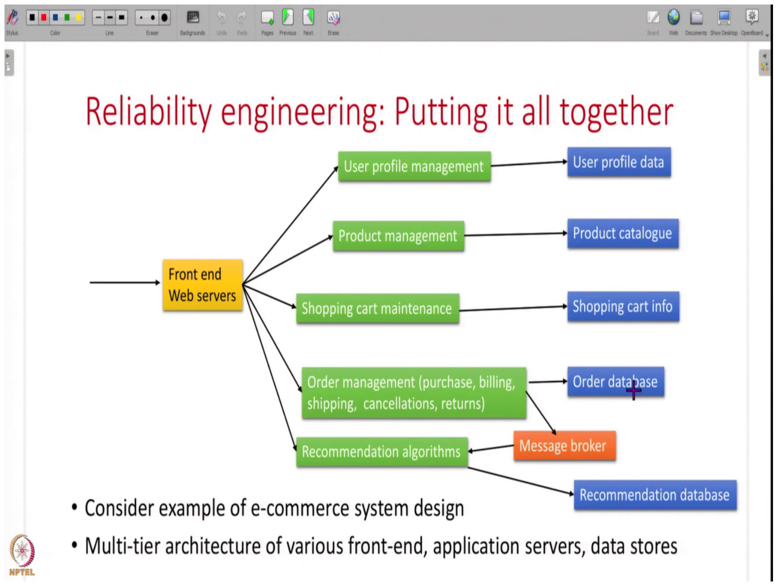
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 55
Case Studies of Distributed Systems Design

Hello everyone, welcome to the 40th and the last lecture in the course, Design and Engineering of Computer Systems. So, in this lecture first, we will try to put together all the concepts we learned this week to see how we can design reliable systems. And we will also try to wrap up and summarize everything we have learnt in this course. So, let us get started.

So, of course, throughout this course, we have been discussing examples of computer systems, specifically, I have taken the example of an e-commerce systems several times in the course, you know, because this is something of course practical, and you all would have come across it interacted with an e commerce website in your real life.

(Refer Slide Time: 0:57)



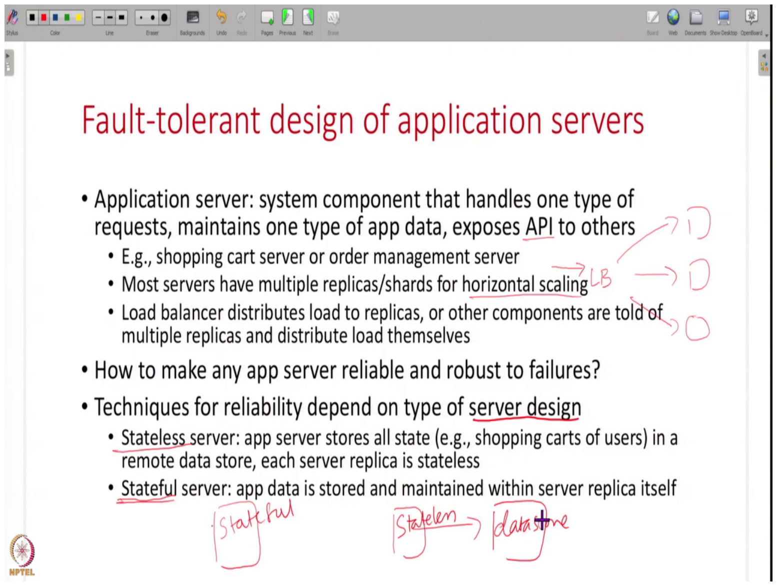
Reliability engineering: Putting it all together

- Consider example of e-commerce system design
- Multi-tier architecture of various front-end, application servers, data stores

So, if you look at it, this was the modular design of an e-commerce system that we studied a few weeks back, you know, there are a bunch of front-end web servers. And depending on the type of request, if it is for user profile, or searching for products, or shopping cart or orders, recommendations, the front end will contact, different application servers, each of which is responsible for a certain kind of requests and you know maintaining a certain kind of data and it will assemble all of these pieces together and return the webpage that you as a user of the e-commerce system will see.

So, this is a multi-tier architecture, you know, there is a front end, there is the business logic, the application layer, there is the data layer, the back end, data stores, and all of that. So, this is an example that we have seen before now, for this system, let us try and apply these concepts of reliability engineering of replication, consistency, all of this that we have learned this week, let us try and apply it to a system like this and understand how it will look like.

(Refer Slide Time: 2:04)



Fault-tolerant design of application servers

- Application server: system component that handles one type of requests, maintains one type of app data, exposes API to others
  - E.g., shopping cart server or order management server
  - Most servers have multiple replicas/shards for horizontal scaling
  - Load balancer distributes load to replicas, or other components are told of multiple replicas and distribute load themselves
- How to make any app server reliable and robust to failures?
- Techniques for reliability depend on type of server design
  - Stateless server: app server stores all state (e.g., shopping carts of users) in a remote data store, each server replica is stateless
  - Stateful server: app data is stored and maintained within server replica itself

So, if you have an application server, you know, it can be anything, it can be front end, it can be in the middle, it can be a data store, whatever any server in your system, usually handles certain kinds of requests, exposes some API, you know, maintain some data. And on this data, there will be the request coming in, that is what any server in a system at a high level of abstraction, this is what a server looks like.

For example, it could be a shopping cart server that is, you know, maintaining all the shopping carts of users and requests come to add items, delete items, view, the shopping cart, and so on, it could be a checkout server that is, you know, getting request to manage the particular order, and it will do the billing shipping and so on. In this way, any system will have multiple different application servers. And all of these servers will typically have multiple replicas or shards for horizontal scaling, you know, in real life large system, it is very unlikely that a single machine can handle all the load that is coming at you from all over the world.

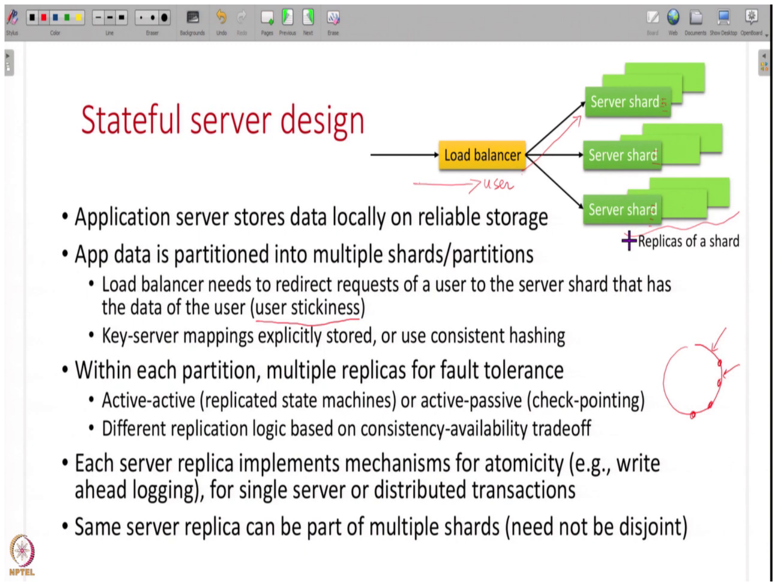
So, you will typically have multiple replicas or shards of the system to handle the load. And then of course, there will be something like a load balancer that will distribute traffic to these different replicas either you have all traffic comes to the load balancer and the load balancer will split it to these shards or you can tell your previous component that look there are all of these shards, you know, you can tell the front end that there are three replicas of the applications or that itself will distribute traffic, either way the traffic gets distributed to these different shards.

Now, you have a system like this and now how do you make this system reliable and robust to failures by applying the concepts that we studied in this week. So, let us you know, put all of that together in the context of this system. So, what techniques we use will depend on the type of server design. So, one thing to understand is, there are two ways in which you can build application servers, one is called a stateless design and the other is called a stateful design.

So, what is a stateless design is an application server stores all of its data in some other data store in the backend and there is nothing stored inside the server, any request the server has to get the data from the data store process the request store the data back. Within the application server itself, there is no data stored. So, if it crashes nothing, no information is lost, that is a stateless server design. So, the other is a stateful server design where the application data is stored within the server itself in memory or on disk or so on.

Where you know, no external database is used, but the data is stored within the server itself. So, in this way, you can either have stateless design or stateful design and usually you will have something in between you know, you some data is stored remotely, some is stored locally and so on. And this is a general design decision you have to make when you are building servers that you want each of these components to be stateless or stateful. Now, in each of these cases, your reliability techniques will differ.

(Refer Slide Time: 5:12)



Stateful server design

- Application server stores data locally on reliable storage
- App data is partitioned into multiple shards/partitions
  - Load balancer needs to redirect requests of a user to the server shard that has the data of the user (user stickiness)
  - Key-server mappings explicitly stored, or use consistent hashing
- Within each partition, multiple replicas for fault tolerance
  - Active-active (replicated state machines) or active-passive (check-pointing)
  - Different replication logic based on consistency-availability tradeoff
- Each server replica implements mechanisms for atomicity (e.g., write ahead logging), for single server or distributed transactions
- Same server replica can be part of multiple shards (need not be disjoint)

So, let us start with a stateful server, if you have a stateful server design, you know, application server will store all the data locally within some disk or hard disk or some such mechanism within the server itself. And then in such cases, your application data what you lose, you will partition it across these multiple shards. For example, if there is a checkout server or say a shopping cart server, you know, some shopping carts are stored here for some user, some shopping carts out here, some shopping carts are here, you will partition the data across these servers.

And then the load balancer will somehow know how this partitioning is done, say consistent hashing or however this partitioning is done, the load balancer will know that information. And whenever a request comes that the load balancer will compute for this user's request, which shard has this user's data, it will compute that and it will redirect the request to the appropriate shard to the appropriate slice of the server.

That is your load balancer must do what is called user stickiness, we have seen this concept before you know any request that comes up, you cannot randomly send it to any shard. Why? Because you have to send it to the shard that has the data of that particular user of that particular shopping cart. Only then will the request be handled correctly. That is you need some kind of users stickiness. And how does a load balancer know these mappings? You either store them explicitly or you use consistent hashing depends on the policy of the load balancer.

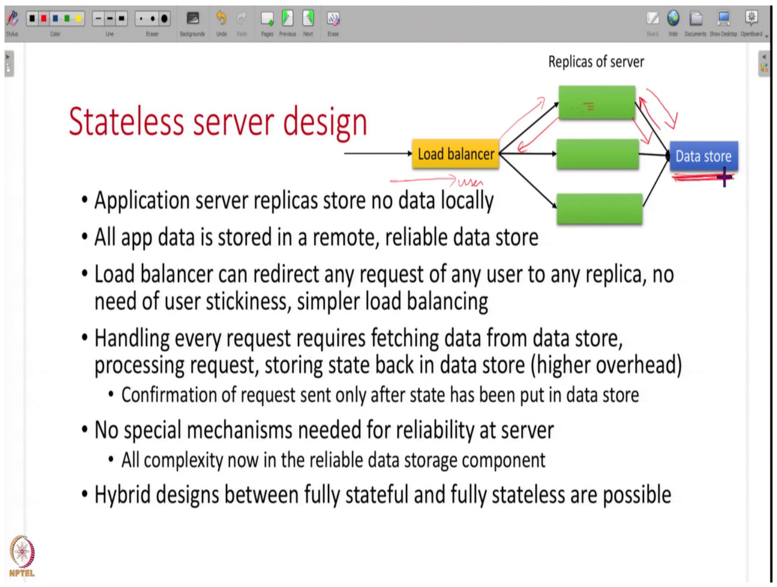
So, now you have partitioned application data into shards. Now, within each partition, you might need fault tolerance, you cannot just store all the shopping carts in just this one server. What if the server crashes, you cannot lose all the shopping carts. So, therefore you will have multiple replicas of each of these partitions. So, you first have partitioning, then you have replication within each partition, it can either be active-active, active-passive, whatever. And, you know, how do you replicate at a majority, do you replicate at a minority depending on how much consistency you want all of that stuff we studied here works across these replicas.

So, this is how you visualize a system, so every request is first redirected by the load balancer to a shard, within a shard across the replicas of the shard, it is replicated according to the consistency model, according to the desired consistency requirements, and then the response is sent back to the user. And also, within each of these servers, you know, within each of these replicas, you also have to think about atomicity whenever you are executing any operation within a server, if this server fails is the atomicity preserved, you know, do some write ahead logging and so on. And across these suppose a request requires changes at multiple servers across the servers atomicity, you will use distributed transactions.

So, all of the concepts we learned, you can see how they all are being applied in a real-life system. But more than one thing to note in this figure, somehow these all look like different different, you know, disjoint sets, you know, these servers are different from the servers, but you can have some overlap. For example, a same server can belong to multiple different shards, it can be a primary in one backup in one and so on.

For example, if you have this consistent hashing, you know, a key that hashes at this point in the ring is stored at these three servers. A key that hashes here is stored at these three servers. In this way, you will have overlap between these different shards. Also, they are not disjoint. As shown in the figure, I have shown them like this just for simplicity. So, for a stateful server to summarize, you will partition the data across the shards and the data in each shard will be replicated across multiple replicas for fault tolerance, your load balancer has to you know, keep this partitioning in mind and do some stickiness in its load balancing strategy.

(Refer Slide Time: 9:09)



Stateless server design

- Application server replicas store no data locally
- All app data is stored in a remote, reliable data store
- Load balancer can redirect any request of any user to any replica, no need of user stickiness, simpler load balancing
- Handling every request requires fetching data from data store, processing request, storing state back in data store (higher overhead)
  - Confirmation of request sent only after state has been put in data store
- No special mechanisms needed for reliability at server
  - All complexity now in the reliable data storage component
- Hybrid designs between fully stateful and fully stateless are possible

Next is a stateless server suppose your server replica does not maintain any state, you know, all the state is stored in a data store. In such cases, your system design becomes significantly simpler you know, there is no data stored here. All the data is stored in a remote data store. So, now load balancer, whenever it gets a request from any user, it can redirect the request to any replica does not matter because there is no state with anybody anyways.

So, requests can go to any replica. And that replica to handle this request, it has no state of the user says add an item to my shopping cart, this replica does not know what is the shopping cart of the user looking like at this point. So, therefore, for every request, it has to go to the data store fetch the latest state of the user, then update that state then again, store it back in the data store, and then send a reply back to the user.

So, handling every request you will first fetch, contact the data store, fetch the data process the request store back, you know, after adding the item to the shopping cart, this is the new value of the shopping cart, you have to store it back into the data store. Why? Because the next request to view the shopping cart might go to some other replica. And that replica will fetch it from the data store.

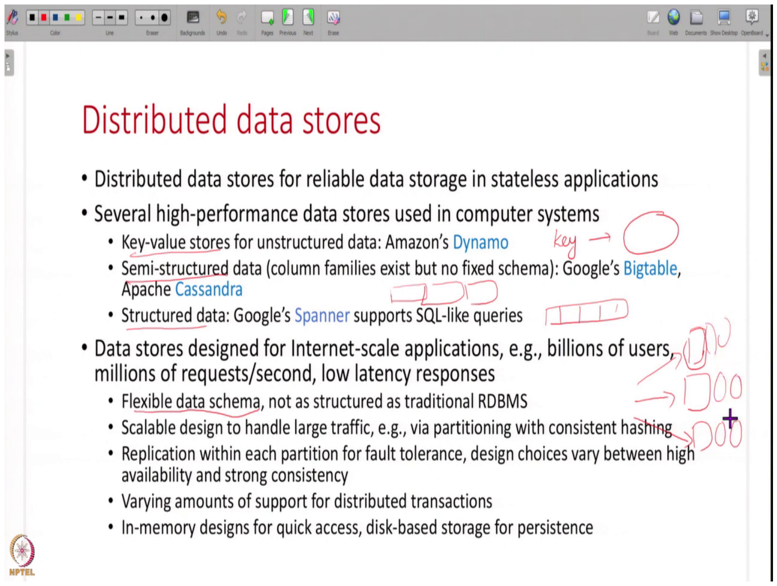
So, therefore, the updated value should once again go to the data store. And you can send a confirmation back to the user only after this check pointing is done, before that, you will not send a response back to the user. Why? Because what if you could not contact the data store, then if you have sent a confirmation, you could lose that data if a failure occurs.

So, handling every request, fetch the current state, update the state, store it back in the data store and reply back to that request. So, the handling of each request, as you can see, there is some high overhead here of multiple repeated communications with some other machine somewhere else the data stored machine. But this simplifies a few other things, your load balancer design is simple every request, you know, of course, at a TCP connection level, you have to send the same connection to the same node, you cannot distribute a TCP connection.

But any request comes all packets of a connection, just go to any replica, you do not have to think, oh, where is this user's shopping cart located. So, it simplifies the load balancer design. And you also do not need any special mechanisms for reliability at the server. Why? Because any replica has no state. So, if it crashes, you send the next request to another replica, you know, if the server in the middle of processing crash, the user will retry, that request will go to another replica, it will handle that request, nothing is lost.

But all the complexity shifts to the reliable data store. Now, this data store must use some, you know, replication techniques, and all of that to store the data reliably. So, you have shifted the complexity to the data store, but your replica, your server design itself is fairly simple. And of course, in real life, you may not have a pure stateless or fully stateful, you might have some hybrid in between where some store state is stored locally, some is stored in the data store, that is also possible.

(Refer Slide Time: 12:02)



Distributed data stores

- Distributed data stores for reliable data storage in stateless applications
- Several high-performance data stores used in computer systems
  - Key-value stores for unstructured data: Amazon's Dynamo
  - Semi-structured data (column families exist but no fixed schema): Google's Bigtable, Apache Cassandra
  - Structured data: Google's Spanner supports SQL-like queries
- Data stores designed for Internet-scale applications, e.g., billions of users, millions of requests/second, low latency responses
  - Flexible data schema, not as structured as traditional RDBMS
  - Scalable design to handle large traffic, e.g., via partitioning with consistent hashing
  - Replication within each partition for fault tolerance, design choices vary between high availability and strong consistency
  - Varying amounts of support for distributed transactions
  - In-memory designs for quick access, disk-based storage for persistence

So, now if we look at these data stores, which are used for reliable storage in the stateless applications, there are many different data stores that are available in real life systems. Of course, in this course, we do not have the time to go into details about all of these, but if you take a course on cloud computing, or distributed systems, you will study the design of all of these data stores in a lot of detail. But here, I will just summarize the high-level concept of how you go about building a distributed data store.

You know, you have many data stores, some are for, like Amazon's dynamo, we have seen this before, this is a key value store. This is for unstructured data, you have a key some identifier and some blob of value, it can be anything, you can put anything in this value, then you have semi structured data, which is your data has some structure, you know, it has, you know, some what are called column families, you have these these these types of columns. And you have fully structured data where you have like, you know, fixed table kind of things.

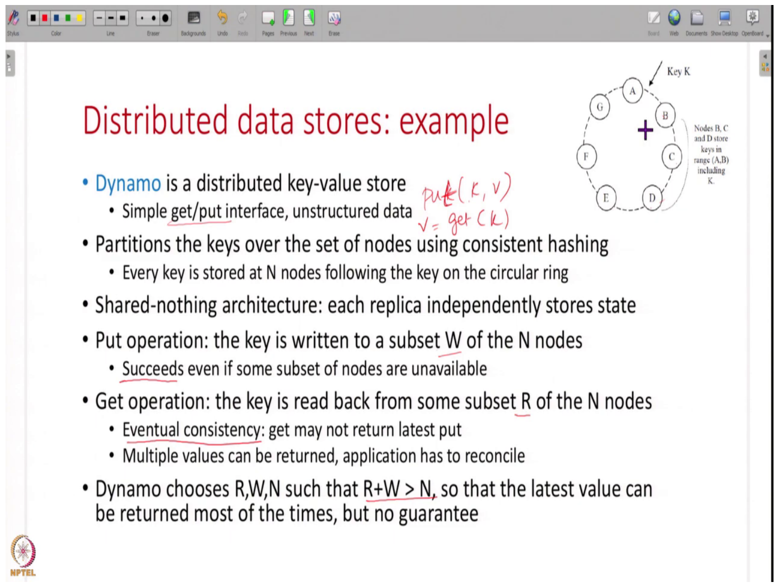
So, you have a wide range of choices available for data stores. And all of these data stores, of course, there are a lot of subtleties in their designs. If you look at the papers, you know, from dynamo, there is Google's big table, Cassandra, Spanner, you have many such data stores in use today. But across all of them, they have some common design principles.

You know, they are all designed for internet scale billions of users, millions of requests per second throughput, very high throughput, low latency, they want all of these properties and how do they achieve them, they achieved them by having a flexible data schema, you know, they are not as rigid as databases with fixed schema, then they have scalable designs, you know, you do not have a single replica single machine storing all of Google's information, you know, they usually have horizontal scaling, you have multiple replicas, data is partitioned across multiple replica, say using consistent hashing.

And each of these replicas or shards is once again, this information is replicated at multiple other servers for fault tolerance. And, of course, they provide different amounts of support, some support distributed transaction, some do not. And some of them store all the data in memory for quick access some store it on disk.

So, in this way, if you read the information about these data stores, there are many different designs, but across all of them the basic idea of you know, partitioning for horizontal scalability, replication for fault tolerance, some support for atomicity in a single server in a distributed setting these concepts that we have learned, you will see the same pattern across all of them, even though the individual details might vary.

(Refer Slide Time: 14:47)



Distributed data stores: example

- Dynamo is a distributed key-value store
  - Simple get/put interface, unstructured data
- Partitions the keys over the set of nodes using consistent hashing
  - Every key is stored at N nodes following the key on the circular ring
- Shared-nothing architecture: each replica independently stores state
- Put operation: the key is written to a subset W of the N nodes
  - Succeeds even if some subset of nodes are unavailable
- Get operation: the key is read back from some subset R of the N nodes
  - Eventual consistency: get may not return latest put
  - Multiple values can be returned, application has to reconcile
- Dynamo chooses R,W,N such that R+W > N, so that the latest value can be returned most of the times, but no guarantee

For example, I will just give you a small example of Dynamo that we have seen so far you know, it's a key value store is a simple you know, get put interface, you put some key and some value, and then you can do a get of that key, then it will return whatever value that you have put it it's a simple interface. And your value can be anything you can store whatever you want.

For example, Amazon uses something like this to store shopping carts. Key is the shopping cart ID, the value is the contents of the shopping cart. So, how does this Dynamo work, it has a partitioned, shared nothing architecture, it's called, you know, you have multiple nodes, you do consistent hashing, you hash the key to this ring. So, there is a figure from the paper, you hash, the key, the nodes are also hashed on to the ring, and every key is stored at the nodes immediately following it on this ring, you do not store it at one node, you store it at multiple nodes for some kind of fault tolerance.

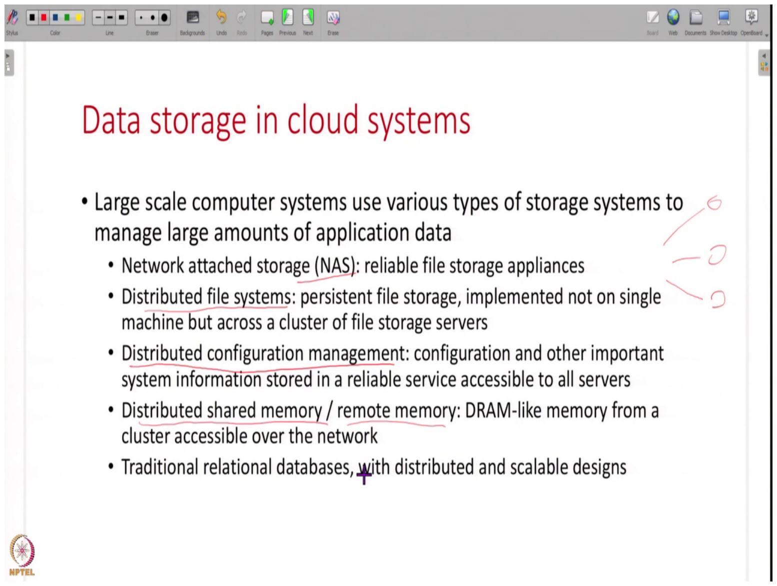
So, the put operation, it is written at all of these nodes, the get operation will get from all of these nodes, and you will put at some W node you will get from some r nodes, and you will ensure that r plus w is greater than the total number of nodes so that there is at least some intersection between the nodes, you have written to the node you read from so that you can get back the latest value.

So, this is the basic idea it is a quorum protocol but it also guarantees only weak eventual consistency, you know, why is that you try to write to all the nodes, but if you cannot contact all of them, you will still return a response, this put operation, all your access to the data will succeed even if a subset of nodes cannot be contacted your get operation also whatever values you get, you get two different values, you will return all of them, it will return multiple values to the application, tell the application, there are many versions, I do not know which is correct, you do what you want.

In this way, it only guarantees weak eventual consistency, but it is very fast, it has high availability, because an operation will always succeed. Even if there are node failures, even if nodes are a majority cannot be contacted, and so on. In this way so this is one sample point. On the other hand, you can have a highly consistent key value store, which is you know, across all of these replicas where a key store it could be running something like raft to ensure strong consistency and you know, return a response to either add an item to a shopping cart or whatever it is return a response only when the application succeeds at a majority, you can have a design like that also.

As you can see, there is a wide spectrum of data stores depending on different amounts of support for consistency models, different amounts of support for transactions, how do you learn the CAP theorem, how do you trade off consistency versus availability, many different designs are possible, but as you can see, the basic principles are the same of what we have studied.

(Refer Slide Time: 17:44)



Data storage in cloud systems

- Large scale computer systems use various types of storage systems to manage large amounts of application data
  - Network attached storage (NAS): reliable file storage appliances
  - Distributed file systems: persistent file storage, implemented not on single machine but across a cluster of file storage servers
  - Distributed configuration management: configuration and other important system information stored in a reliable service accessible to all servers
  - Distributed shared memory / remote memory: DRAM-like memory from a cluster accessible over the network
  - Traditional relational databases, with distributed and scalable designs

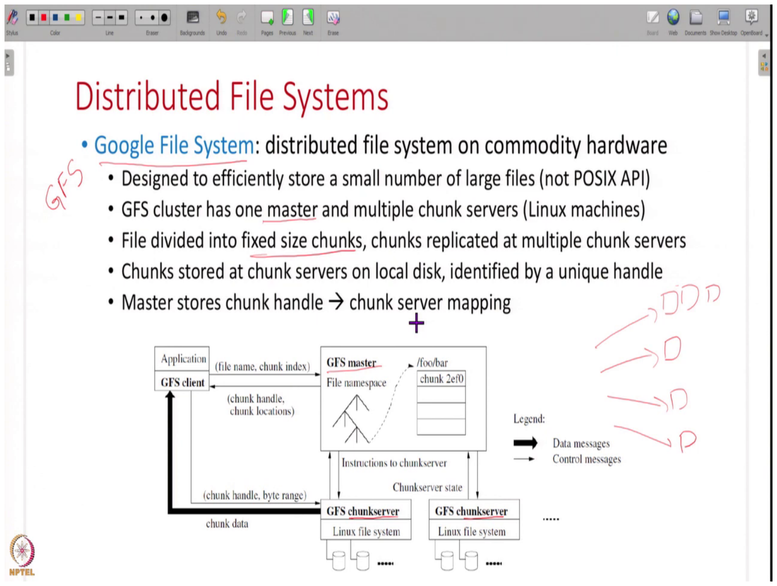
And there are many other options to store data in cloud systems, you know, key value stores data stores is one thing, you also have what are called NAS storage, that is you know, network attached storage, you have some file storage hardware boxes that you can buy, you also have file systems that are distributed, instead of storing a file in one disk you can actually distribute it to multiple different disks on multiple different machines and store a file.

So, that you have some reliability, you know, it is not a key value store is actual file storage, but that is distributed. There are also what are called distribution configuration systems, you know, in a large system, you have to store various pieces of information like configuration and small small things that you have to know how many nodes are there in the system, all of this information, you have to store somewhere.

So, these also are stored in a distribution configuration system, where multiple servers are once again, running some consensus protocol replicate this information, you can read and write this configuration information over the set of servers, then you also have distributed shared memory, you know, or what is called remote memory, you can access the DRAM of cluster of servers and you know, you have its as if your system has a large amount of DRAM itself, even when it has very limited DRAM physically.

And you have relational databases, which are once again having a distributed, scalable design. In this way, if you take a course on databases, or distributed systems or cloud computing, you will see that there are many different options available to store data and cloud systems. And across all of them the common theme is you replicate thing and things in multiple places, you partition and then you replicate so that you have scalability as well as you have fault tolerance.

(Refer Slide Time: 19:26)



Distributed File Systems

- Google File System: distributed file system on commodity hardware
  - Designed to efficiently store a small number of large files (not POSIX API)
  - GFS cluster has one master and multiple chunk servers (Linux machines)
  - File divided into fixed size chunks, chunks replicated at multiple chunk servers
  - Chunks stored at chunk servers on local disk, identified by a unique handle
  - Master stores chunk handle → chunk server mapping

So, for example, I will just show you a very simple example of a distributed file system, which is called the Google File System, or GFS. This is one of the first distributed file systems that has come up and it is one of the reasons for you know, the great success of Google where it could store large amounts of data of that it got from crawling the web in its search engine, it could store it very efficiently in a distributed file system on commodity hardware on regular servers but they were made reliable by using all of these concepts like you know, replication and partitioning.

So, how does GFS work? You have many servers, what are called chunk servers. Your file is basically divided into fixed size chunks. This, again is an idea we have seen, you know, divide things into fixed size chunks, it's easier to manage, you divide a file into fixed sized chunks. And these chunks are distributed over all of these chunk servers. And of course, not just one chunk server, you will also replicated multiple chunk servers, you know, the same file some chunks are stored here, some are stored here.

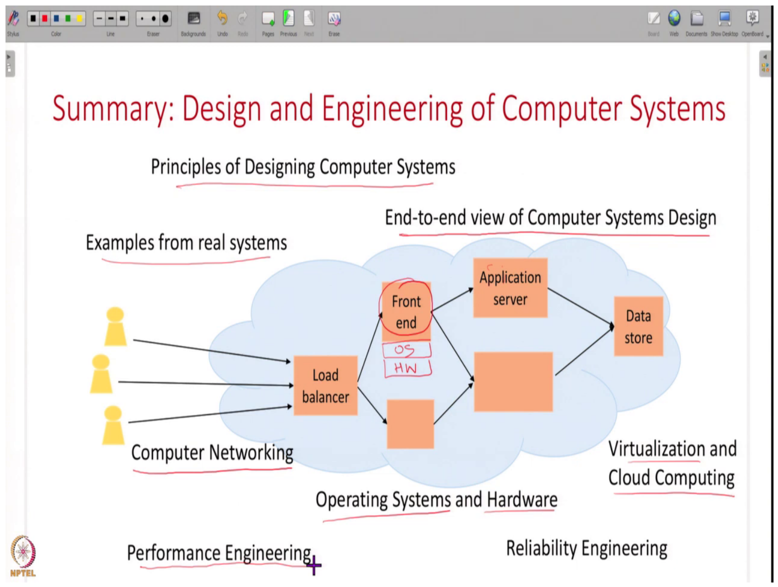
And each of these chunks is replicated at multiple different servers for fault tolerance. So, in this way, file is divided into chunks, and each chunk is at multiple chunk servers. And all of this information is maintained at what is called a master node, this master has information about all files, where are all the chunks of that file located. So, whenever an application, somebody wants to read a file, they will talk to the master get this information about, you know, which servers have which chunk of the file, and then directly talk to the chunk server and transfer the file data.

So, this is not an exact POSIX API compliant file system. But it is very efficient to store large files. You know, in Google, if you have large files corresponding to the information you get from crawling the web, for the search engine, all of those files are stored in a file system like this that is distributed because you have partitioned to multiple servers, you have very high capacity more than what you get by storing at a local disk at the same time, you have reliability, because you have replicated and all of this tracking is done by the master.

Now, this master can also be made fault tolerant. Now, this information in the master you perhaps replicated using Paxos or raft or something, make sure it is safely stored. So, that this mapping of you know which chunk is at which server is also stored reliable. So, this is a sample example of a distributed file system. But there are as you can see, the similar concept you can apply to build many other kinds of distributed storage systems.

So, with this, I would like to wrap up the course we have reached the end of the last lecture in the course, I hope you all have enjoyed taking this course and you have enjoyed the journey that we have come on throughout the last 40 lectures and or 8 weeks. So, I would like to just briefly summarize what we have studied or what we have hopefully learned in this course.

(Refer Slide Time: 22:26)


Summary: Design and Engineering of Computer Systems

Principles of Designing Computer Systems

End-to-end view of Computer Systems Design

Examples from real systems

Computer Networking

Operating Systems and Hardware

Virtualization and Cloud Computing

Performance Engineering

Reliability Engineering

So, from now on, if you look at real life system, like you know, an e-commerce system or a ticket reservation system or any other real life system out there, if you look at it, you should be able to visualize it into its component, you should be able to see all the components of the system, for example, you know, users are accessing some system, typically, it is composed of multiple tiers, and all of these servers are hosted either on VMs, or containers, and are hosted in a large data center or a cloud.

You know, we have learned about concepts from virtualization and cloud computing as to how a large system like this is architected, then, you know, these servers each of these servers underneath is just a user space process you know, underneath this, there is an OS, there is a hardware. So, we started out the course by learning about the basic platform layer, you know, how computer hardware works, how operating systems work, we studied all of that, then we studied how these application processes either run directly on the hardware or run inside VMs or containers and are hosted on a cloud management system and are automatically orchestrated by a cloud management system we have seen that.

Then we have seen how users and the system and you know the system components themselves, communicate over the internet or the computer network, you know, we have studied concepts from networking. And through all of this, you know, I have hopefully explained to you how the end to end view of a computer system looks like you know, right from the user go over the network, to the various tiers of a multi-tier application and each tier when this application server is running, how it has a multi-threaded or a multi process design, how it runs on the underlying operating system on the underlying hardware, how all of these tiers are managed by a cloud orchestration system, you should be able to visualize the entire system end to end.

And across all of these examples that we have taken from real life system, you should see that there are some common design principles that we repeatedly come across in the course you know, for example, concepts like caching concepts like partitioning concepts like replication, you know, doing fixed size allocations, you know, virtualization, all of these things, we have seen them multiple times in different different contexts in different modules throughout this course.

So, now you have an appreciation of, you know, the principles of designing computer systems, you can see that all of these are in the end, common sense and the same common-sense principles are used at multiple layers throughout a computer system. And we have seen this end to end view, and then we have said, how do you measure performance, how do you improve performance, once again, performance engineering also has many different parts, you know, we have seen at the memory layer at the hardware layer, your memory accesses how to make them faster across a system, how do you make it faster by doing scaling by doing caching, we have seen at multiple layers, the networking part, how do you make it faster file systems how do you make them more efficient.

So, across all the layers, and across the end to end view, we have seen how to improve performance of a computer system. And similarly, we have studied about reliability, how to make the system more tolerant to faults, how to ensure atomicity, how to ensure atomicity in a distributed setting, what are you know, various consistency models, how do you replicate to achieve strong consistency, weak consistency all of these concepts we have seen.

So, by the end of this course, I hope that you are able to piece all of these smaller parts together, and they all somehow fit in your head to form one coherent end to end picture of how real-life computer systems are built. And tomorrow, if you look at any computer system, you should be able to break it down into all of these parts. And you should be able to understand all of these parts individually.

And also, you should be able to design computer systems on your own. If somebody asks you to, you know, hey, design a system like YouTube, you should be able to, you know, come up with this modular design, you should be able to say how to improve the performance, how to make the system more reliable. You know, how do I go about building a good system that performs well, that is reliable, that is functionally correct, you should hopefully have all the tools at your disposal to apply them to build any real-world system.

And I hope that you all have enjoyed this course and, in the future, if you like computer systems, and this is sort of your one of the first courses in computer systems and I suggest you to take more advanced courses on virtualization, cloud computing, performance analysis, distributed systems. All of these future courses will offer you more insights and will deepen some of the concepts that we have studied in this course. So, that is all I have. This is the last lecture of this course and I hope you all have enjoyed this course I have enjoyed teaching it very much and I hope you have enjoyed learning as well and all the best for your future. And I would like to sign off on this note. Thank you