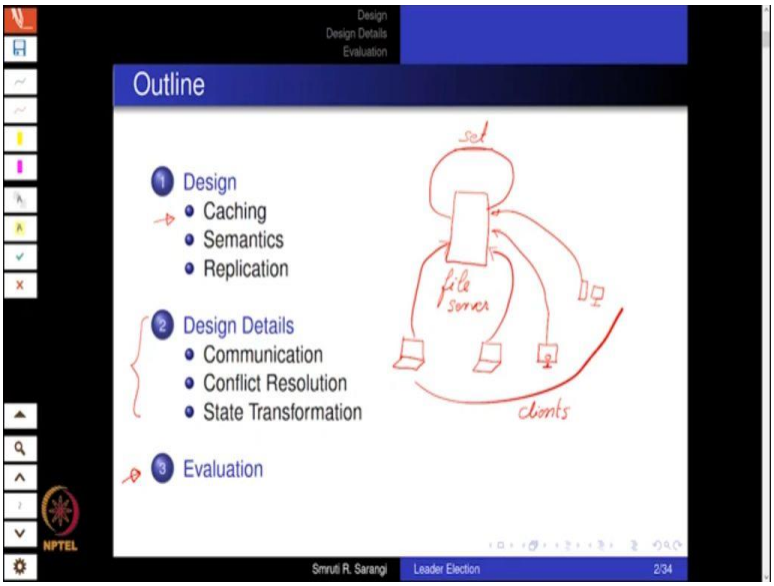
Advanced Distributed Systems
Professor Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Lecture – 17
Coda File System

We will discuss the Coda distributed file system in this lecture.

(Refer Slide Time: 00:23)



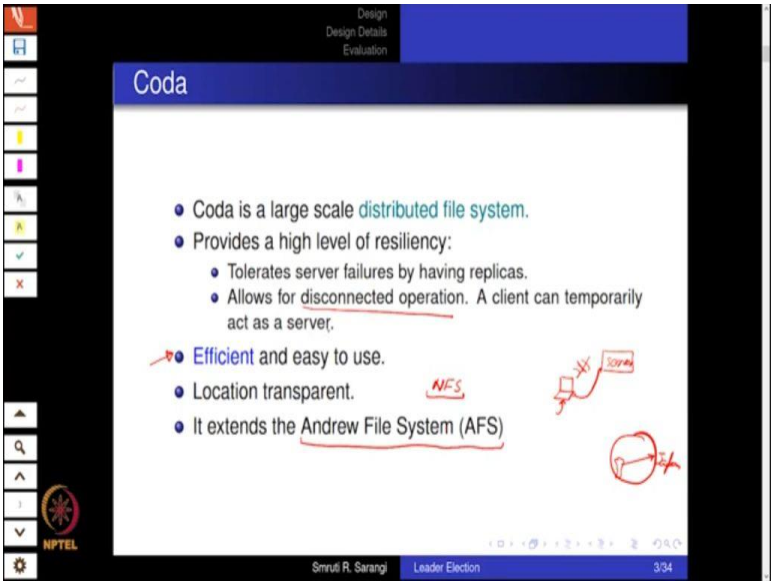
So, a distributed file system typically has a lot of client caches; so, we will discuss that in some detail. Then, we will discuss the semantics of the file system, finally replication. So, first we will discuss the broad overview of the design and then the details of the design, and give a little bit of an introduction to the evaluation. So of course, the detailed evaluation will be there in the paper. So, in this presentation we will just be covering the main points. So, a distributed file system is something like this. So, it is very much similar to what many of us would have seen in the universities that there would be a centralized file server; and there would be a set of clients.

So, client can be a laptop, a desktop does not matter; so all of them would be connecting to this file server. So, nothing would be stored on the laptops or on the desktops. So, these machines are also known as client machines; and the client can be really thin. So, these are also called thin clients; which means that is just just about a monitor with a very bare bones processor. So, the thin client can also connect with the file server. And the entire set of files will be resident resident on the server is just a small part will be brought in whatever the user wants to work on.

And after the user is done, they get transferred back to the file server. So, the advantage here is that I can do something on my laptop, I can disconnect it. I can then go to let us say some other machine that is I can go to a desktop machine; then I can connect that to the file server. I will immediately get back the same state of files that I had left when I was working on my laptop. So, it is like a large virtual hard drive that we want to implement. The same set of files seen across all of these machines.

So of course, all of these machines in this case are the clients; and we need not have a single server, so we actually can have a set of servers. And so they will be storing replicas of the files; so that if one of the servers fails, the rest of the servers can take over.

(Refer Slide Time: 02:58)



So, Coda is one such large scale distributed file system. So, it provides a very high level of resiliency, in the sense that it is not a single server; it is rather a set of servers. And it tolerates server failures by having a lot of replicas. So, the replicas ensure that if one server is down, the rest of the servers can serve the request to the client. And furthermore, it is possible. It allows for disconnected operations, which essentially means that if I have connected a laptop to the server; and let us say I decide that I am connected by a network connection; and let us say drop the wireless connection.

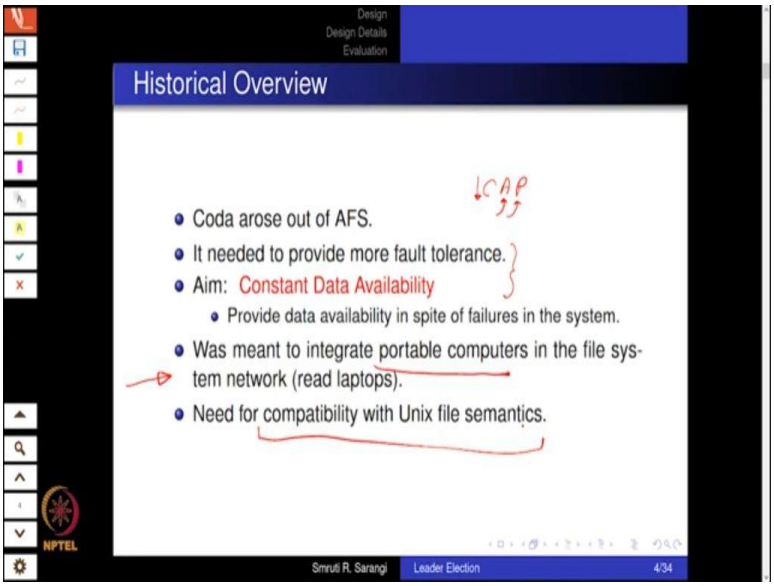
So, then I can still continue to work on my laptop with whatever files I have cached locally. And then once I reconnect back, so once let us say reconnect back, then whatever changes I made are

synced to the server; so it is rather efficient and easy to use. And of course, all file server systems have to provide something called location transparency.

So, location transparency basically means, it does not matter where you physically are. So, I can take my laptop, go to another; say this is the globe; I can start from the South America. And then sorry, I am a bad drawer; and then from there, I can go to Japan. And then I can plug in my laptop, and will get back the same set of files; so location does not matter.

So, there are two kinds of very basic file systems NFS and AFS. Most university campuses have NFS, where there is one central server. And all the home directories of students are mounted from NFS. The other is the Andrew File System which is also reasonably popular. So, Coda extends the Andrew File System AFS.

(Refer Slide Time: 05:09)



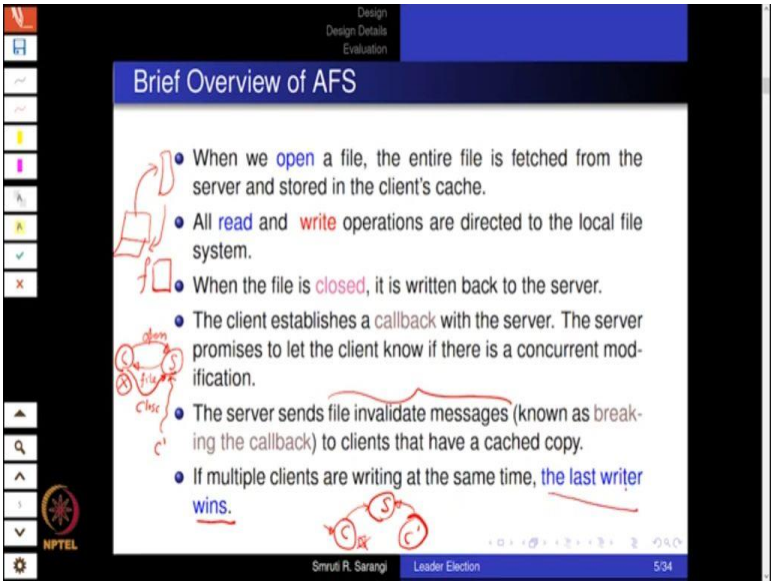
So, it arose out of AFS primarily because the AFS versions at that time did not really provide the amount of fault tolerance and data availability that Coda actually needed. So, what we know by the CAP theorem is that if we want reasonably high data availability, if we also have to want to be reasonably tolerant with partitions, then we our consistency has to come down. So, these are some of the difficult decisions that designers had to make when they were actually designing Coda; and so this was a kind of tough decision for them.

And we will see what were their decisions and what were the tradeoffs in the sense that they had to make. And one more need was that when this paper was being published, increasingly laptops

were becoming very popular; they were becoming very prevalent. So, what was really happening in the world of laptops is that people were connecting to the file server for some time, getting a dump of the files, a view of the files; and then they were disconnecting from the network, because network connectivity was not ubiquitous. And then they were working on their personal laptops.

So, managing portable computers was definitely one of the issues. And furthermore, there was a need for some degree of compatibility with existing Unix like file semantics. So, this was one more of the key design decisions that was made.

(Refer Slide Time: 06:59)



So, the first thing for us is to understand what is it that AFS provided and how did Coda extend that? So in AFS, what happens is if a client opens a file, the entire file is fetched from the server. So, which means that if I have a client over here, and then I fetch a file from the server; what comes to me is the entire contents of the file, the entire file that comes and it resides in the clients cache. All the read and write operations then are directed to the local file system; because the file in this case is cached locally. And then when the file is actually closed, then the contents of the file are sent back to the server.

So, what happens? So, initially if let us this is the client, this is the server; the client sends an open call to the server. So, what the open call does is that it returns the entire file. Also, what the client does is it establishes a callback with the server, which means that if there is a simultaneous

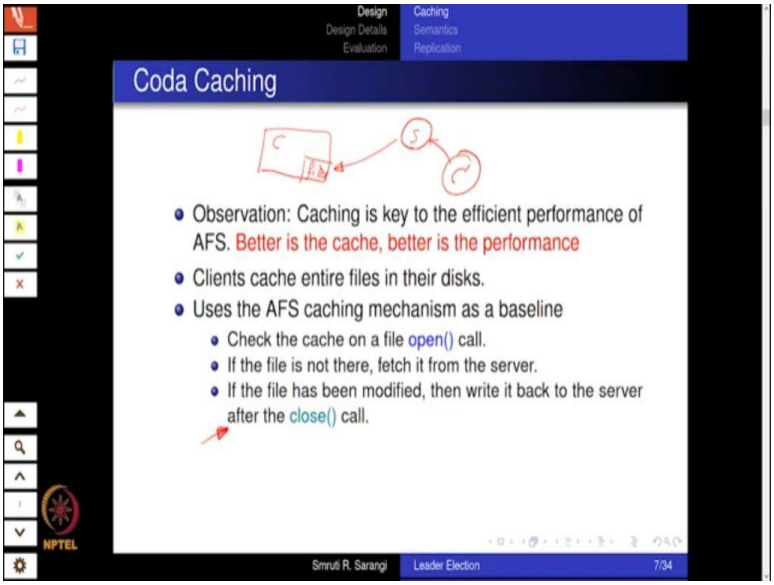
conflicting access from another client. Then, the server will let this client know that the copy of the file that it has is actually invalid; so there is a concurrent modification.

So in this case, the moment that the client C receives such an invalidate message. So, this process is known as breaking the call callback. What the client will do is that it will essentially discard the contents of the local file; so the operation that is going there.

So, let us first assume that the client has not opened the file; and so the client has not opened the file, then that is okay, I will just mark the copy as invalid. But if it is this case, that the client has opened the file, it is working on it; then, it will just keep on working as it is. And once it is done, it will send a close request.

So in this case, the semantics is that the last writer wins. Which means that if let us say two clients C and C' have opened the file at the same time and they are working whoever commits the last, whoever commits means, closes the connection at the end. The right of that file stays in the other write gets overwritten; so that is the last writer wins. But, assume that let us say this client has finished client numbers, the client C has finished its access; the local cached copy that exists, that copies discarded. But, if concurrent requests are alive at the same time, then of course the last writer wins.

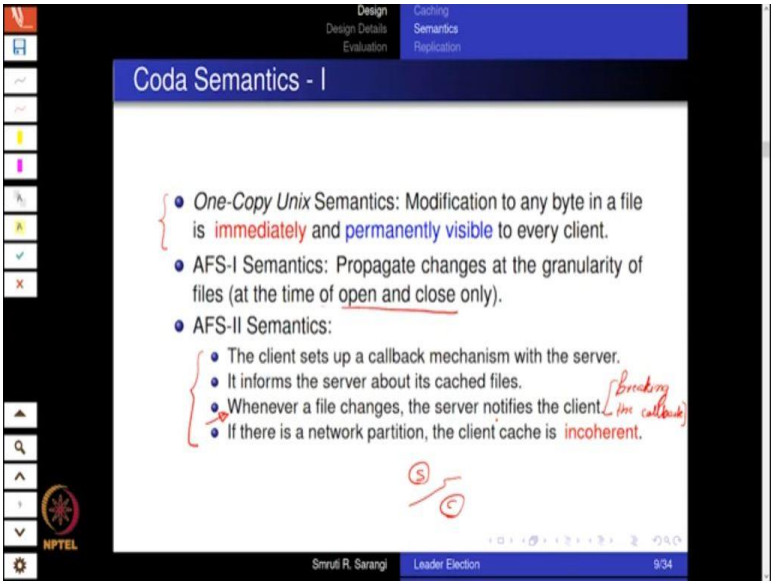
(Refer Slide Time: 10:12)



So, now let us look at the caching semantics. So of course in AFS, every client has a local cache; and in the local cache, it stores all the files that it has ever cached. So, a file is removed from the crash from the cache, only if it receives a callback broken message where it needs to invalidate a file. And why would a callback be broken? That callback would be broken at the server if another client writes to the file. So, then the server which in this case maintain state, let us client C know that a given file is not valid anymore. So, how does AFS manage the caching? Well, it checks its cache on a file open call.

If the file is not there, it fetches the file from the server. And if the file has been modified, then of course, you need to write it back to the server after the close call. So, this ensures that files can be dropped from the clients cache seamlessly; even if it is modified, it is not an issue. Because, after it is closed, it is in a sense not modified anymore; because the client copy and the server copy are the same. So in this case, the file can be dropped from the clients cache.

(Refer Slide Time: 11:34)

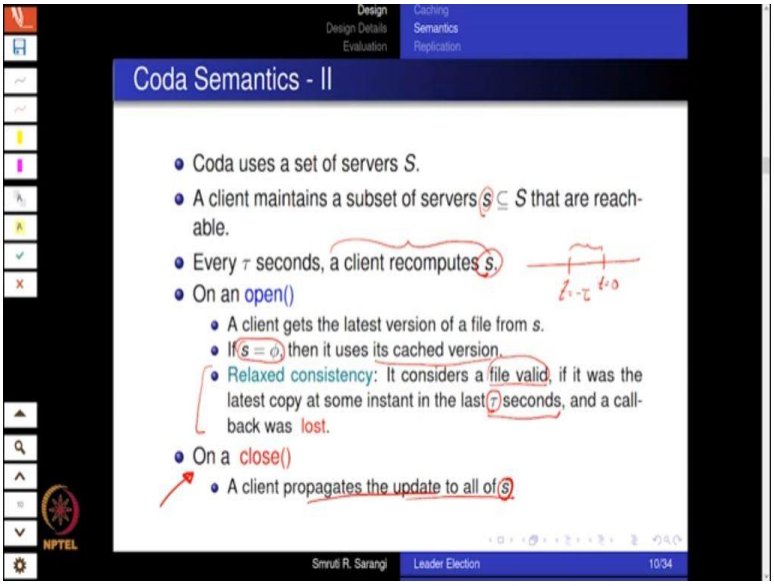


So, now let us look at the semantics of Coda. So, Unix has a semantics called the one-copy Unix semantics, which mean that modification to any byte in a file is immediately and permanently visible to every client. So, the one copy semantics in Unix is okay if you are talking of a single machine; but of course, it is not okay it is not acceptable for a distributed file system. So, we have the semantics of AFS-I, which propagates changes at the granularity of files at the time of open and close, which we just discussed.

AFS-II does something better, which we also discussed, which is that the client sets up a callback mechanism with the server; and the server is aware of the files that are cached to the client. So, whenever a file changes, the server will notify the client by breaking the callback mechanism.

And so this mechanism of letting clients know that look your cache files are not valid is called breaking the callback. If there is a network partition, of course, then the client cache remains in coherent; so that is the server is over here, client is over here. It remains incoherent and nothing can be done. So this as far as Coda was concerned, the Coda paper AFS-II was the latest semantics that it had seen.

(Refer Slide Time: 13:14)



So, now let us take a look at the Coda semantics. So, in a coda semantics, it uses a set of servers capital S. So, a client maintains a subset of servers S, which of course S is, as you can see small s ⊆ capital S. And this set of servers small s is the ones that are actually reachable; because note that all the servers that store replicas may not be reachable. Say, every τ seconds what the client does is it pings all the servers and re-computes the set S. On an Open message, what the client does is that it gets the latest version of a file from s.

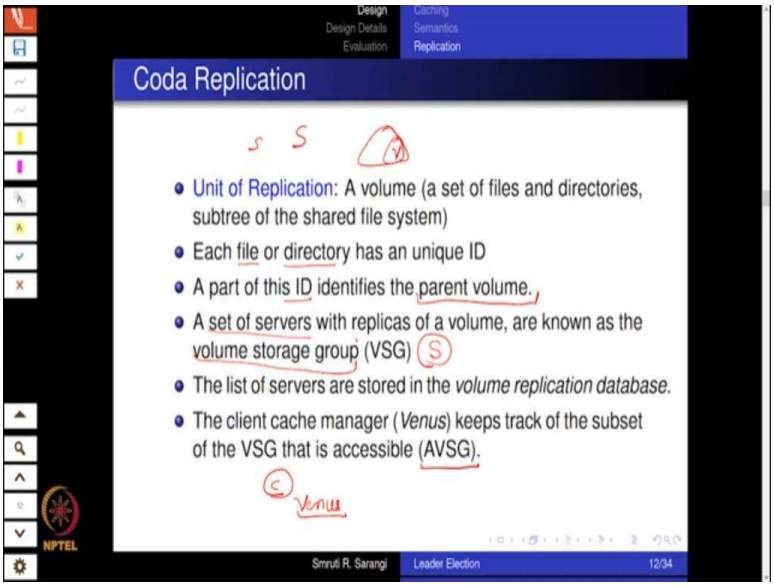
And so, we will see there is a way to ensure that a version is the latest version of a file. If of course if s = ϕ, which means that if it was not able to contact any server, then the client uses its cached version if it has one. In addition, what Coda actually introduced, so we discussed that availability and partition tolerance cannot be achieved with a strong consistency guarantee. So, consistency in a certain sense has to be relaxed; so Coda does that it relaxes to consistency. So, it considers a file valid, the contents of a file to be valid. If it was the latest copy, if the file was the latest copy at some instant in the last τ seconds.

So, what it does is that if I were to consider this as a number line, and let us see if this is like the current time t = 0; and let this be t = - τ. Within this time, at any instant within this interval, if the contents of the file were valid; and after that, so it does not matter, the files contents will have changed after that. But, if the callback messages, the callback breaking messages, informing the client that the contents are not valid if those messages were lost; we will still consider the copy

that the client has to be a valid copy. So, this essentially is a slightly a weaker version of synchrony, where we are allowing errors to scatter survive for τ seconds.

So, errors means stale copies of files. So, file is not considered to be stale, if it was modified in the last tau seconds; and the modification and the message informing the client about the modification was lost. Once a file closes, the client propagates the update to all of smallest; so all the servers are informed, notified, intimated about the update.

(Refer Slide Time: 16:28)



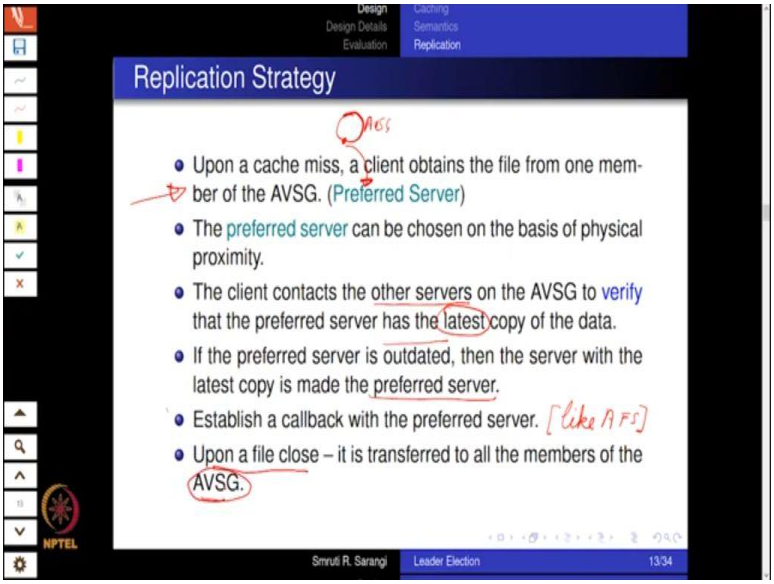
So, now let us discuss replication in Coda. So in Coda, we will not use s and capital S; since instead we will give them slightly technical names, technical terms. So, we will talk about the unit of replication, say volume which is a set of files and directories. Let that be a subtree of the shared file system; so, let us call that a volume. So, volume can also be the entire file system or it can be a large subtree of the file system; so let us call this a volume. So, each file or directory within a volume has a unique ID. So, this is also something that AFS introduced, that every file within a volume the moment it is created, is given a unique ID, which is allows us to identify it.

So, it is not it is path; it is a unique file specific ID. So, part of this ID indicates the parent volume; the volume it is a part of and the other is the file ID. So, for every volume, we have a set of servers, which store replicas of the volume. And this set of servers is known as the volume storage group, which can be thought as the set capital lists that we were talking about. The list of servers are

stored, of course in a volume replication database where we know what are the contents of the VSG. And every client, so every client has a cache manager; it is called venus.

So, this is also the same term that AFS used. So, what the clients cache manager will do is, it will keep track of the subset of the VSG that is currently accessible; that is currently fault free and that is accessible over the network. So, we will call this as the AVSG or the available servers in the volume storage group the AVSG.

(Refer Slide Time: 18:36)

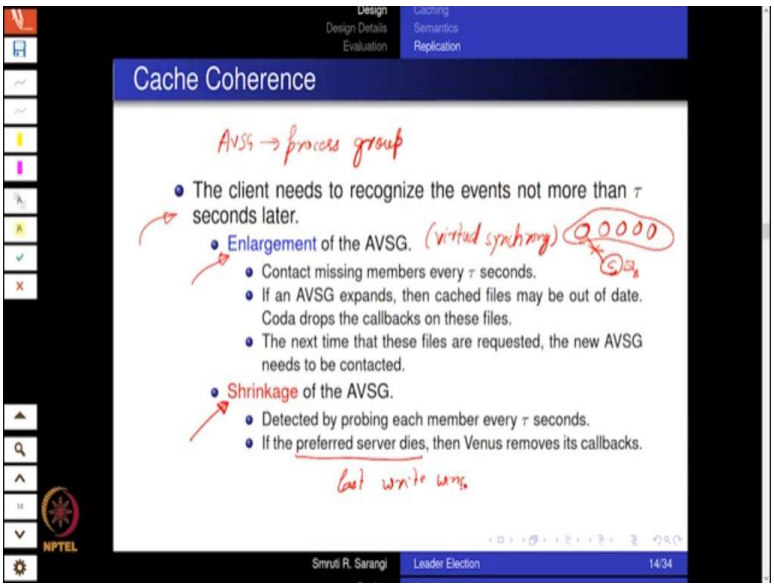


So, what is the replication strategy here again? So upon a cache miss, a client obtains the file from one member of the AVSG. So, the client what it does is that within AVSG, one of the members is designated a preferred server. We will see that this is a standard approach in the sense that many many proposals in this space in the in the world of distributed systems. Distribute the replicas among a set of servers; one among them is preferred, it is called preferred server. So, the preferred server in this case can be chosen on the basis of physical proximity or other considerations.

So, the preferred server is the one that is contacted the first. So, the client then contacts the other servers on the AVSG to verify that the preferred server has the latest copy of the data; so this has to be the case. So, of course, the preferred server is outdated. Then, the server with the latest copy is made the preferred server, so you change the preferred server. Then, similar to AFS like AFS, we establish a callback with the preferred server; which means that if there is a modification by

another client, then all the cache copies are invalidated. And similarly upon a file close, the contents of the file are transferred to all the members of the AVSG.

(Refer Slide Time: 20:28)



So, also recall that this idea of this AVSG is nothing but a process group, which is something that we studied in the lecture on virtual synchrony. And some very similar concepts will come up over here in the space of a distributed file system, where we will be talking about virtual synchrony. I mean something concept similar to virtual synchrony; and how they are that relevant over here.

So, since we have given ourselves a window of τ seconds, almost all the events in our system have to be recognized within the last, within τ seconds. So, one of the events could be enlargement or the AVSG, which means that some server that was not available, became available. Recall that we had a similar issue with virtual synchrony as well.

So, how do I detect that the AVSG has enlarged? Well, we contact missing members every tau seconds. If the AVSG expands, then it is possible that cached files maybe out of date. For example, it is possible that these servers are alive when a client is fine with them. But, then this server has a more recent update.

And it is possible that with this there might be a conflict. So, then what will happen is that for all of these files, Coda is pretty much going to drop the callback; which essentially means that if the AVSG expands, Coda will automatically assume that these files maybe out of date. And this means

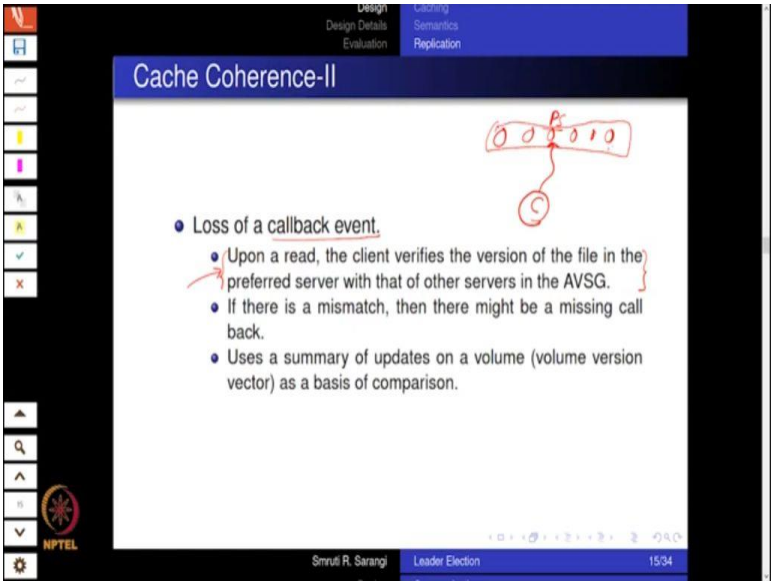
that if any callbacks have been registered in any server; those callbacks will be removed, they will be dropped. Either file is concurrently not being modified; then of course, the cached copy will be kind of thrown out of the cache. And next time the new AVSG, which is all of these servers will be contacted for the latest update.

So, one thing that you quickly need to note is that of course the idea of virtual synchrony can fit in very nicely into this. That of course, virtual synchrony's main idea is anything that happens in the view remains within the view. So, any message or any action that was happening in the previous AVSG that is stopped; and definitely, once the AVSG expands the cache, the callbacks are dropped, the cached copies are invalidated. It is not necessary to do all; but of course, some nuances possible here.

And then the next time that there is a request from the client, the new AVSG is contracted for the latest copy. Similarly, it is possible that the AVSG can shrink; so, this is also detected by probing each member every τ seconds. And of course, it is also possible that the preferred server dies. Say that happens, then of course the Venus client system will remove the callbacks from the preferred server; in the sense it will make a record that the preferred server will not send it callbacks; and it will try to work with the shrunk AVSG.

So, since the AVSG shrinking, there is really no great fear that an update will be missed. But of course, it is possible that another client might send an update to the preferred server. But of course in that case, since we have gone with the last right wins philosophy that will be taken care of.

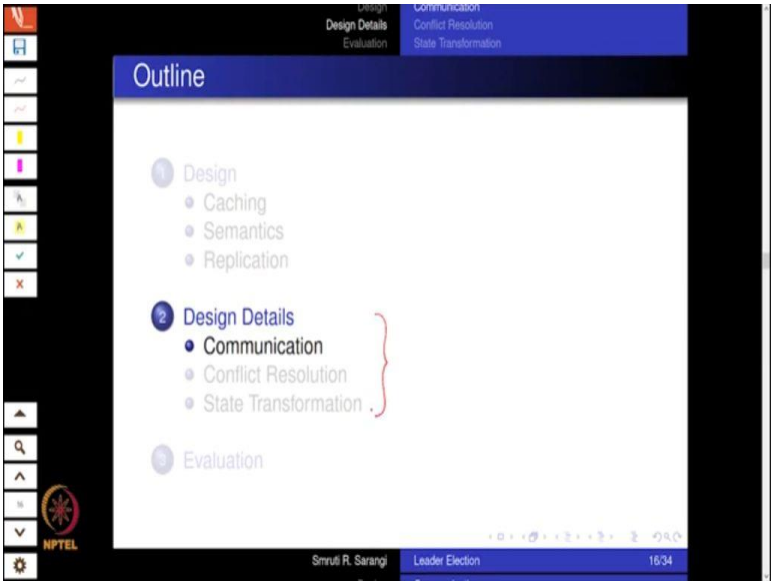
(Refer Slide Time: 24:21)

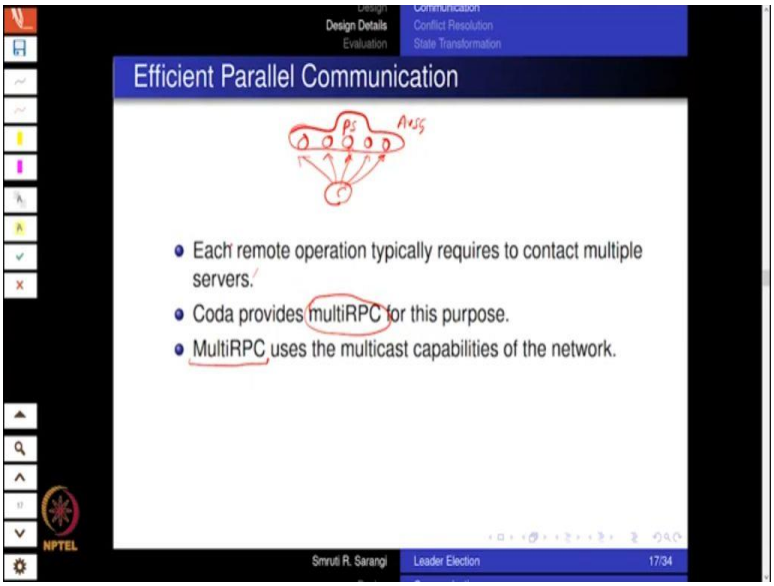


So, what happens if there is a loss of a callback event? Which means that a callback is kind of missing the message got lost. So, upon a read, the client verifies the version of the file in the preferred server with that of other servers in the AVSG; so, we have already seen this. So, what we have essentially seen is that out of all the servers in the AVSG, one of them is the preferred server. So, what the client does is that it contacts the preferred server to find if its updates are the latest.

Otherwise, if that is not the case, and of course it takes the updated verifies with the rest. If there is a mismatch, then most likely there is a missing callback. So then, there is a need for some degree of reconciliation. Then, we will see what the reconciliation is in the next few slides.

(Refer Slide Time: 25:29)



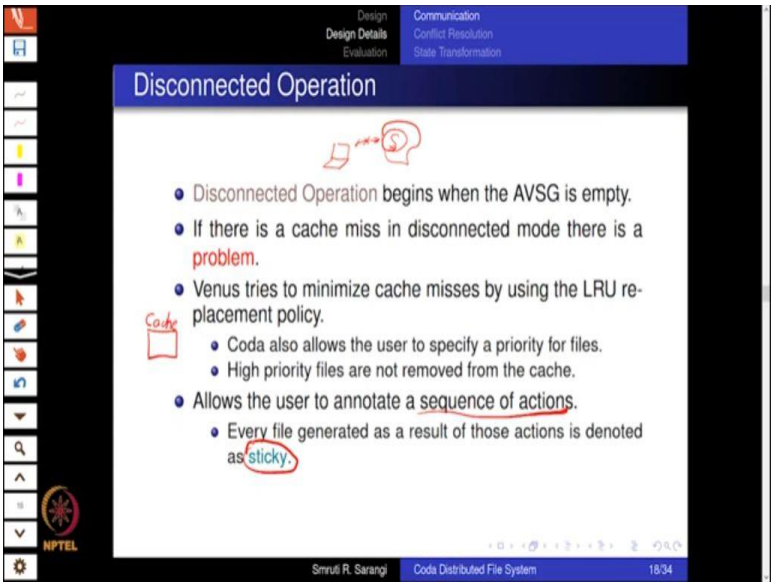


Now, let us discuss the communication aspect of the design details of how exactly this is done. So, given that we have seen the broad idea of the Coda system, which is basically we have a client; we have a set of servers. One among them is the preferred server; and the set of servers is known as the AVSG. There is a need that we need to read a version from the preferred server; and then verify it with the rest of the servers. And essentially verify it is the most latest up to date data. And the rights clearly have to be sent to everybody in the AVSG.

Of course, we also nuanced this position later; but at the moment, let us assume that it is sent to everybody. Can be sent to a quorum, but I will discuss that when we discuss the Dynamo paper.

So, Coda uses multicast RPC for this purpose instead of a unicast message. It relies on multicast messages, which most networks natively support. So, multiRPC is clearly more efficient, much much more efficient, far more efficient than a unicast method. So, this is where there is some advantage that we see from that accrues from the properties of the network; and will also measured this.

(Refer Slide Time: 27:08)

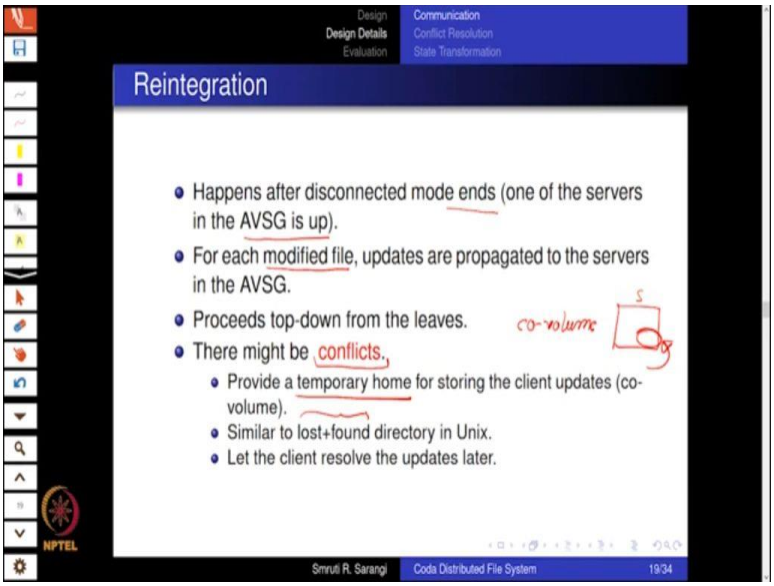


Let us now discuss disconnected operation, which is one of the main strengths of Coda. So, disconnected operation begins when the AVSG is empty, which basically means that the client is disconnected from the server. So, the link between the client and the set of servers all the servers rather in the AVSG is lost; so this is a problem. However, the main aim was that we should not perceive this to be a major problem in the Coda distributed file system. So in this case, the client cache manager Venus tries to minimize the cache misses. Why is this the case? The client has a cache, which is of course managed by Venus.

And what Venus tries to do is that it uses least recently used cache placement policy eviction policy, to ensure that all the files that are recently used are kept in the cache to minimize your chances of a cache miss. So, this is a software cache that resides on the client. Furthermore, Coda allows the user to specify a priority for files, so the high priority files are not removed from the cache.

In addition, it is also possible to annotate certain sequence of actions, such that anything that is produced as a result of these actions is marked to be sticky. For example, we can have a sticky open. And anything that is opened via this these mechanisms, gets the highest priority in the cache.

(Refer Slide Time: 28:53)



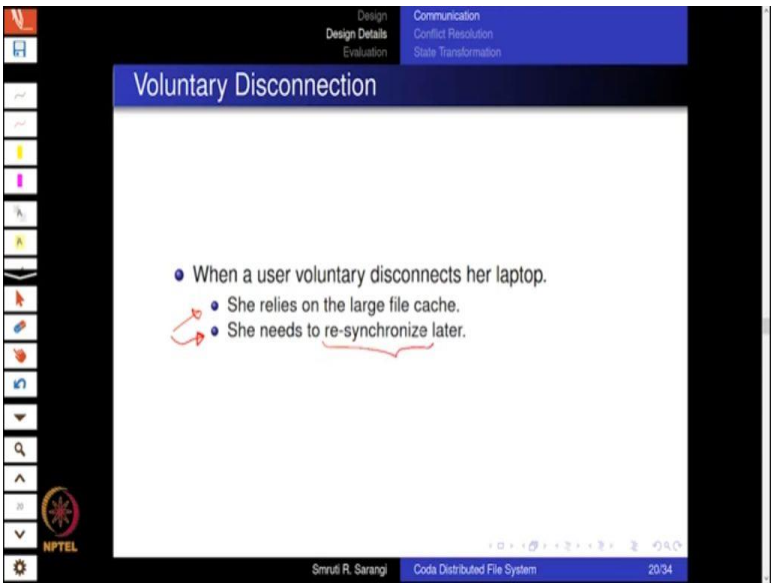
Once the disconnected mode is over, which means the client is reconnected; so, which means one of the servers in AVSG is accessible. For each modified file, that updates are propagated to the servers in the AVSG. So, the what the client does is, it takes a look at all the modifications; and it propagates these modifications to the servers in the AVSG. And in the cases for a file, of course it is easy; it is treated as normal. But, for directories, it is kind of hard; and so we will discuss this slightly later. But also another thing is that there might be conflicts; in the sense we might be dealing with concurrent writes.

So, in this case, what a server does is that it provides a temporary home for storing the client updates; so this is referred to as a co-volume, which is essentially a space within the server; where it is like one directory, where all the conflicting files are stored such that they can be reconciled later. This is very very similar to the lost plus found directory.

So, many of you actually take a look at your home directories, even on an NFS server; you will find a directory called lost plus found. This essentially refers to all of those conflicting files, which had some issue with the server; and they could not be automatically reconciled. And so in this

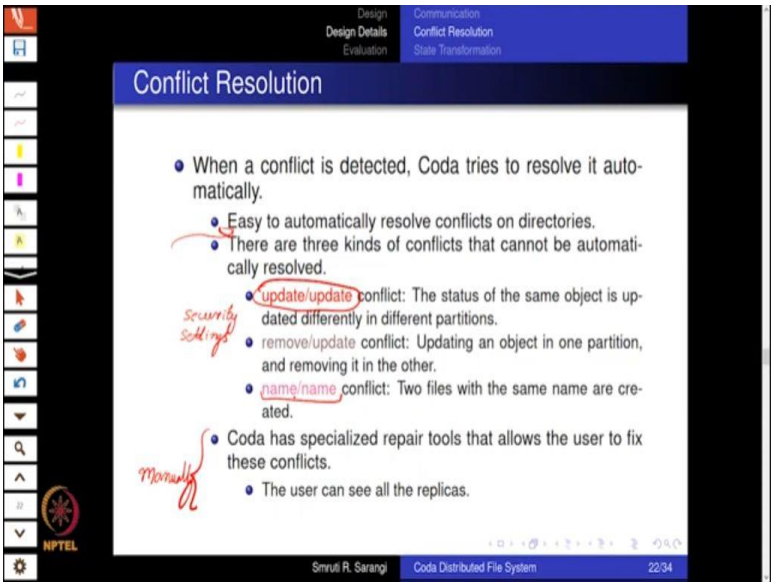
case, they are kept in the lost plus found directory. So, this means that the client can update, resolve all of these updates at a later point in time.

(Refer Slide Time: 30:48)



So, when a user voluntarily disconnects a laptop, she essentially relies on the large file size and then whatever writes that she does, has to be resynchronized at a later point in time.

(Refer Slide Time: 31:03)



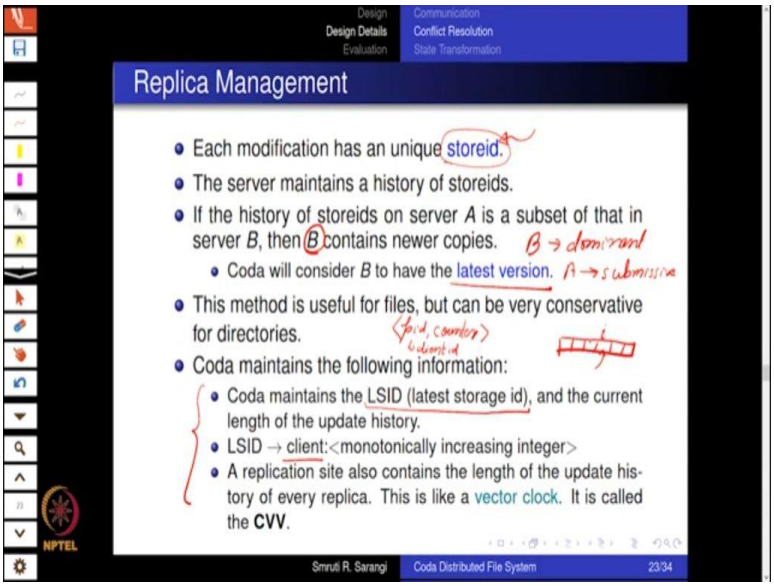
Now, let us look at the most important aspect of Coda, which is conflict resolution. So, when a conflict is detected, Coda will try to automatically resolve it. So, what we will see is that conflicts and files are easy to resolve; it is also relatively easy to resolve conflicts and directories. But, there

are three kinds of conflicts in directories which cannot be automatically resolved. One is let say update/update conflict. So, this is where the status of the same object is updated differently in different client partitions. For example, this can refer to the security settings.

So, in this case, it is possible that the access control of a certain file is changed in a certain manner on this client, and in a different manner in other client. So, this would be an update/update conflict in the directory. We might also have a remove update conflict, where file is removed in one partition and updated in the other. Also name/name conflict, where two files, of course, with different contents are created.

So, all of these things have to be kind of resolved manually at the level of the user. So, Coda has specialized repair tools that allows the user to repair these conflicts, fix these conflicts. So, the user can see all the replicas that are there on the different servers; and these can be manually fixed.

(Refer Slide Time: 32:58)



So, now let us look at the replica management. So, each modification that is made has a unique store id; so this is important. So, it is like immutable history. So, these are all very similar concepts as the original blockchain idea that we have discussed, that whenever any modification is made, it is essentially a new entry, and it is assigned a new store id. So, this is a very common idea, very common concept in distributed systems that any new update is given its unique new store id.

The server maintains a history of store ids. So, the history of store id is on server A is a subset of that in server B; then you can be sure that B contains new copies. So, Coda will consider B to have

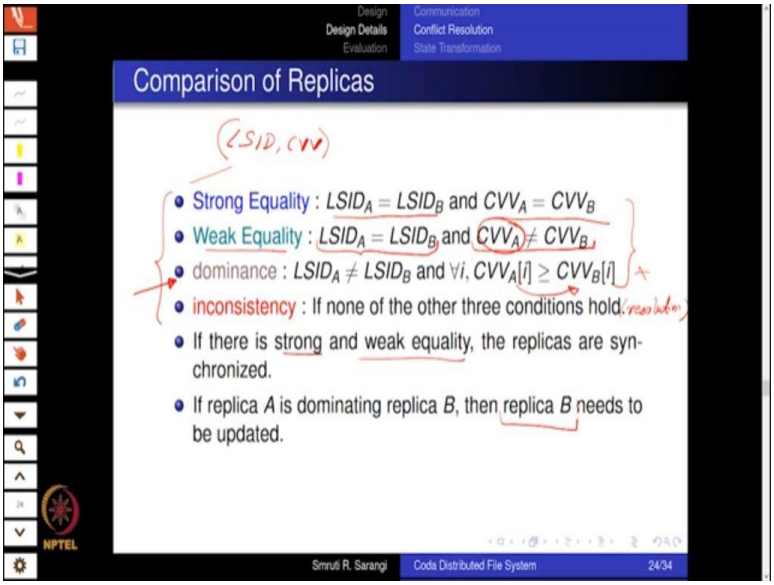
the latest version. So, B in this case is considered to be dominant; and A in this case is considered to be submissive. Why submissive? Because it does not have all the updates; and B have all the updates. So, the useful method for files, it can be reasonably conservative for directories; because of course in a directory, we might be modifying different files without an actual overlap; but Coda uses the same more or less for both.

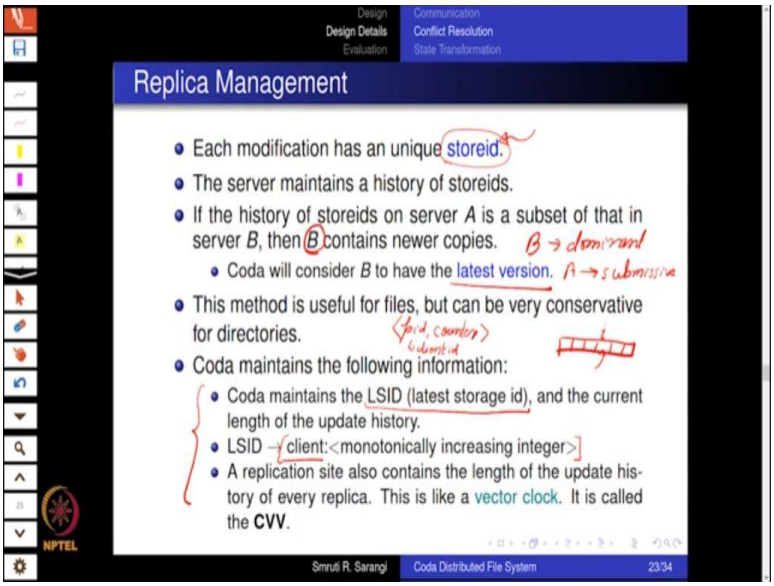
So, Coda maintains the following kinds of information. So, the first thing is that for every file, it maintains a latest storage id. So, the latest storage id is basically you can think of it as a scalar clock, which is a combination of a pid and a counter. So, this is clearly, it can be pid is like a client id and a counters; you can think of this as a scalar Lamport clock. And where every subsequent store id is guaranteed to be unique; and it is a tuple of the client id and a counter. And of course the, it contains the current length of the update history. And the LSID, as I just mentioned is a client, client id and a monotonically increasing integer.

In addition, the replication site contains the length of the update history of every other replica. So, this is like a vector clock, where basically if you recall, a vector clock is essentially a vector for each of these elements is a count. And for ith entry, this is an estimate of i's local clock; and the vector clock as we have seen is very important. It can be used to infer causality; it can be used to infer concurrency and all kinds of things.

So, this is a vector clock, which is called the CVV; it is a version vector. And so essentially with every object, it will maintain a latest store id, which is the latest version of the object. And also the number of updates that each site, each replicating site has actually made; and this information will also be there in this vector clock.

(Refer Slide Time: 36:44)





So essentially, we are looking at the tuple of LSID and the CVV; so, let me just go through this once again. The LSID is a tuple of the client id and monotonically increasing integer. So, from the perspective of the client, this is a scalar clock. In addition, the CVV is a vector clock from the perspective of the replica servers. So, here this contains the length of the update history of every replica; so this kind of tells the servers how recent their entries are. So, now when we are comparing an LSID CVV tuple for a given file, we can have one of these four conditions; so let us see.

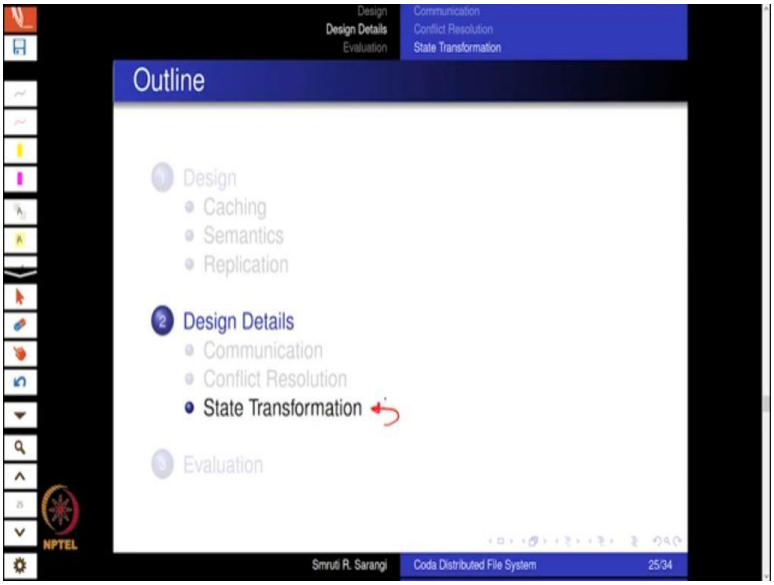
We can either have strong equality, which means that between the replica A and B, the LSID is the last store ids are the same; and the histories are also same. So, this basically means that both of the replicas are on the same page. So, their last store ids which are very much the last update that they have processed that is the same; and also they have the common view of the rest of the replicas, so they are equal. Then of course, we can have weak equality. In weak equality, the latest update that they have processed is the same; which means LSID A = LSID B.

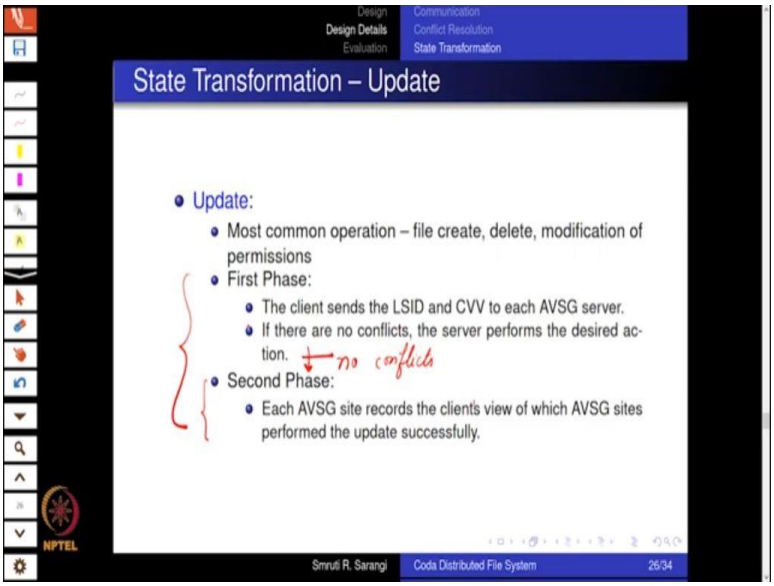
However, this part is important. Their view of the world, that CVV vectors are not the same; so, this is weak equality. So this can be that, they might have just missed some updates in the past; and then the replicas have not been updated, or some messages have been lost. So, that is why the CVV will diverge. But, the fact is that they are still, they still have the latest copy; because the LSIDs are the same. Then, we can have dominance; since dominance what will happen is of course, the LSIDs are not the same. But, the $\forall\, i,\ CVV_A$ [i] $\geq CVV_B$ [i].

So, this is a typical vector clock condition, which says that A is the more recent. A clearly has far more up to date information than B; because the vector clock which is CVV. For every single entry is greater than equal to the corresponding entry of B; so in this case A is dominating and B is submissive. In inconsistency is when three of these conditions do not hold; and this is where some kind of resolution is required. So, the resolution subsystem has to be invoked. So, if there is strong and weak equality, we can say that the replicas are synchronized, in a sense the replicas or rather both the servers.

Both of the replicas are fully the same, they are fully synchronized; and they have the most up to date data. But of course with the weak equality, one of the servers does not know enough about all the replicas; but that can be fixed. Say, if replica A is dominating replica B, then of course replica B clearly has old information which needs to be updated.

(Refer Slide Time: 40:21)



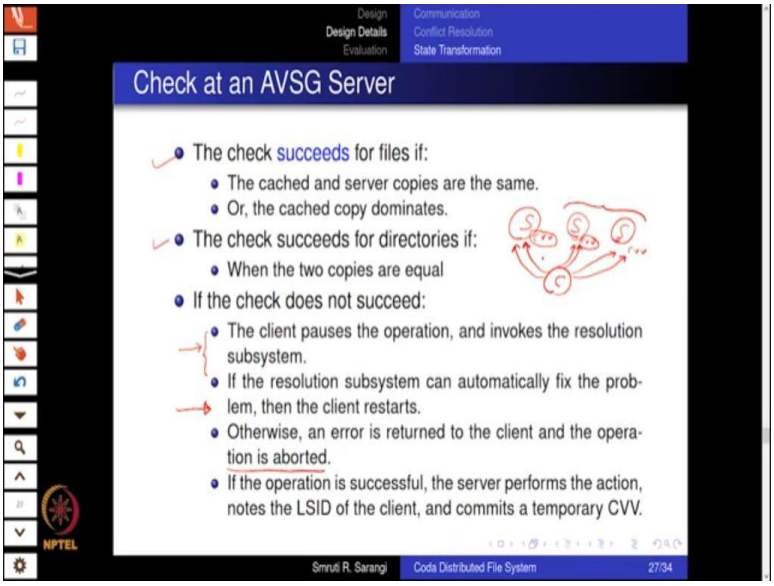


Now, let us take a look at the entire details of the protocol, and how the state is transformed via the state transformation logic. So, what are the most common operations in a file system? File create, delete and changing permissions, modification of permissions. So, a typical Coda file operation is broken into two phases; a first phase and a second phase. So in the first phase, the client sends the LSID and the CVV; so, whatever CVV it knows, and the last store id to each AVSG server. If there are no conflicts, so you do not expect conflicts generally; but you can have conflicts, if let us say there was a disconnection and another client updated the value.

So, then the server essentially makes a temporary record; so this is something like two phase commit. So, this operation is very much like two phase commit, where if there are no conflicts, the server performs the desired action and keeps a temporary record.

In a second phase each, so what happens is that the client. So, of course, from the first phase, it will go to the second phase, if there are no objections, if there are no conflicts. As I said, I drew an analogy with two phase commit, which is very apt in this case. So, then each AVSG site records the clients view of which the AVSG sites perform the update successfully; and the vector clocks, the CVVs are updated accordingly. So, let us take a look at far deeper look into the first and second phases.

(Refer Slide Time: 42:11)



So, let us look at the checks that are performed at an AVSG server in both the first phase and second phase; so this is for the first phase. So, the check succeeds for files if the cached and server copies are the same; and it is a same LSID or the cached copy is dominating. So, then clearly the check succeeds, otherwise it fails. The check succeeds for directories clearly when the two copies are equal. So, this we have been arguing it is conservative, but Coda does follow this because otherwise we need to maintain a lot of state.

Now, let us see what happens if the check does not succeed. So in this case, the client pauses the operation and it invokes the resolution subsystem. This tells the client that pretty much it has a missing update or it has a stale update. So, the resolution subsystem tries to automatically fix the

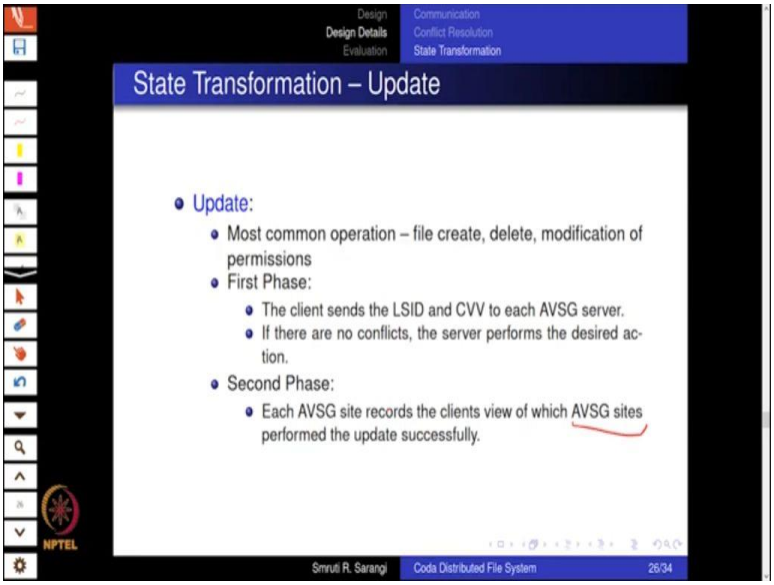
problem, which means fetch those updates that the client did not have and tried to perform a merge. If the merge is not successful, then an error is returned to the client and the operation is aborted; it is stopped over there.

However, if the client has an up to date copy, then what happens is that all the servers perform the action; and the note the LSID of the client and commit a temporary CVV. So, it is important to understand that if the multiple servers and the client is over here. So, in phase one what happens is that the client sends the version to all the servers. And the servers then indicate if they are okay or not to go forward, or go ahead with update; and they also note the temporary CVV that the client is proposing. So there is the CVV, the latest CVV of the data that the client has; so this is phase one.

So, in phase two, what happens is that after the client has gotten the responses from the entire AVSG and if there is no error, if there is no conflict; then, the client sends a commit message informing the servers that they are ready to perform the update on their site. And also each server is made aware of the rest of the servers that have agreed to perform the update.

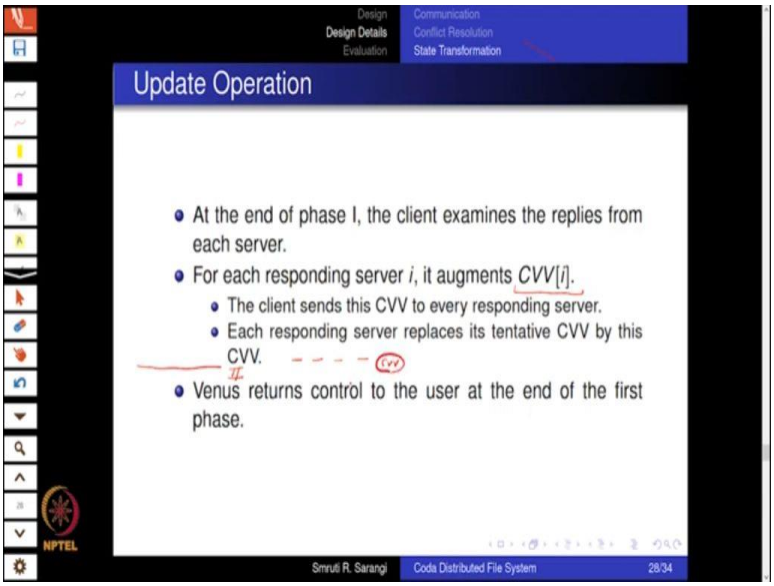
So, then what happens is that the server increments its own vector clock as well as the vector clocks of the the entries corresponding to the rest of the servers. So, this ensures that by the end of this process, by the end of phase two; the entire AVSG, they have the same CVV further data or the same vector clock.

(Refer Slide Time: 45:25)



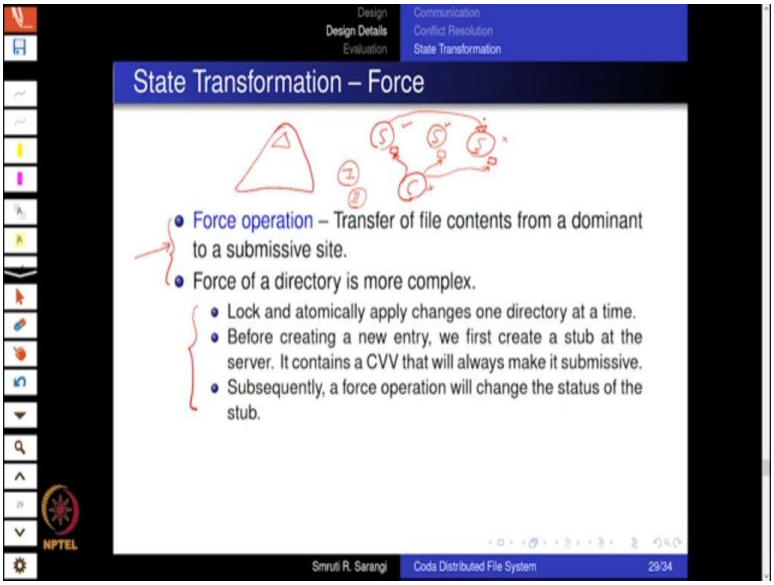
And why is this is the case? This is a case because in phase two, if you would see over here, it records the views of each of all the AVSG sites that have performed the updates successfully. So, all of the AVSG sites at least that perform the update successfully are guaranteed to have the same value of the vector clock, which is the CVV in this case, a further explanation of this.

(Refer Slide Time: 45:52)



So, each responding server augments CVVi and the client sends this to all the servers as we just discussed. And in phase two, the tentative CVV is replaced by the final CVV. And so the aim is that by the end of the phase two, the AVSG has updated data; as well as it has the updated CVV for the data, and all the replicas pretty much have the same CVV after the end of phase two.

(Refer Slide Time: 46:31)



So, if there is a conflict, then the Venus file system the client file system manager does a force operation, where it tries to attempt a merge, particularly transfer file contents from a dominant to a submissive site. And so in this case, the submissive site can either be their client or it can be a server; so it does not matter, since we have a full version history. Whoever does not have the updates, that is a submissive site; and updates need to be transferred. However, doing a force operation for a directory is far more complex. So, we need to lock and atomically apply changes one directory at a time; and so an entire directory has to be locked and done.

So, basically, this is kind of a hard operation; but, because, it is possible that a directory tree can be really deep and this has to be done recursively. So, that is the reason we do it one directory at a time; we do not do the entire tree at a time. So, if let say a new entry has to be created. What we do is we first create a stub at the server which is guaranteed to be submissive; because its CVV can pretty much contain all zeros. And subsequently a force operation will change the status of the stub, which means give it all the updates. So why is this done this way? This is very interesting.

So, if we have a client and we have a set of servers, and let us say we are trying to create a new file in the directory. What I am trying to say over here is that we first create a stub in the directory for the file, which is pretty much a file with that name; but an empty entry with no contents and which is guaranteed to be submissive to every other update. So, once the stub is created, so this is so consider this as phase one. In phase two, what happens is that all the updates if the client has a

dominant copy that is are transferred to the stub. And there is a very similar process that is followed between one server and the rest of the servers.

So, an astute viewer can ask that why not do both of them in the same go at the same time. So, what would that mean? That would imply that the entire contents are transferred to the server in one go. The problem is that if there is any node or network failure, then we might have the file in an inconsistent state. Or some of them will have an update and some of them will not have an update. It is possible that maybe this and this server have the contents of the file and this server does not.

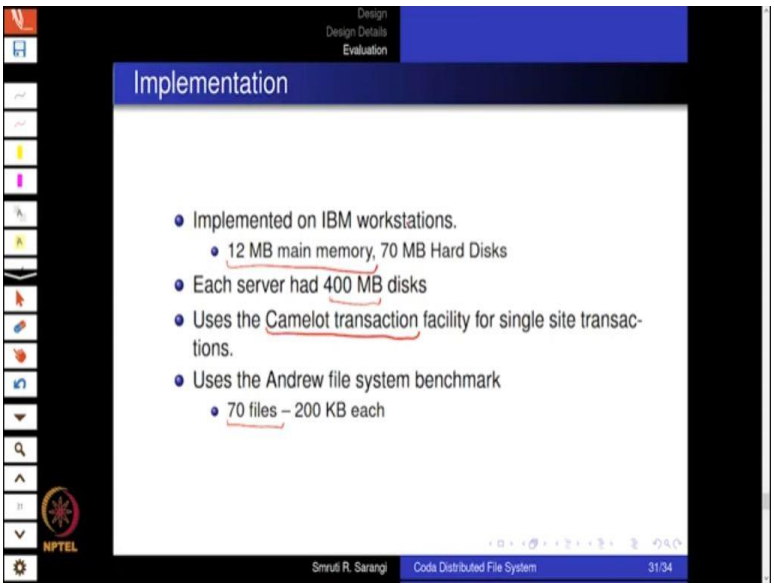
So, to stop that what we do is we just create a stub entry that at least indicates to the server that look a file with this name exists. And then in the second stage, all the updates, so creating the stub is very quick. These are very quick short messages that can be sent in a single RPC. Once all of them have it, the process of transferring the file can then be done later. And even if let us say this process gets interrupted, then at a later point of time if this server has all the updates; it will see a submissive entry and transfer the updates to this server. So, this increases the reliability of this, the robustness of this process.

(Refer Slide Time: 50:14)

So, we also have repair and migrate operations; so repair operation is used to fix inconsistent updates. So as we said, whenever an inconsistency is detected, the file is moved to a co-volume. And so in this case, they are stored there for the user to manually take a look; so all accesses to

inconsistent objects fail. And so basically, in this case, we wait till the conflicts are actually resolved.

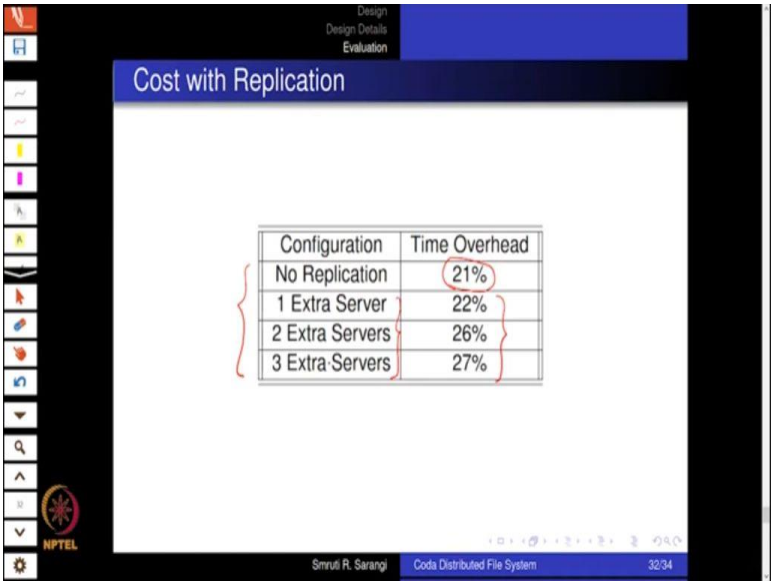
(Refer Slide Time: 50:57)



So, coming to the implementation, this was implemented on IBM workstations. So, when it was implemented, 12 megabytes of main memory was a lot; but of course nowadays it is nothing, even smartphones have much more memory than that. But for those days, 12 megabytes of main memory and 70 MB of hard disks was all that was there.

And so this is what the workstations had and the servers had 400 megabytes of disks. And the Andrew file system benchmark was used, which is essentially 200 KB, each 70 files. And there are a set of operations read, write, read directory, scan directories; it is a mix of operations. And for single site transactions, the Camelot transaction engine was used to ensure that operation on a single site happens atomically.

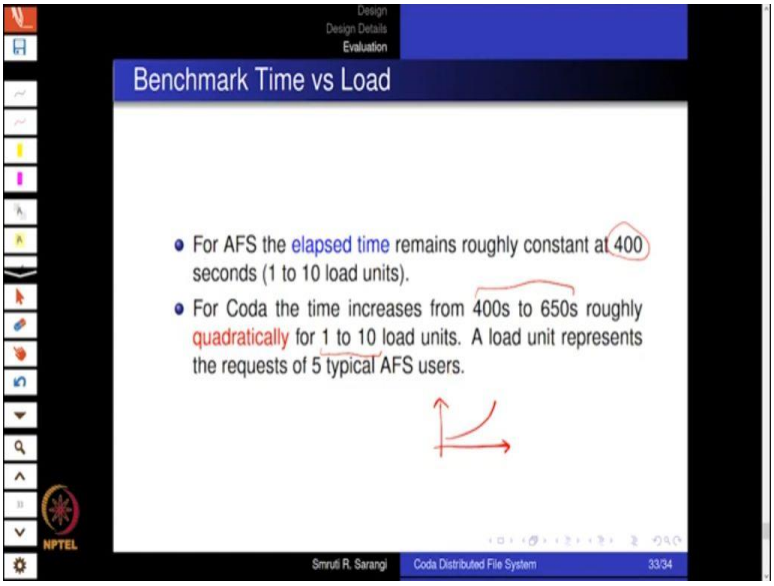
(Refer Slide Time: 51:52)



So, four configurations were studied no replication, which means there is no additional servers. It is like this is a baseline AFS overhead. So, the time overhead per operation just increased by 21 %. So the nice thing about coda, which was really appreciated at that time; and this is why this paper remains a classic, is primarily because the additional overheads with the extra servers is a kind of minimal. So, it increases to by 1 %, 4 %, and 5 % respectively, with additional servers.

And the main reason is that we use an RPC based mechanism; updates can be lazy, we maintain versions. So, the critical path in a certain sense is actually not LinkedIn. And since many of these operations are not really very network dominated; this does not turn out to be an issue.

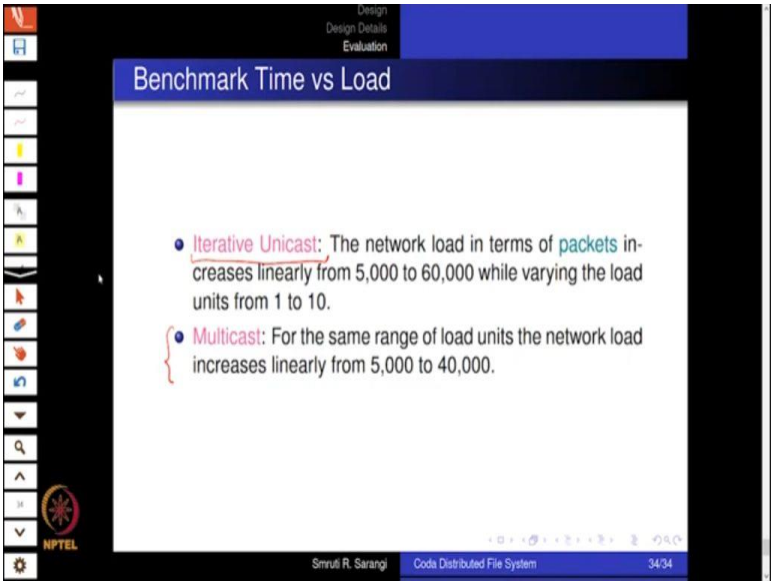
(Refer Slide Time: 52:56)



So, for AFS, the elapsed time for for different load units; so, where of course we are defining a load unit as the unit, the load generated by five users. So, basically, the total time for running the benchmark was roughly 400 seconds; and this increases to 400 to 650 seconds as we have seen across loads for 1 to 10 Load units, it increases quadratically. So, it is not a linear increase like a quadratic increase like that.

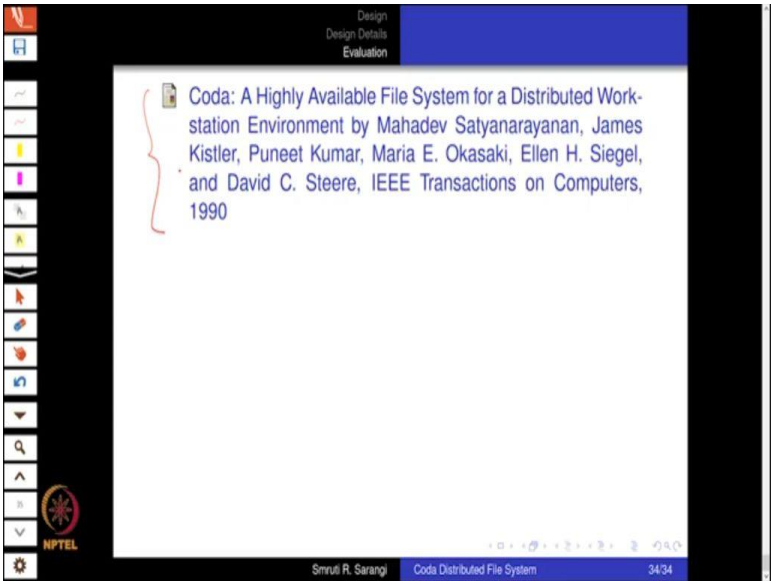
So, as we increase the load, Coda does increase super linearly. And the main reason being that it puts a higher load on the network synchronization, server management, version management, migration, et-cetera. But nevertheless, this is a very scalable file system; and this is why this paper has remained a classic.

(Refer Slide Time: 53:57)



Then, we will discuss the effect of multicast and unicast. So, if unicast is used in an iterative fashion, the network load in terms of packets increases linearly from 5000 to 60,000 across lower units. And for multicast, of course the number of packets reduces. And I would advise viewers to take a look at the paper; so, the paper also talks about latencies and so on. So, in general, while designing a distributed system, it is a good idea to use the multicast and broadcast capabilities of the network.

(Refer Slide Time: 54:40)



So, this was the reference IEEE Transaction on computers 1990, a very old paper. But, this paper has inspired successive generations of distributed file systems, and many of today's file systems or much of their origins to the Coda paper.