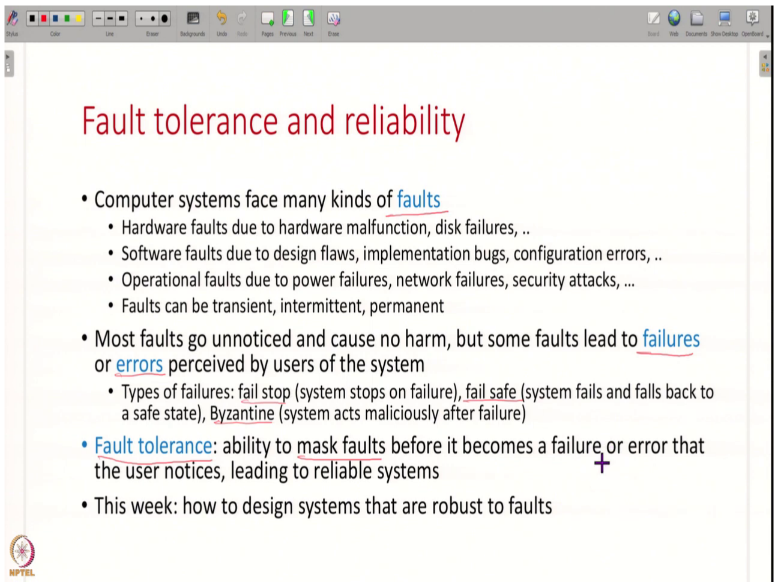
Design and Engineering of Computer Systems
Professor Mythili Vutukuru
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Lecture 51
Fault Tolerance and Reliability

Hello everyone, welcome to the 36th lecture in the course Design and Engineering of Computer Systems. This week in this last week of the course, we are going to study more on the topic of reliability. So, let us get started. So, the topic of this week and beginning with this lecture is Fault Tolerance and Reliability.

(Refer Slide Time: 00:37)



Fault tolerance and reliability

- Computer systems face many kinds of faults
  - Hardware faults due to hardware malfunction, disk failures, ..
  - Software faults due to design flaws, implementation bugs, configuration errors, ..
  - Operational faults due to power failures, network failures, security attacks, …
  - Faults can be transient, intermittent, permanent
- Most faults go unnoticed and cause no harm, but some faults lead to failures or errors perceived by users of the system
  - Types of failures: fail stop (system stops on failure), fail safe (system fails and falls back to a safe state), Byzantine (system acts maliciously after failure)
- Fault tolerance: ability to mask faults before it becomes a failure or error that the user notices, leading to reliable systems
- This week: how to design systems that are robust to faults

Let us spend some time trying to understand what these words mean and what is reliability, and what is fault tolerance? So, all computer systems they are not perfect, nothing in life is perfect. Similarly, computer systems also they have various kinds of faults or errors from time to time, so these can be hardware faults like your hardware stops working, your hard disk crashes, there can be software faults, there is some bug in your code, some implementation, design issue, some configuration errors, you did not configure it properly, then while the system is running at operational time, that could be faults like the power fails, network fails, a security attack.

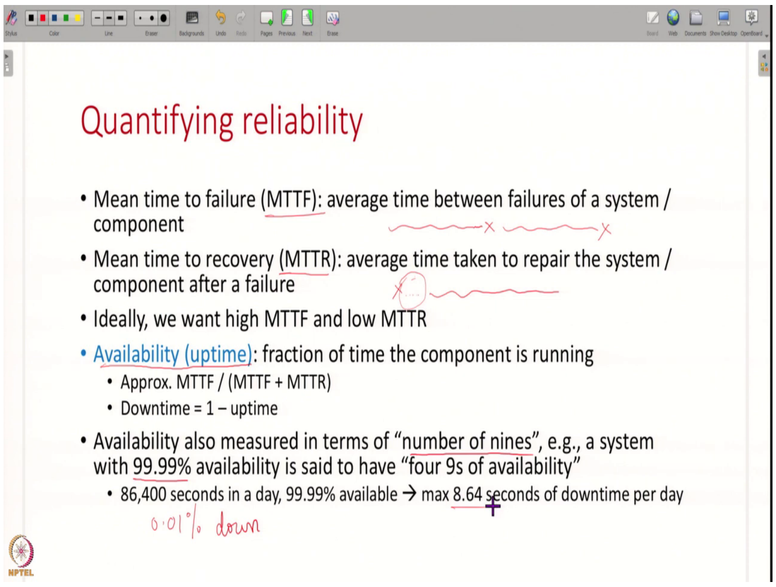
So, there are many kinds of things that can go wrong, many kinds of faults in a computer system and these faults can be transient that is they happen only briefly go away, they keep on happening intermittently, or they can happen permanently, cause permanent damage, whatever it is, there are many things that can go wrong. And most of these faults they might go unnoticed and you may not even notice that the fault has happened everything is as usual going on as usual. But sometimes these faults cause failures or errors, which are noticed by others; noticed by the users of the system. You try to access a ticket reservation website; it does not work that is a failure that the users perceive.

And these failures are also many types, when your system fails, it can completely stop upon failing; those are called fail stop failures. The other mode is the system fails, but it still continues to work in some sort of a minimal safe state that will continue to work those are called fail safe systems or the worst kind is your system fails, there is some failure error and your system starts acting weird malicious after the failure those are Byzantine failures. So, whatever it is all of these failures, they will be noticed by the users of the system and that is what we want to avoid.

So, fault tolerance is a way in which you can design systems such that you can mask these faults, whatever faults are there in real life that are inevitable, you will somehow design around them, so that they can be masked, they can become harmless and it will not become a failure or something that the end users of a system notices. So, the goal here is not to completely get rid of faults that will not happen, that is the nature of the world that faults will keep happening. The goal here is only that we try to somehow hide these faults, so that they do not result in some failures or errors that others will notice.

So, this week, we want to see various techniques by which we can make systems robust to faults and therefore make them more reliable. So, before we try to see the techniques, let us try and understand how do you measure reliability? If you can measure something, then you can say my reliability is getting better, it is getting worse and so on. So, there are a few metrics using which we can quantify, measure the reliability of a system.

(Refer Slide Time: 03:55)



Quantifying reliability

- Mean time to failure (MTTF): average time between failures of a system / component
- Mean time to recovery (MTTR): average time taken to repair the system / component after a failure
- Ideally, we want high MTTF and low MTTR
- Availability (uptime): fraction of time the component is running
  - Approx. MTTF / (MTTF + MTTR)
  - Downtime = 1 – uptime
- Availability also measured in terms of "number of nines", e.g., a system with 99.99% availability is said to have "four 9s of availability"
  - 86,400 seconds in a day, 99.99% available → max 8.64 seconds of downtime per day

One is the mean time to failure between the system is working fine for all this time, then it fails, then you repair it, it again is working fine, then it fails again. So, the average of these durations, the time between successive failures of a system that is called the mean time to failure, obviously, you want this to be as long as big as possible.

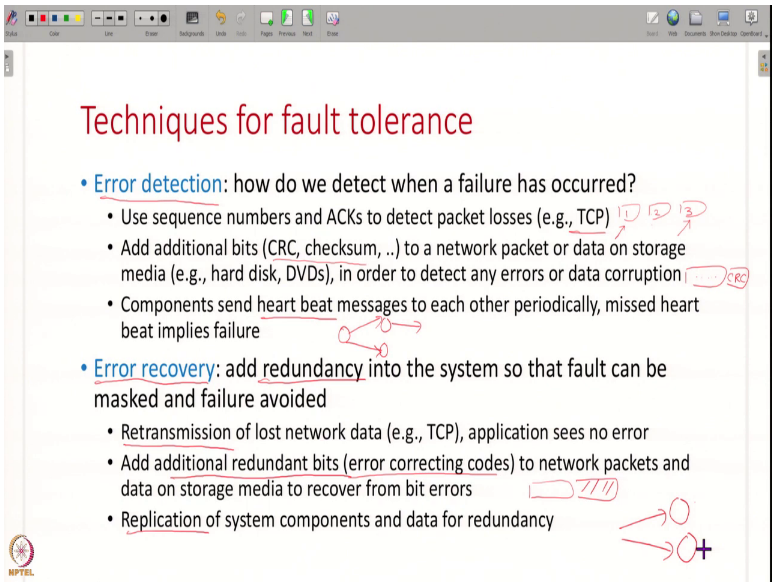
Then the next metric is mean time to recovery that is, once your system fails, you do some steps to fix the failure and then it starts to run again, the average of these durations where your system is down getting repaired, and then it recovers and runs again, this average of these durations, that is the mean time to recovery. And you want this to be as low as possible; you want your system to fail, very rarely and whenever it fails, you want it to recover or get repaired as quickly as possible.

So, there is a metric that captures all of these behaviors of a system that is called the availability or the uptime, which is the fraction of time the component is running and this is approximately the duration between failures divided by the total duration which is the duration between failures as well as the time taken to repair. So, this is the availability definition and of course, one minus this uptime is the downtime of your system. So, this uptime or downtime measure what fraction of the time your system is running and what fraction of time it is down and it is being repaired.

And there is also another way to measure availability, which is, this is the colloquial number definition that you see which is in terms of number of nines, if your system has 99.99 percent availability, this is called four nines of availability, 99 percent availability is two nines of availability like that; the number of nines that are there in the availability that is said to be a common way of quantifying your availability of the system. So, what does these mean? Four nines of availability means that your system is only down for 0.01 percent of the time it is down, that is a very small fraction of time.

For example, in a day, you have so many seconds in a day; so, maximum of 8 seconds, during the day your system can be done anything more than this it will not have four nines of availability. So, in this way when people build system they say my system has four nines, five nines, six nines of availability and so on. So, these are ways in which you quantify how available or how reliable your system is. So, now let us broadly look at what are some of the techniques for fault tolerance.

(Refer Slide Time: 06:48)



Techniques for fault tolerance

- Error detection: how do we detect when a failure has occurred?
  - Use sequence numbers and ACKs to detect packet losses (e.g., TCP)
  - Add additional bits (CRC, checksum, ..) to a network packet or data on storage media (e.g., hard disk, DVDs), in order to detect any errors or data corruption
  - Components send heart beat messages to each other periodically, missed heart beat implies failure
- Error recovery: add redundancy into the system so that fault can be masked and failure avoided
  - Retransmission of lost network data (e.g., TCP), application sees no error
  - Add additional redundant bits (error correcting codes) to network packets and data on storage media to recover from bit errors
  - Replication of system components and data for redundancy

So, the techniques fall under two broad categories first is of course detecting when failures or errors occur and the other is recovering from these failures or errors. So, the techniques that we are going to discuss in todays lecture fall under these two buckets roughly. So, how do you detect when a failure has occurred? Some of these we have seen before for example, in the case of transport protocols like TCP, TCP sender is sending many segments and each segment has a sequence number 1, 2, 3, whatever, they are actually byte-based sequence numbers, but you get the idea.

So, every segment has a sequence number, so, when the receiver gets segment 1 and then the receiver gets segment 3, then the receiver knows or something in between here is lost. So, you can use sequence numbers and acknowledgements to detect packet losses in networking transport protocols.

There are also other ways, which is you can add some additional bits like CRC or a checksum a few additional bits here as your packet and some additional bits like the checksum or CRC are added to a packet or to any data that is stored on your hard disk so, DVDs or CDs, anytime you store some data, you add some additional bits, and these bits are useful to detect errors.

For example, after some time you re-compute these bits again when a network packet is received or when you read some data from a DVD, you re-compute these bits again, if it does not match, it means that something is wrong, some bit here is corrupted that is why the CRC or checksum is not matching. So, this is one way to detect errors.

Another way is, if you have a multi tier system, you have multiple components in a system, all of which are talking to each other exchanging messages, then you can do something like send heartbeat messages periodically, all the components in your system periodically keep saying hello, hello, hello to each other. And if you do not hear from somebody for a very long time, then you start to worry maybe that component has failed.

So, these are some of the techniques to detect errors when failures happen in a system, the first step to fix a failure are of course, to detect it. So, these are the error detection techniques, and then the next step is to actually recover from the error, to make the fault or the error go away. So, what are the error recovery techniques again, some of this we have seen, but the basic idea across all of this is to add redundancy into the system, of course, bad things keep happening. So, you just have a little bit of extra slack into your system so that you can recover from these bad things.

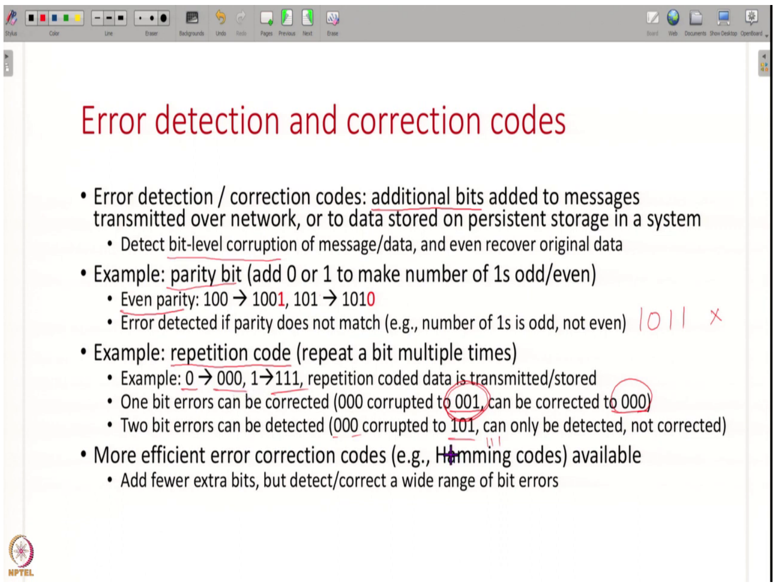
For example, again, in the case of TCP, you have retransmissions, whenever the sender realizes that a packet is lost then the sender will resend, retransmit this packet so that the end to end application layer does not perceive this fault at all, as far as, if you are browsing a web page, it is not like you see a hole in some part of your webpage that error is recovered even if the packet is lost once that TCP sender will resend it again and you will see the complete webpage or you will see a complete file that you have downloaded. In this way transport layer protocols like TCP they mask the network losses, network packet losses and recover these errors. So, retransmission is one way of adding redundancy into the system.

There are also other ways, when you send network packets or when you store data on, storage media like DVDs, what you can do is you can add some additional redundant bits, these are called error correcting codes, instead of storing these bits, you also add some extra bits so that together with all of these bits together, you can recover any errors that may have happened. So, we will see what these are in a little bit. And of course, the final technique is replication of your system components or system data.

For example, if some web server is prone to failures, then what you do is you have two copies of the web server so that if this guy fails, this web server will handle the HTTP requests. In this way have multiple replicas or multiple copies of your system components or data so that if one fails the other is still available. So, these are broadly the techniques for fault tolerance. Now, some of these will go into a little bit more detail in todays lecture.

So, first is error detection and correction code, so this is a very vast topic on its own. And if you take a networking course, or an information theory course, you are going to learn much more about it. In this one slide, I will just briefly touch upon the idea so that you know the concept of water error detection and correction codes.

(Refer Slide Time: 11:34)



Error detection and correction codes

- Error detection / correction codes: additional bits added to messages transmitted over network, or to data stored on persistent storage in a system
  - Detect bit-level corruption of message/data, and even recover original data
- Example: parity bit (add 0 or 1 to make number of 1s odd/even)
  - Even parity: 100 → 1001, 101 → 1010
  - Error detected if parity does not match (e.g., number of 1s is odd, not even) 1011 ×
- Example: repetition code (repeat a bit multiple times)
  - Example: 0 → 000, 1→111, repetition coded data is transmitted/stored
  - One bit errors can be corrected (000 corrupted to 001, can be corrected to 000)
  - Two bit errors can be detected (000 corrupted to 101, can only be detected, not corrected) 111
- More efficient error correction codes (e.g., Hamming codes) available
  - Add fewer extra bits, but detect/correct a wide range of bit errors

So at a high level these are nothing but you add some additional bits to either message that you send over the network or, data you store on storage systems like hard disks and DVDs and, wherever you store something or you send something instead of the original message alone, you add some additional bits so that together these additional bits will help you detect or correct some errors. Any bit level corruption that happens in your message or your data you can detect it and sometimes you can even recover what the original data was even if a few bits have been corrupted.

So, let us see some examples, one very simple error detection code is what is called adding a parity bit to your message. So, this parity bit can be either an odd parity or an even parity, what does it mean? Suppose, you are doing an even parity scheme, what you will do is at the end of your message, you will add one extra bit to make the number of ones in your message even, of course, if you are doing odd parity, you will try to make the number of ones odd.

So, whatever it is, the basic idea is the same for example, if your original message is these three bits 100, the number of ones is odd so you will add one extra 1, if your original message is 101, you do not need any more ones, if you add another 1 the number of ones will become odd again, which you do not want. So, therefore, you will add a 0.

So, now whatever message you send, it will always have an even number of ones. So, that is called an even parity scheme. Now, how is this useful? Now, if you receive a message and the number of ones is odd then that something is wrong, this is not suppose if you receive a message like 1011 then that the sender would never have sent something like this because the sender is doing an even parity scheme.

Therefore, you know that, this packet received or this data you read from the DVD, there is something wrong with it, you do not know what is wrong, but that something is wrong. And therefore, you can ask for a retransmission or ignore this or whatever depends on the application. So, this is an example of an error detection mechanism.

Then, even more complicated example is what is called repetition codes. That is you repeat a bit multiple times so that you can be more stable if errors occur. For example, you might say instead of sending bit 0, I will send I will repeat each bit three times instead of sending 0 I will send 000 instead of sending one I will send 111. So, this is a simple repetition code.

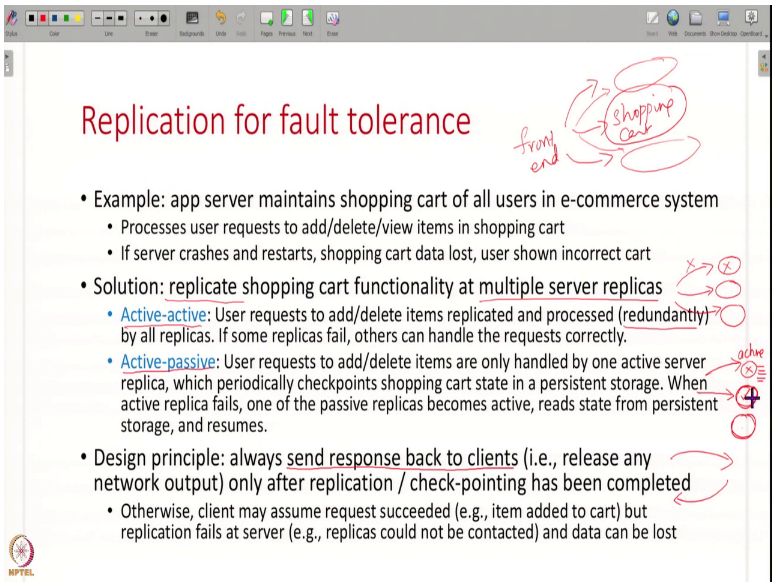
And these three bits instead of this one bit is what you will transmit or you will store on persistent storage. Now, when you are doing this, suppose one-bit error occurs, the 000 was corrupted to 001 then you know, because every bit is repeated three times, you can look at this you can take like some majority vote and see that most likely this was actually 000- and one-bit error has occurred. Therefore, if you know that one-bit error has occurred then you say, this is 000 there is no other option. So, not only have you detected the error, you have said that this is wrong data, you have also corrected it to the original data, and you have recovered the original data.

So, this is how error correction can also happen. But of course, this will not happen all the time. So, suppose two bit errors have occurred and your 000 two bits have flipped, and it became 101. Now, if you try to do error correction, you will most likely say, the original data is 111. But that is not correct; your original data was 000.

So, depending on how many bit errors you expect to happen, some codes can only detect errors, whereas some codes can also detect as well as correct errors, you can detect a larger number of errors, but you can correct only a smaller number of errors. So, that depending on your probability of bit errors, you can decide whether you want to be more aggressive and correct the errors or you want to be a little bit more conservative and only detect the errors.

And this is of course, a very simple dumb example, a repetition code, in reality, there are many more efficient error correction codes each bit is being repeated three times, this is so much waste of network bandwidth and all of that, but there are error correction codes, which add very few extra bits, but can detect and correct a large number of bit errors, like for example, the hamming codes. So, if you take a class on networking or information theory, you will learn a lot more about all of these different types of error detection and correction codes. So, this is one technique to detect and correct errors.

(Refer Slide Time: 16:17)



Replication for fault tolerance

- Example: app server maintains shopping cart of all users in e-commerce system
  - Processes user requests to add/delete/view items in shopping cart
  - If server crashes and restarts, shopping cart data lost, user shown incorrect cart
- Solution: replicate shopping cart functionality at multiple server replicas
  - Active-active: User requests to add/delete items replicated and processed (redundantly) by all replicas. If some replicas fail, others can handle the requests correctly.
  - Active-passive: User requests to add/delete items are only handled by one active server replica, which periodically checkpoints shopping cart state in a persistent storage. When active replica fails, one of the passive replicas becomes active, reads state from persistent storage, and resumes.
- Design principle: always send response back to clients (i.e., release any network output) only after replication / check-pointing has been completed
  - Otherwise, client may assume request succeeded (e.g., item added to cart) but replication fails at server (e.g., replicas could not be contacted) and data can be lost

The other technique is what is called replication. So, what is replication, all of us intuitively understand what it is. Suppose you have a server, let us take our example of an e-commerce website that we have been using all through the course, you have an e-commerce website, you have a front end, and you have many application servers handling different kinds of requests, for example, there is a server that is maintaining the shopping cart of users. whenever the user adds an item to the shopping cart, that request is processed by the server. And whenever the user wants to view the cart, this server will return for that user, the latest version of the user's shopping cart. It processes user requests, like adding, deleting, viewing items in a shopping cart.

Now, if the server, it has handled a bunch of requests, and then it has crashed, or user has added 10 items to the shopping cart, and the server crashes due to a power failure, or whatever. Now, when the user says, fetch me my cart, if the server restarts, all the memory is erased, and what will happen, the server will basically show an empty cart to the user, or the server is down some other server comes up to be the shopping cart server. And that guy says, I do not know what all you did in the past, but here is your empty cart. That is not what we want them. And that is ridiculous to be doing something like that.

Therefore, what you need is inspite of failures, you want the server to be able to show the correct data of the application and serve user requests correctly. And the solution for that is replication. So, instead of having one shopping cart server, you will have multiple replicas of the shopping cart server. And all of these together will be able to handle user requests and somehow be fault tolerant, there is some redundancy built into your system, instead of giving the job to one guy, you give it to like three or five different guys so that somehow the job gets done. So, the key idea is basically replication instead of doing it in one place, do it multiple server what are called replicas or copies of your server.

So, there are two broad ways in which you can do this replication. One is what is called active-active replication. That is you have multiple replicas, and all of these replicas are active replicas. That is, whenever a user request comes, user says add an item to a shopping cart, that request goes to all the replicas and all of them will update their shopping cart, when the user says view an item, the request will go to all of them, when the user says delete an item, it will go to all of them. In this way, all of these replicas are actively participating in this replication process. And they are all handling all the requests and they are keeping the application state all the time.

So, there is of course, you might think this is wasteful, but that is the whole point there is some redundancy in the system, instead of one guy doing the job, three guys are doing the same job or four guys or five guys how many replicas you decide to have, so that now you can see how this is a good idea. If one of these replicas fails, then you will stop sending requests to that server, you will send it to the other servers and they can return back the latest application state no data is lost. So, this is active-active replication.

The other technique is active-passive replication. That is, all the replicas are not actively handling requests all the time. So, all the user requests are sent to only one of the replicas, which is the active replica. And this replica only handles the requests and maintains all the application state and this replica will periodically checkpoint the state. Every so often it will store all the shopping carts of all the users somewhere say in some desk or something. And there is only one guy who is handling all the requests and if this guy crashes, then there are a few other replicas on standby, they are just on the side waiting and watching. They are not doing anything, they are not handling traffic all the time, but they are readily available.

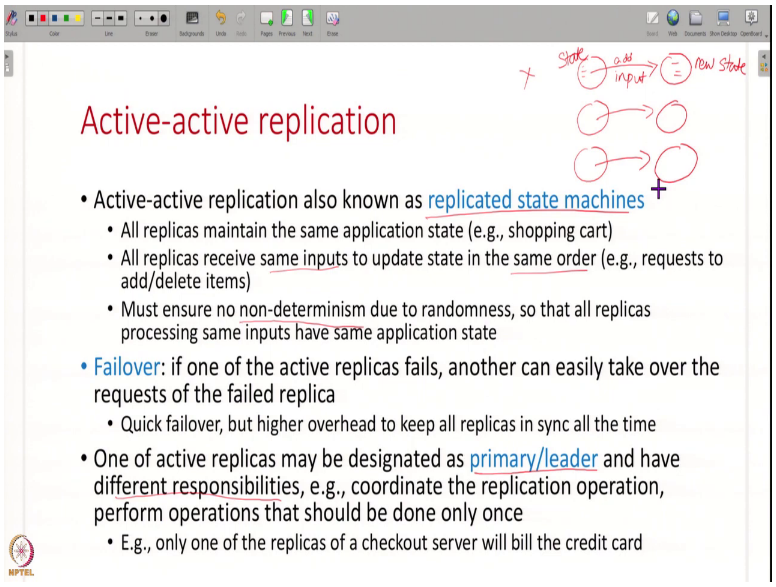
And if this replica crashes, then we will make one of these other passive replicas as the active replica and this guy will read whatever information the active replica had stored, information about all the shopping cards, this passive replica will become active, read all the state from persistent storage, and now it will start being the active replica and requests will come here.

So, until now, this passive replica was on the sidelines, once the active replica dies fails, and the passive replica will become active, and then it will start serving user requests. And because it has taken all of the state from the active replica, it will continue to correctly serve user requests. When the user has added items to the previous active replica, this new active replica has read all of that state and it can correctly display the shopping cart of the user.

So, these are two ways active-active and active-passive replication. And across both of these, you should always remember that you should send the response back to the client only after replication has been completed, whether it is active-active or active-passive, only after you store the state, only after the request is handled by all the active replicas, you are sure that there is redundancy in the system, only then the client sends a request only then you will send a confirmation back to the client, before your replication finishes, never send a confirmation back to the client.

Why? Because if you do that, and then your replication fails later on, then you already told the client, I handled your request, but in reality, his data is lost, he added an item to the cart that is lost because you did not replicate it and if the server fails, then the data is lost. Instead, make sure the data is safe, make sure the data is replicated at multiple nodes, or checkpointed, the status stored somewhere and then send a reply to the client. So, that even if a failure happens before you send a reply, then the client will retry the client will repeat the operation. But once you give a confirmation to the client, the client thinks it is done; there should not be any failure in the system.

(Refer Slide Time: 22:24)



Active-active replication

- Active-active replication also known as replicated state machines
    - All replicas maintain the same application state (e.g., shopping cart)
    - All replicas receive same inputs to update state in the same order (e.g., requests to add/delete items)
    - Must ensure no non-determinism due to randomness, so that all replicas processing same inputs have same application state
- Failover: if one of the active replicas fails, another can easily take over the requests of the failed replica
    - Quick failover, but higher overhead to keep all replicas in sync all the time
- One of active replicas may be designated as primary/leader and have different responsibilities, e.g., coordinate the replication operation, perform operations that should be done only once
    - E.g., only one of the replicas of a checkout server will bill the credit card

So now let us understand this active-active and active-passive in a little bit more detail. So, active-active replication is also known as replicated state machines. What does this mean? So, you can think of all the replicas as finite state machines, they are maintaining some state of the system, some application state, for example, a shopping cart, each of these replicas is maintaining.

And whenever they get some input, add an item or something request comes in an input is received then the replicas go to a new state. Here, the shopping cart was empty, and you add an item, then one item comes up into the shopping cart. So, in this way, the system has a certain state and then a request comes, an input comes and then it goes to a new state, you can think of every application server like this as a finite state machine.

Now, with active-active replication, all of these state machines are maintained in sync, they are all replicated state machines, that is the same input is being given to all the replicas and they all start in the same state, and therefore they will all end up in the same state at any point of time they are all in lockstep, they are all in sync with each other.

Because they start with the same state, they get the same inputs in the same order, and therefore they will end up in the same state. Of course, you should ensure that there is no non-determinism in the system, there is no randomness in going from one state to the other, so that for the same inputs, you reach the same end state. And this is the concept of replicated state machines and it is very important that all the replicas start with the same initial state, receive the same inputs in the same order and therefore end up in the same final state, the application state is always in sync across all the replicas.

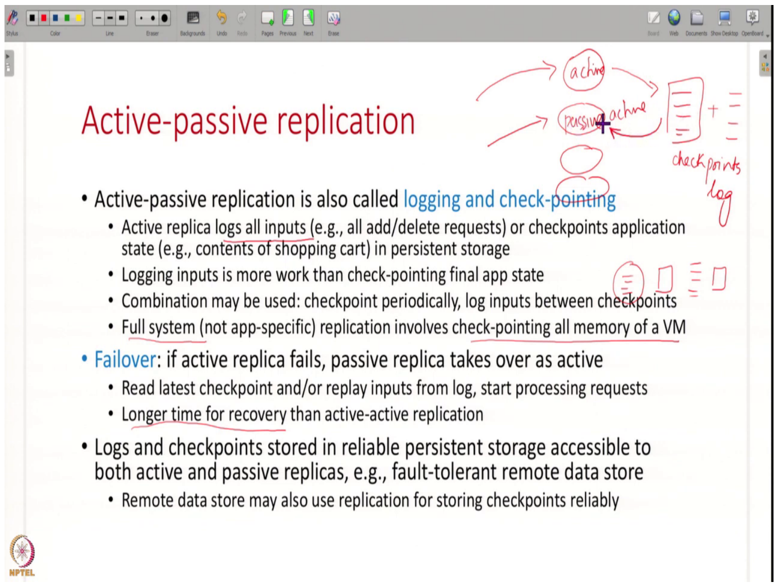
Then what happens upon failure, if there is a failure, one of the replicas fails, it is okay, the other replicas also have the same state so they can any time instead of sending requests to this replica, you start sending it to the other replicas, and all of them are identical so it does not matter who handles which request.

So, your failover recovery from failure is quite fast, but of course, this is higher overhead because you have to always keep all the replicas in sync with each other. So, how this is done, we will see a little bit later, this week we are going to study techniques for how to do this, but I hope the basic concept is clear here.

Then the other thing to notice, even though all these replicas are active, and they are all doing the same work, they all may not have the same responsibility. There might be slight difference in responsibilities. For example, one of them can be usually what is called the primary replica or the leader, that a leader might take more responsibility. For example, the leader might be coordinating this replication and sometimes there are operations that should be done only once.

For example, if you have a server that is doing the checkout and billing users in an e-commerce website, you cannot say, I have three replicas, therefore, I will charge the user's credit card three times, because all three of them are handling the request, you cannot say that. So, among these multiple replicas, some of them might have different responsibilities. And they are called like the master or the leader, or the primary replica, but all of them are handling all inputs, and they are all maintaining themselves in the same state with respect to the application.

(Refer Slide Time: 26:00)


Active-passive replication

- Active-passive replication is also called logging and check-pointing
  - Active replica logs all inputs (e.g., all add/delete requests) or checkpoints application state (e.g., contents of shopping cart) in persistent storage
  - Logging inputs is more work than check-pointing final app state
  - Combination may be used: checkpoint periodically, log inputs between checkpoints
  - Full system (not app-specific) replication involves check-pointing all memory of a VM
- Failover: if active replica fails, passive replica takes over as active
  - Read latest checkpoint and/or replay inputs from log, start processing requests
  - Longer time for recovery than active-active replication
- Logs and checkpoints stored in reliable persistent storage accessible to both active and passive replicas, e.g., fault-tolerant remote data store
  - Remote data store may also use replication for storing checkpoints reliably

So, the next is active-passive replication, this is also called logging and checkpointing. So, there is only one active replica, which gets all the user's requests and handles all the requests and periodically, this replica will checkpoint all the state of the application, so this is the active replica. So, what will you save? You can either save all the inputs that you get, if you are maintaining a shopping cart, or request to add delete items into the shopping cart, you can store or you can just store the final application state, after all of these requests here as my shopping cart, you can store the final answer or you can store all the intermediate steps also.

Both these are valid ways of checkpointing, but usually, if you are logging all inputs that is a little bit more work than just checkpointing the final state. So usually some combination of this is used, periodically, you will checkpoint the final state, and along with this final state, maybe you will also keep a log of intermediate inputs. So, you might do some combination of all of this, log a few inputs, then after some time, instead of all of this log here is the final state then again logs some inputs, or here is the summarized final state, you can keep doing that.

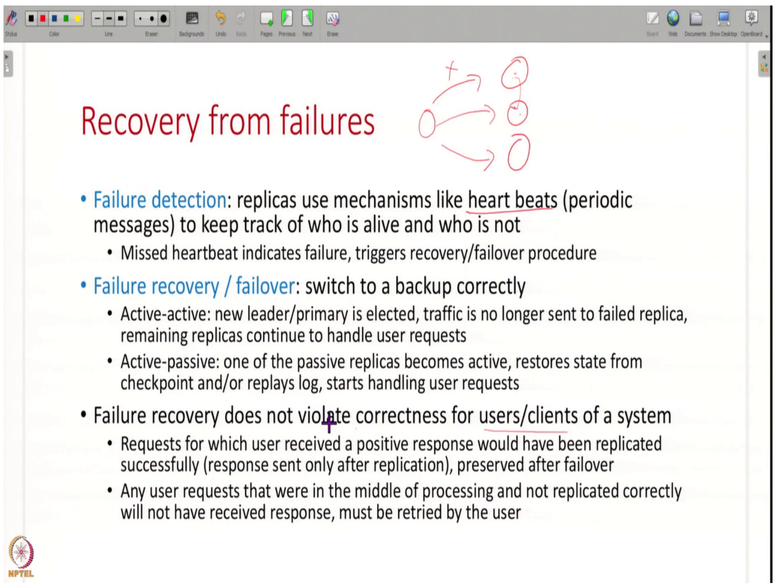
And sometimes what you can do is you can also check point, the complete memory of a system like if your server is running inside a VM, you checkpoint the all the memory of the VM periodically. So, this will give you full system replication, instead of just replicating the application data or the application server, you replicate the entire memory of a VM, so that all the applications, all the processes everything in it is checkpointed that is also possible. So, whatever you do, the active replica is basically either logging all the inputs, or periodically checkpointing the entire application state just the summary of processing all of these inputs, either one of these two, it will keep doing.

Now, when this replica fails, you have a few passive replicas on standby one of them will decide, I will become the active replica. So, this passive replica will take over and become active, request start going to this passive replica, but before that what this passive replica will do is it will read all this checkpointed state, it will update itself with whatever has happened in the past, it will read the checkpointed state, it will replay these long imports, add item, delete item, it will process all of those, so that it will come up to date with the application state, and then it will start to accept user requests. Now it is ready, it has become active.

So, you can see that there is a longer time for recovery as compared to active-active replication, because you have to do a little bit more work before you become active. And the other important thing is all these checkpoints, these logs, all of these things have to be stored in some persistent storage, some safe storage, you cannot say I checkpointed it but that log that checkpoint is information is lost because a hard disk failed that has to be stored in some fault tolerant data store, you might use a remote data store something else, some key value store or something that is present in some other location.

And that data stored may also use replication, whatever it is, you have to make sure that these checkpoints are stored somewhere safely that is accessible to both the active and passive replicas, so that the active replica can keep writing these things and the passive replica can read them when it is its turn to become active.

(Refer Slide Time: 29:31)



Recovery from failures

- Failure detection: replicas use mechanisms like heart beats (periodic messages) to keep track of who is alive and who is not
  - Missed heartbeat indicates failure, triggers recovery/failover procedure
- Failure recovery / failover: switch to a backup correctly
  - Active-active: new leader/primary is elected, traffic is no longer sent to failed replica, remaining replicas continue to handle user requests
  - Active-passive: one of the passive replicas becomes active, restores state from checkpoint and/or replays log, starts handling user requests
- Failure recovery does not violate correctness for users/clients of a system
  - Requests for which user received a positive response would have been replicated successfully (response sent only after replication), preserved after failover
  - Any user requests that were in the middle of processing and not replicated correctly will not have received response, must be retried by the user

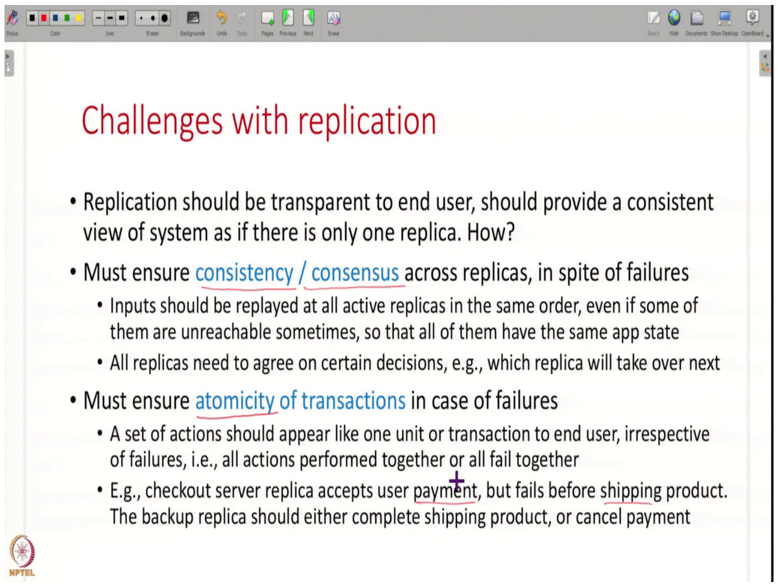
And now briefly, to summarize, how do you recover from failures? How do you achieve fault tolerance? So first, there are multiple replicas in your system. Each server is not implemented as one entity but as multiple replicas. And these replicas keep using messages like heartbeats; they keep periodically sending messages to each other to see who is alive who is not and so on. And if a replica or there is a load balancer that is disrupted getting traffic to all of these replica, this load balancer keeps monitoring, so everybody keeps monitoring each other. And when a failure is discovered, then you will do a failover procedure, if it is active-active, you will stop using the failed replica.

And you will redirect traffic to the other active replicas. If it is active-passive, then your failover mechanism will start this passive replica, it will restore the state from checkpoint replay logs, it will catch up and start behaving like the active replica. So, all of this, you detect a failure and you run this failover procedure, this recovery procedure. And all of these should make sure that as far as the users or the clients of a system are concerned, for them any correctness is not violated, any request for which a user has sent a request and you have sent a confirmation back, those requests are replicated safely in the system, you are sending a confirmation back only after replication.

So therefore, for those requests, you should not see any impact of the failure. Why? Because you have replicated it, you can recover that state. But suppose the user sent a request and the active replica was in the middle of processing that request and it did not replicate and it failed, then the user will not get a response back, the user will retry, the next time you will handle that request. So, you should always ensure that anything that you send a confirmation back to the user is replicated properly can be recovered.

And anything for which you did not send a confirmation back, that is okay, you are in the middle of replication, it did not complete that is okay, because the user will retry once again in the future, so this has to be kept in mind at the end user, it should never be the case that the end user has sent a request to you, you said I handled the request, and you sent a confirmation back. And then later on, you say oh, no, I lost the data due to some failure at mine that is not acceptable.

(Refer Slide Time: 31:53)



Challenges with replication

- Replication should be transparent to end user, should provide a consistent view of system as if there is only one replica. How?
- Must ensure consistency / consensus across replicas, in spite of failures
  - Inputs should be replayed at all active replicas in the same order, even if some of them are unreachable sometimes, so that all of them have the same app state
  - All replicas need to agree on certain decisions, e.g., which replica will take over next
- Must ensure atomicity of transactions in case of failures
  - A set of actions should appear like one unit or transaction to end user, irrespective of failures, i.e., all actions performed together or all fail together
  - E.g., checkout server replica accepts user payment, but fails before shipping product. The backup replica should either complete shipping product, or cancel payment

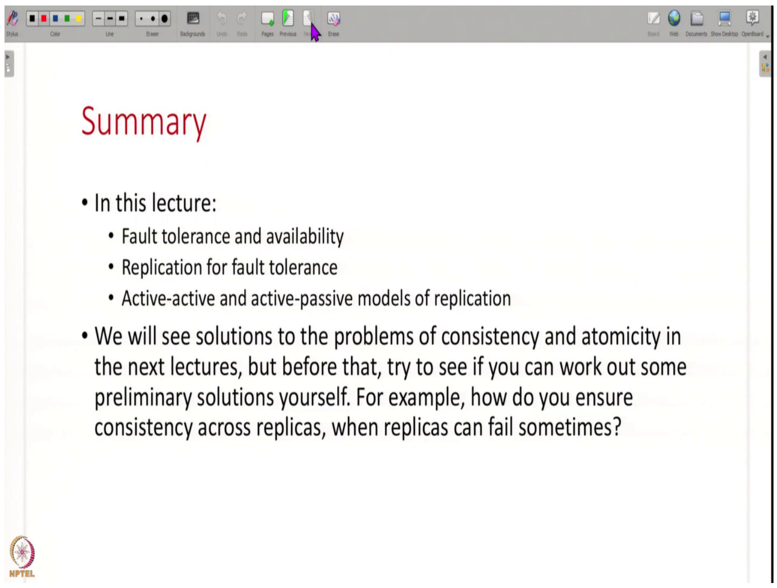
So now, there are certain challenges with doing this replication, it is not as straight forward, so what are these challenges? One thing is of course, what is called consistency. That is all these replicas, the fact that there are multiple copies of the data, multiple replicas of the server, all of this should be transparent to the end user, the end user should not realize any of these things.

So, all these replicas should somehow act as one, there should be consistency in the system, all the input should be replayed correctly, at all the active replicas, the checkpointing of the state should happen correctly, all the replicas should agree on who will take over when the active replica failed. So, there should be some agreement or consistency or consensus across all the replicas. It is not that one replica thinks your shopping cart has five items, the other thinks your shopping cart has six items that should never happen, so consistency that is an important thing.

And the other thing, important property is atomicity that is, if a set of actions are happening, they should appear like one unit to the end user, it should not be that you did half an action and you did not do the other half of the action due to a failure. For example, if the user says checkout a particular purchase, the user has placed an order and says checkout, then this checkout server will do the payment, and then it will do the shipping.

So, you should either do both, you should collect the user's payment and then ship the product. Or if the failure has happened, you do not do anything and then the user will check out again. But you should never do half and half you should never charge the users credit card, but then say a failure happened, I forgot to ship the product to you that is not acceptable. So, in spite of failures, you should ensure consistency across all your replicas and you should ensure atomicity within the processing of each replica. So, in the next couple of lectures, we are going to see how we are going to achieve all of these things. So that is all I have for today's lecture.

(Refer Slide Time: 33:56)



Summary

- In this lecture:
  - Fault tolerance and availability
  - Replication for fault tolerance
  - Active-active and active-passive models of replication
- We will see solutions to the problems of consistency and atomicity in the next lectures, but before that, try to see if you can work out some preliminary solutions yourself. For example, how do you ensure consistency across replicas, when replicas can fail sometimes?

In today's lecture, we have studied the basic concept of fault tolerance, we have seen what availability is and then we have seen the techniques for fault tolerance, like replication, active-active, active-passive replication, and so on. And we have said that there are some challenges to doing this replication, which is consistency and atomicity.

So, in the next couple of lectures, we are going to see how these challenges are solved in real systems. But before I tell you the answers, I would like you to think about, these are sort of common problems that in your day to day life, you want to store the same item in three different places. How do you ensure the three copies are consistent?

Think about what some of these solutions are when servers are failing. Of course, if all replicas are running all the time, consistency is easy. You store the same data at three places and it is done, but what if a replica fails? It missed adding an item to the cart because it failed? And now the shopping cart is inconsistent with other shopping carts.

In such situations what do you do, how do you ensure consistency, how do you ensure atomicity? So, please think through some of these solutions yourself, and in the next lecture we are going to discuss the answers. Thank you all that is all I have for this lecture and let us continue the discussion in the next lecture. Thanks.