that is basically tolerance to partitions. Now this brewers cap theorem, basically has been shown that it has basically shown in this cap theorem that not all 3 can be may be satisfied at the same time; that is, either can I have 2 of them 2 out of 3 a data. I can either have consistence and availability without this property, or I can have this 2 without this property, or these 2 without this property.

So, this cannot all be guaranteed. So, all these 3 things are typically important for us. I want my data to be consistent, I want the data to be available. I do not want my data to

be not accessible to me just because some partition takes place somewhere else. I do not want distribution, where I cannot access something just because some failure is taking place somewhere else. So, all these things are important, but suppose surprisingly or according to this theorem, it does not it is not possible.

(Refer Slide Time: 21:51)



So, we can see many as may many cases of this. What are the kinds of situations that we are coming to. For example, I can have consistency and availability only. If you take a single site file system; that is, on a single desktop system, I can have consistency with the database. And I can have a availability; that is, I do not expect any failures essentially, if there are no failures I can give you some something reasonable. So, all the systems as long as there are no failures they work well. If there is a failure, they just block. They block or they have to get into some uncharted territory of inconsistency or availability. So, all the systems basically work well as long as there are no failures.

Or we can go to some situations where you drop the notion, drop the idea of consistency; you say that consistency it is that hard, I am willing to accept weaker notions. And then I will try to leave with it. You will see that most of web is based on smaller. In web availability is far more important than consistency. Because most of the web information is usually not very critical. So, for example, if I am reading a newspaper or whatever it is not that critical. Because news often gets updated all the time. So, if I get slightly inconsistent version the news, it is sort of tolerable.

Same thing with cricket scores. I think, it is a common experience, but when you are looking at some of these cricket scores; it turns out that after an event has happened, you might get a another information which essentially invalidates the previous event that has seemingly happened. For example, my scorecard be 345 for some 3 wickets, and then I might get a another information saying it is 340 for 3 wickets, right. Later, it happens because, the information usually travels on the web through multiple routes. So, I got the first correct information 345 for 3 through one part, and it somehow became the not the preferred route for this when the second came. And that actually had been slightly more stale information.

So, it turns out both DNS web caching and disconnected of clients, all these things usually do not provide consistency, they give availability some information available some score is available to you. But no guarantees on how good looks score is. Same thing with DNS also. Because it also does some kind of caching, and gives the information is incorrectly updated for a small window of time, it can be given to you. And basically, in all these cases essentially you are optimistic.

So, you are basically saying that it is to have slight inconsistent values. It will be corrected soon sometime soon, this optimistic way. So, because of this sometimes you will see that there is a need for conflict resolution; that means, I got 2 scores, I mentally at least have to resolve what happened. And in the case of crickets scores you might not do any conflict resolution, but in some other cases you need to think about what could happen why this is the case. And for example, email might have come one way and some other email might not have come some other way, and I might have to understand what really happened and based on that reason something about it.

And typically, here, we have the notion of leases. Basically, because it is difficult to be strict about consistency. So, what I can say is as long as my I give you some exclusive access to something for a period of time; and this is governed by time, as long as my time synchronization is very good. Then the I am reason be safe, but time synchronization itself is not possible to be very accurate, because there is some it is got a you cannot do time synchronization to better than few tens or 100s of millisecond accuracy it is not possible, means that there is still a problem. And that basically means that, I might assuming that (Refer Time: 26:16) lease there could be some corner cases,

where 2 parties think that they can go ahead and update this thing at the same time. That is possible.

So, you can also have the situation where there is consistency and what is a tolerance to partitions. But you do not have availability. And this is a typical situation, your distributed file system and database and distributed locking. And basically, here basically what happens is that, if you find that some network partition has taken place, you basically stop giving access to certain data. Because you are concerned that people can get wrong data, you basically take the extreme step of say, I do not have any data. For you look for the time being till things are little bit clear.

So, if you use what is called majority protocols, what is this? It basically you if there is a partition in the system, what you do is you have something called majority protocol. Basically, you vote the majority people agree on a particular value, you basically say that there is a value. So, in a sense you are able to tolerate partitions, but now the minority partitions, they may not be available. So, that is why there is no availability. So, some part of it can be proceeding. And there is tolerant to partitions, because some parties are still able to get some work done. And so, but certain things are not available.

So, here typically it turns out that pessimistic locking is done; that means, you take as extremes that is possible to ensure that consistency is, let us say guaranteed. So, you will see that databases usually is work in this area. And I made a mistake sorry. Here is where web usually works in this area. Databases work in this area. File systems work in this part partly sometimes partly here and partly sometimes here. We will come to that details here. Yeah sorry, I will do it right now.

(Refer Slide Time: 28:18)

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. Uses "ACID" semantics:
  - **A**tomicity, **C**onsistency of data: DB's responsibility
  - **I**solation from other transactions also
  - **D**urability: storage system's responsibility

So, let us just look at high level, what is the difference between file systems and databases at a high level. What is the file system? It is a persistence and naming service for all up for all applications. The critical is aspect about this is that, there is no information from application, and what information is critical. That is not given to you. All that the system can do is differentiate between 2 types of information data and metadata. That is only thing that that the system has got accessed to it. So, what can a file system do, because it has got only 2 types of information, a 2 ways of classifying things data and metadata. It can guarantee possibly consistent of metadata, that is one possibility.

Anyway, this is needed because metadata is kept by the file system to manage the system, manage the data. So, this is something which is file systems. So, it is needed anyway for file system sanity. It can not abandon consistency of metadata. So, it is something it is critical. The only issue is that in spite of worrying about consistent metadata, even then it can get corrupted. If it is corrupted, the file system says I have no real serious information about what to do next best, I will fix it in some suitable way. So, the system can function, again I want to make a system available. That is my perspective. And that is what is some of you might have used something called fsck a older systems. And even if you have what is called journaling file systems, some once in a while you have to do fsck, it is still necessary.

You can avoid it to some extent, but numeric corruptions all those things when they come in you can not really avoid it. So, consistency some metadata's taken seriously in the file system, but data consistency may not be considered very seriously, because this file system has no idea, because nobody has told it how important certain piece of data. So, the file system optimization is basically following I do not want to be very slow, because very slow then nobody losing the file system. And all the basically is the most basic service for persistence in the system if this is slow, then the whole system really it becomes atrociously slow. Hence the idea here in file system is to sort of weaken the data consistency model, and main system reasonable responsive; that is a basic idea. And why is this an important issue, because you are dealing with things like disks, which are amazingly slow compared to a memory.

They are order of about 5 5 orders of magnitude slower than memory. So, because of this reason, the file systems usually I take something the bargain they have entered into is the following that. Since you want some reasonably responsive system, I will relax the consistency condition for data. I will try to be as far as possible consistent with respect to metadata, because that needed for my own sanity. So, how does this a file system do it? It uses a synchronous operations in case, you need to have data consistency. Not what exactly do I mean by this? Basically, in file systems, you can if some operation has to be done. It has to finally, make it is way to the disk. So, that it becomes persistent.

Now, I can ensure that I do everything synchronously. Every time something changes, I immediately flush it to disk that is one possibility. The second possibility is if I say something, I do not flush disk immediately. All I do is achieve the flushing, but I do other things after work. That is another possibility. A third thing is I do not even (Refer Time: 32:08) it is what is called delayed rights, the 3 possibilities. One is, I do it and wait for the flushing to have happened on the disk. Then only I proceed. That is the most conservative or the synchronous type of operation.

The second one is, I am doing it asynchronous right. What I mean by that is; I cue the right I do not know how long it is going to take. So, I do not want to keep waiting. So, I just go and do other things. That is asynchronous option. The third one is delayed right. I do not even do not cure it. I will expect there some time later somebody, because of some urgent reason is going to do it for me. I am not going to do it myself. I am not even going to cue it. I am not going to wait for it. I am not going to cue it.

Now, because of this, it can turn out that if you use for example, the delayed right. Whoever is going to flush it to disk, has no idea about the dependencies of each piece of data with respect to others. So, something which is important to be they might be in particular order in which you have to flush the data. For example, if you look at that cricket scores, what you wanted was that 260 runs after 3 wickets should come first followed by 260 followed by 260 whatever, right? You want a particular all the request to come right.

Similarly, data also can have certain orderings. But because in delayed rights, somebody else other than mu is flushing it to disk, they might not know the ordering between these things. And they will flushing problem in correct orders. And if there is a crash of the system, you might see in inconsistent ways of the data that has been flush to disk. So, this is one problem the file system has got as long as you are trying to be make a system responsive. So, there is a in a sense, we have to understand what the file system is trying to do for you. It can not give you the guarantees, that you seek for except with the cost of doing extremely slow. So, that is why it is not the default.

On the other hand, a database uses something called acid semantics. Actually, a file system also can use acid semantics. It is does that it becomes too slow for it will be useful. A database other hand usually deals with some a real world important information like, bank balances and other things. It can not take this approach. Because if your bank balance becomes inconsistent, it is going to be a serious issue. Therefore, databases in spite of slowness, they will insist on a particular set of semantics. This is the acid semantics. Let us do databases cannot scale to the way a file system can scale. So, the number of operations that number of transactions and database can do typically is at least one or 2 orders of magnitude slower, than the kind of operations that a file system can do.

So, what are the kinds of things that a database is let us say trying to guarantee. When it is atomicity, what is atomicity? It turns out that you want to ensure that either some transaction happens and what happens. You cannot have some in between situation. Either your bank transaction happened or not happened. The consistency of data. If you have multiple transactions, there might have to be consistency across these transactions; that is example if I withdraw some amount, I try to check from my account to somebody else’s account. You should decrease from mine an increase in other persons account.

But the amount of decrease should be equal to the amount of increase account. That does a consistency condition. Again, this is an application idea about consistency. It is not inherently known for example, to anybody else or an application. Let us the consistent data. So, database also has to be responsible for this. And isolation from other transaction; that means, one transaction would not interfere with other in transactions. Each even if I concurrently run it, it should not be the case that one transaction somehow negatively impacts other transaction. Each transaction should seemingly in spite of being done concurrently should seemingly be just as if it was isolated done as an isolation.

Finally, durability, I do a transaction any effects that I should see should be persistent, that is durability. Now this one part which the database depends on storage system. Rest of it is again actually it has to handle it itself. Now oh, if the database is running on a file system, then the file system database it depends on the file system to provide the durability. And the file system in turn has to find some mechanisms by which whatever things are happened with respect to database, that has to be made persistent has to be made durable.

So, that if the machine dies and comes up back comes up again. Whatever changes have taken place actually can also be seen again. So, this turns out to be the responsibility of the storage system, and the database can directly do it itself, but it can go to the file system right. So, if it goes to the file system, and then it has to make sure that the file system understands importance of whatever it is changing. So, it has to indicate through some mechanism. Usually, this is done not by through what is called mount options. A file system can be configured saying that all this operations have to go synchronously or in some particular fashion which ensures that things become durable in some particular way.

So, it has to be some understanding between database and file system. Or the database can directly talk to the storage system, and the database can tell the storage system specifically please do the synchronously. I am going to wait till you are done. So, that also is possible. It turns out the durability is not a trivial thing I will come to that soon.

(Refer Slide Time: 38:13)

Remarks

- OS/Networking good at availability, but not good at consistency
  - NFS a good example here
- Can have consistency & availability within a cluster, but hard
- dfs/db better at consistency than availability
  - Wide-area DBs or Disconnected clients neither
- Durability is HARD: a large storage system itself composed of many parts
  - Recursive problem: how does it keep its own metadata or data "consistent" or "atomic" wrt changes and what persistent store can it depend on for its operations?

So, let us some brief remarks. Basically, it turns out if you look at operating systems or networking etcetara typically they are good at availability. Not so good at consistency. We already look at nfs as a good example of this right. So, basically in nfs what we are doing is to avoid too many round trips. Clients keeping on checking whether some data is valid or not. Sometimes they do cache the attributes therefore, they can takes a window during which they assume that data is not been modified with somebody else, when it has been modified.

Similarly, you can have in a cluster, it is possible to have consistence and availability, but generally it is very hard. And basically again because of reasons for failure etcetera. If you have distributed file systems are databases, they are better consistency, but they are not good availability. Same is the case with wide area databases, or disconnected clients. There are some problems here also. So, basically you have to decide what kind of model you want, and make sure that the system is engineered for the particular reason.

Let me briefly mention durability. Why it is so hard? First of all, if I am talking about the large storage system which is going to guarantee durability; it itself is composed of many parts. Now there is a recursive problem you can see now, because if there is a large storage system, it has lot of data itself. Now how does it keep it is own data or metadata consistent or atomic with respect to changes? For example, what are the kind of change that can be there it is possible that one node dies. Somebody has to keep track of that this

particular node are discussed field. This has to be made persistent, because there is some kind of a table somewhere saying that this particular disk died, right.

And other people could be referring to it. So, it has it is own metadata. Now how does how is this also guaranteed? So, is a recursive problem here. So, in a sense, if you look at acid property; the first 3 parts the database tries to achieve. But the durability itself is recursive it actually can be lot of people say in the storage systems area, that to give you that quote acid property for this storage system they might again use acid, but again the d part of that acid with storage system itself does we can be guaranteed through some other mechanism.

So, there is a fairly tricky issue here itself. And so, these are basically done by some careful engineering to ensure that the problem becomes smaller and smaller so that finally, you can say with some high probability. I guarantee that something is persistent therefore, the whole chain can be made to a stand on that particular assumption.

(Refer Slide Time: 41:08)

However!
- Consider the TCP protocol
  - TCP, the basis of Internet, finalized in RFC 793 in Sep 1981. Only clarifications to this design since then.
  - "Just Works" inspite of theoretical result of impossibility of implementing reliable comm in the face of crashes
  - Careful engineering to avoid problems due to "reincarnation" of a connection
    - A TCP cnxn identified by (IP addr, port num) of src and dst ***(finite size)***
    - *If this repeated in a new cnxn, (incorrect) packet insertion possible!*
    - but still mandates (but not followed) "cannot reboot a system faster than 2 minutes or remember last seq# before crash" !!! (see Sec 3.3, RFC793)
- So an impossibility result does not have to make us drop our plans
  - Just make sure the engineering is very good!

So, basically all I am trying to say is that, it is true that there are many impossibility results, right. For example, we looked at all these cases, right all these issues.

So, but that does not mean that you cannot do certain things. I will give you an example about a protocol like the TCP protocol. It we know the web is very widely used because we use TCP protocol. And TCP protocol really works quite well. But we notice that there

is a result which basically shows impossibility of implementing reliable communication in case of crashes. So, it turns out if you think carefully about TCP, it has been possible that TCP really works for us because, our extreme very well carefully done engineering. Let me give you an example of the kinds of engine that has to be done.

Suppose, what is that TCP connection identified as? This identified as IP address and port number on the source file, and IP address and port number on the destination side. Now you should think about this, this is a finite sized piece of information. Now that means, it is a finite system. In a finite system you can always repeat. In infinite system, it is difficult to say that you can keep repeating. But in an finite system, for example, if the number of possibilities is 2 to the power of 16. It is always possible for me to if I keep reusing port numbers etcetera, I can possibly repeat that thing, right.

So, because at the finite size, if I start a new connection, and by shear bad luck, I have the same signature as this one; that is, it has the same IP address port number of source and destination. Then it is possible that some packet of the previous connection, which was hiding somewhere in internet somewhere. Suddenly comes and drops into my current situation, then I might have what is called a packet insertion problem. Why is this and not unlikely. Because you notice that if I am talking to 2 machines, only 2 machines. The IP address on the source on the IP address on the destine are both are same, anyway it is not going to change.

If I am talking to the same server on the destination side, the server port number also is the same. The only thing that is going to be changing is the port number on the source side, right. Now if I use my port number I cycle through port numbers very fast. Then I can essentially repeat the same identifier for the connection. So, because of this, there are various methods in a TCP protocol. If you look at the rfc tells you what all to do. To make sure that this kind of packet insertion is as unlikely as possible. I do not have time to go through all the details exact how to do it.

But there are many methods by which you will do it. There are something called sequence numbers you might have heard about which are used it has got something called 3-way handshake, which is also used to prescribe at this particular problem. Again, the details of what why it is 3-way handshake, not 2-way handshake; it is because of priceless this reason. And in spite of all the methods they have used to avoid this

problem, just as a fallback in case all these methods fail, this have one more requirement if you look at this section 3.3, they tell you that a node cannot reboot faster than within a 2 minutes after has crashed. That is the machine crashes, it should not reboot the thing it has to reboot at least only a 15 minutes some such rule is there.

Or we have to ensure that, it remembers the last sequence number before cash; that means, that anytime you have any sequence numbers, you have to put it on stable storage. It means that it has to be any time you change the any sequence number, because you send a packet which in the sequence number. It has to be available in there in a stable storage generally you can send the packet. Now you know that sending packet is going to be much faster than saving it to stable storage, because stable storage nowadays means typically only disk.

So, either you do the hard thing of remembering last sequence number which is very, very slow. Essentially you cannot send packets faster than disk operations. Either that, or you have this rule and there is other rules and typically this rule is one which is not very well known. You cannot reboot a system faster than 2 minutes to avoid disk (Refer Time: 45:56). So, it is lot of engineering that has gone in to make sure that you still survive all this impossibility results.

So, that is the reason why TCP is still surviving, if it was not done well there was no chance it would work on the scale and web acid designing. So, our idea is that even if you have some problems with various impossible results. It is possible for you to do something reasonable.

(Refer Slide Time: 46:27)

Ways to Get Around CAP

- Avoid failures? Impossible but can we reduce freq of failures by redundancy (say, 99.9999% uptime)?
  - Design and impl not easy + very costly
- Make sure partitions are repaired within latency requirements for a request: Too costly/Difficult?
- Assume "Timed Asynchronous Systems" (TAS): unstable periods followed by sufficiently long stable periods
  - "Failure aware" design
  - '96 FAA project based on this model but was a colossal failure as solution based on TAS subtle and needs care in impl.
- "Do not do anything when partitions present"
  - Fundamentally impossible to detect failure (aka partitions) reliably: FLP result
  - Availability is not possible

So, a various ways to get around some of these problems; I will just go through a few of them right now. One obvious way to say is why do not you avoid failures in the first place. Now this is not a trivial thing to do. Because we are talking about real systems, and real systems are main types of ways, in which you can fail disk can fail batteries can fail, the CPUs can fail they can get overheated or (Refer Time: 46:57) that are there.

So, this thing is almost impossible. The only other thing you can do is to reduce the chance of failure and usually if you try to reduce the chance of failure we are talking about extremely expensive systems. So, and this these 2 types of operations are also types of approaches are also followed. I think some of you might know that look at google file system. They decided that instead of trying to reduce the failures. They went for trying to manage the failures, whereas, if you talk about what is called enterprise systems, they try to reduce the chances of failure itself.

So, by having redundancy and other kinds of things so that design becomes typically expensive. Whereas, if you are able to manage failures. Typically, if you do it well you can reduce the cost. So, this approach of avoiding failures is typically extremely costly. Other possibility is make sure partitions are repaired within latency requirements for a request; that is, something fails you someone make sure that the partitions get repaired before the next request comes in this also is a hopeless task typically.

So, or you try to go to slightly more relaxed to situations. One example of a approach is what is called timed asynchronous models. You basically assume that a real system has unstable periods followed by sufficiently long stable periods. And there is this particular aspect has been worked out in some detail. And this is something called failure aware design, and unfortunately this has not been seriously taken up. Because there are some interesting projects that were attempted with this particular model.

But somehow it could not be need to work. Because this particular design failure aware design is fairly complex. This also has failed in some sense to me it looks as a very promising approach. Somehow, it has not taken up other thing to do is do not do anything when partitions are present our big problem is that even if this looks nice on the surface. It is impossible to in detect failure, when failure has taking place, because as I mentioned earlier sometime the systems can be arbitrarily slow. You do not know whether it actually failure or is it slow.

And I gave you that facebook example where the queues can build up to such an extent, that system is unresponsive and even after 2 and half hours nothing is going on all the requests are just sitting there somebody finally, has to disconnect the whole system. And drain out all the requests and then once everything is all the queues are cleaned out then you can send any. So, this also is sort of a not a feasible thing, the other issues of other ways to handle it also.

(Refer Slide Time: 49:48)

Birman: Attack Root Causes?

- As data center networks scale out, sw stack increasingly oriented towards one-to-many (multicast: MC) communication patterns: eg. Facebook/Twitter
  - Publish-subscribe and other enterprise service bus layers use multicast to push data to many receivers simultaneously
  - allows clustered appl servers to replicate state updates and heartbeats betw server instances and maintain coherent caches by invalidating or updating cached info on many nodes
- For scalability, IP multicast critical but deprecated (Amazon) as it lacks reliable packet dissemination, security, flow control and scalability in number of groups
  - IP MC addresses scarce and sensitive resources
  - If hw limits of MC addr exceeded, kernel bu decreases

For example, one researcher has suggested that the reason why these things are happening is because of some root causes. Why do not you attack root cause itself?

Let us try to understand the example of a root cause. Once as data center networks become bigger and bigger, it turns out that the software stack increasingly becomes one to many communication patterns, multicast take facebook or twitter all these things right. Basically, you are talking about one-piece information there has to be sent to multiple parties at the same time, multicast becomes very common. And it is also the case because if I update something in a replicated system update in one place, it has to be replicated the update has to be replicated to multiple places.

So, this multicast is a very natural phenomenon if you are trying to either because of the way the new communication patterns are showing up Facebook or twitter or because of availability on the reasons. Multicast is a very common requirement. So, essentially there is something called publish subscribe models. They push data to many receivers simultaneously. And also, as I mentioned earlier, they allow clustered application servers to replicate state updates, and maintain coherent caches.

So, it is the fact that you need multicast is I think fairly clear now, but the issue is that if you want to do this particular kinds of a multicast, right. It turns out while it is necessary it has certain problems. For example, it does not have flow control; that means, that it can essentially push the system into very unstable situations. So, IP multicast does not have flow control. You can try to understand why that is a case, but the ones we have flow control across such a multicast. It is very difficult because you have to manage multiple nodes all those things it is just to totally hopeless. That is the reason there is no flow control in this.

Similarly, you can have issues about security. It is (Refer Time: 52:16). So, basically for this reason big users like amazon for example, they tell you that you cannot use multicasting applications. They tell you mandatory it is not possible. So, on the one hand it is very critical, but it is impossible to provide that kind of support. It turns out to be if it is provided it will break the system someday. So, we are in a very peculiar situations where it is necessary, but you cannot use it. So, and it turns out the reason why this flow control problem comes in some people have suggested, is because the IP multicast addresses are scarce. Because you have small numbers of this IP multicast addresses.

Therefore, it turns out that if they get used up, a hardware limits of these multicast addresses get used up.

You finally, have to depend on some other facilities manufactured by the kernel for example. And because it is now down not done in the hardware it is now in the kernel, it turns out the system again can get over burdened. So, one of the things that people have succeeded is that, there are some problems why not increase your resources of for example, multicast addresses sufficiently. So, that you are always away from it is costly, but that is one way to handle. It is a lot of cost to it, but if you do a good job many of the problems that we talked about disappear.

So, this one way to do it through money at it, but it is not widely used.

(Refer Slide Time: 53:54)

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 (up to 90%) occurs even if message loss is negligible.

I will stop with one another example of the kind of problems, that we often face in large scale systems. Of course, storage systems also fall into this category. Sometimes you have if you are using multicast, they also have what is called oscillatory behavior. What exactly is oscillatory behavior? See nodes, we are talking about a large-scale system. There are many nodes. Each node is doing something. For example, it could be responding to some requests from a and therefore, it might be doing java execution. Java has something called garbage collection. Well, garbage collection is going on. Usually this it pauses. It can not do other things.

That means the node actually is for the time being not responding to certain things. Similarly, Linux scheduling delays could be there or somebody is flushing data to disk. All this things are can create certain disturbances in the system; that means, that while that node is disturbed, it may prevent the node from forwarding packets. Because the application thread does not respond, etcetera. So, what is the issue here? Because the node is not forwarding packets to downstream. But the upstream something is still coming in. So, the route is still continuing to send it. So, I still getting the packets.

So, finally, what happens is that if it is engineered well, I am going to send a response back saying, I am not able to send out something, but I am still getting too much stuff from a from my upstream guys. So, I will tell the upstream guy, please stop sending it. So, because of this the previous gather the upstream also face a same situation, now he is not able to send out things. But he is getting stuff from his parent also. So, finally, the whole of this network it the disturbance at one level travels all the way to the top.

So, essentially all the buffers on path from root to the node become full, and the roots sending throughput drops to very low values are 0. So, again once this particular thing that disturbance passes away. Again, they will slowly come back towards normal situation it takes time. So, it goes on oscillating between reasonable throughput and very low throughput keeps happening all the time. And actually, people are shown that if a very large tree let us say 10,000 to 60,000 nodes. And this is a actually a small system in currents current days for that.

I think google has for example, at least at least one or 2 orders of magnitude more number of nodes. In a very large tree, they are shown that when each node is disturbed, only for one second in an hour. We have so many nodes, each node is disturbed only exactly one second in one hour; that is, one in 3600. The throughput degradation can be as much as 90 percent, even the message loss is negligible. So, it is a dramatic drop even for something like one in 3600 kind of disturbance for each node.

So, that means, that your behavior of the system is somewhat problematic. You can not really say when some system is working, when it is over loaded, when it is dead, you really can not say any of these things. Because of this it turns out that your anybody using in timeouts in the system. They might sudden discover that system is unresponsive, because it is unresponsive you think it is dead, or you basically say that I will retry it hat

hopefully my packet has been lost somewhere, I will retry it and see if it goes through the second time. Somewhat actually I had more work to the system, and that actually keeps building it up and that might actually derive the system to some really a bad situation.

So, there are various reasons why you can not really say what is going to happen in the system. And so, your ability to guarantee availability or resistance to partitioning etcetera becomes that much weaker. We will continue from here in the next class.