

Advanced Distributed Systems
Smruti R. Sarangi
Department of Computer Science & Technology
Indian Institute of Technology, Delhi
Lecture 15
Virtual synchrony, 2-phase and 3-phase commit protocols

In this lecture, we will study about virtual synchrony and commit protocols. So the main idea over here is that we would like to see how to affect group communication, and that too reliable group communication, and how to use that to solve a very basic problem in distributed systems called the distributed commit problem.

(Refer Slide Time: 00:45)

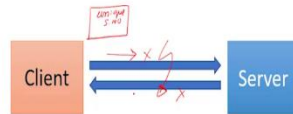
Virtual Synchrony



So, first we discuss virtual synchrony.

(Refer Slide Time: 00:50)

One-to-one Communication: Unicasting



- Client-server
 - Messages can get **lost**.
 - There is a need to **resend** messages.
- Two commonly used **semantics**
 - **At-least-once** semantics: The operation will be carried out by the server at least once.
 - **At-most-once** semantics: The operation will be carried out at most once.



3

So let us consider a one-to-one interaction between a client machine and a server machine. So it is possible that there can be faults in the middle and some of the messages may be dropped. So given the fact that messages can get lost, there would be a need to resend messages. So let us assume that the client sends a message.

So either the unreliable network can drop the message over here, or maybe the server can crash or maybe the response can get dropped. So the client does not really have a mechanism of distinguishing between these three scenarios. So often when clients do not get a response within a pre-specified period, they timeout, and they send their message once again.

So this is where the semantics of the server is very important. The question is that if let us say this is we are drawing money from a bank account, and if the message is sent, once again, we will have a double withdrawal which should not be allowed. So the two commonly used semantics that servers typically provide are At-least-once semantics and At-most-once semantics.

So in At-least-once semantics, if you are sending multiple messages, which are essentially multiple copies of the same message, it is guaranteed that at least one of them will execute, but can be more. So in our bank account example, we do not, do not want this.

We want at most once, which basically means we are fine if the transaction does not happen because in any case, the client has realized that something has gone wrong, if the transaction does happen, it should happen only once, which means that if money is being debited from the bank account, it should happen only once. So this is the, At-most-once semantics.

So different models have to provide one of these semantics, At-most-once being the more common. So what clients typically do is they assign a unique serial number to every message which can be randomly generated. So even if they want to retransmit the message they keep the serial number the same.

And so the server sees the serial number. If that message has been processed in the recent past, then the response is directly returned. the message is not executed once again, the command in the message is not executed once again.

(Refer Slide Time: 03:42)

Notion of Process Groups: Multicasting

- Processes can exhibit all kinds of failures
 - Fail-silent:** Just fails without any intimation.
 - Fail-stop:** The failure can be detected.
 - Fail-safe:** The failure is benign.
- Create a group of processes to service the client's request
 - Replicate the state across processes
 - Give the same user input to all the processes, collate the outputs, and decide the result based on voting
- Failure tolerance
 - To tolerate k fail-stop failures, we need $k+1$ processes.
 - If processes produce arbitrary outputs, we need $2k+1$ processes (use voting)
 - If the process sending the input is malicious, we need $3k+1$ processes (Byzantine)

NPTEL

So now, let us consider a slightly more sophisticated form of reliability where we actually create a process group. So let us first look at the kind of failures we are talking about. So one is a Fail-silent failure, where a process just fails without any invention. So nobody else actually gets to know the process has failed.

The other is the Fail-stop failure mode. So in this case, the failure can be detected. But, so then something can be done. At least we know that a given process is failed. The third is Fail-safe. In this case, the failure is benign, which means that even if a process is failed, we allow it to remain fail because it will pose no harm.

So what we actually do is we have a replicated state machine model which many consensus protocols notably, Raft also use. So what we do is that we actually replicate the servers. Each server has a state machine, which represents the internal logic of the server. So we have a centralized dispatcher.

Any client request, it is sent to the dispatcher. What the dispatcher does is that it creates multiple copies of it and sends it to the different servers. And all of them apply the command in the message to their internal state machine. So one advantage is that just in case, let us say two of these servers fail, at least this server can keep the entire system running. So this adds a layer of redundancy to a distributed system, such that it is immune to failures.

So this is done by replicating the state across the server processes and given the same user input, so same user input is given to all the servers, and they pretty much compute their results, which are supposed to be the same in the normal case. And then what we can do is that when their responses come, we have a voting engine where we essentially choose the majority, and that is passed on to the client.

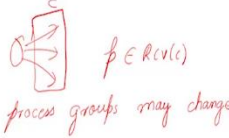
So let us look at some of the failure tolerance guarantees. So if there are k fail-stop failures, we essentially need $(k + 1)$ processes because one of the processes will keep running. If the processes are given the correct input, but they can produce arbitrary outputs. We need $(2k + 1)$ processes with k failures. The reason is that we will have a majority with at least the $(k + 1)$ correctly running ones, hence voting in this case would work.

If the dispatcher is not trusted, so the dispatcher is like the commander in the byzantine consensus algorithm. So if let us say the dispatcher is not trusted, so it can create different versions of the client request and send it. Then, we say that the input is wrong, it is

malicious. So tolerate such kind of failures as we have discussed in the lecture on Byzantine agreement, that we need $(3k + 1)$ processes.

(Refer Slide Time: 07:41)

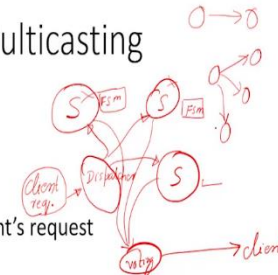
Reliable Multicasting



- Define a **multicast channel**, c
 - Sender group** $SND(c)$ – Processes that can **send** messages on channel c
 - Receiver group** $RCV(c)$ – Processes that can **receive** messages on channel c
- Reliability guarantee**
 - If process $p \in RCV(c)$, **message** m should be **delivered** to p , as long as p does not **change** its membership throughout the **duration** of the message transfer.
- Atomicity guarantee**
 - If **message** m is **delivered** to **process** p , then m is delivered to all the **processes** in $RCV(c)$. *all or none*

NPTEL

Notion of Process Groups: Multicasting



- Processes can **exhibit** all kinds of failures
 - Fail-silent**: Just fails without any intimation.
 - Fail-stop**: The failure can be detected.
 - Fail-safe**: The failure is benign.
- Create a **group** of processes to service the client's request
 - Replicate** the state across processes
 - Give **the same** user input to all the processes, collate the outputs, and decide the result based on **voting**
- Failure tolerance**
 - To **tolerate** k **fail-stop** failures, we need $k+1$ processes.
 - If processes produce **arbitrary** outputs, we need $2k+1$ processes (use **voting**)
 - If the process sending the input is **malicious**, we need $3k+1$ processes (**Byzantine**)

NPTEL

So let us just now define the notion of reliable multicasting because as if, as you see, this operation of sending one message to multiple servers is known as multicasting. So one-to-one message exchange is a unique cast and a one-to-many is a multicast. And this is where we desire some degree of reliability. So this is a very central mechanism in the design of our distributed system.

So let us define a multicast channel c , where if you look at it, this entire thing is a multicast channel c . And so we have a sender group, a sender group is a set of processes that can send messages on the channel. And we have a receiver group, which are a set of processes that can receive messages on channel c .

So the reliability guarantee is like this, that if a process p is a part of, is an element of receive c , then message m should be delivered to p as long as p does not change its membership throughout the duration of the message transfer. So which basically means that we are creating these process groups of a send group and a receiver group, so assume we want to deliver a message m to process p .

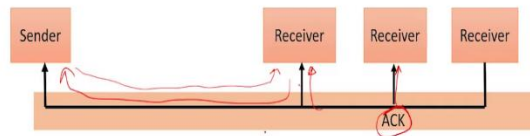
As long as p is an element of the received set and, it throughout the duration of the transfer, it continues to remain an element, the message should be delivered to p . So there is also an atomicity guarantee, which essentially says that if the message m is delivered to process p , it is delivered to all the processes in the receiver set of the channel.

So why do we have the reliability and the atomicity guarantees? Well, the reason we have these guarantees is because the sender and receiver process groups can actually change with the execution. So the process groups, the process groups may change. So given the fact that the process groups change, the question is that will we deliver a message to, let us say one process that was there in a group, and which was not there, what is the correctness?

So the reliability says that, look, if in the entire duration of the message transfer, a process was part of a group, it needs to get it. The atomicity guarantee says that if a message m is delivered to a process p in the group, then it is delivered to all the processes in that group. So we have all and or none logic, which we typically have with most autonomous city correctness constraints, that either a message is delivered to everybody in the process group, all the receivers in the process group, or to none of them. All or none.

(Refer Slide Time: 11:04)

Implementation in a LAN



- The sender **sends** a **message** to a set of **receivers**
- One of them sends an **acknowledgement (ACK)** on a shared channel
- The rest **snoop** the message
- If any receiver hasn't gotten the **original message**, it requests for a **re-transmission**



6

So this is not very hard to do on a simple LAN of course, local area network, where all the nodes are connected with each other on a bus. So the sender in this case would simply send a message to a set of receivers. One of them sends an acknowledgement, an ACK on the shared channel that it has gotten in. The rest are pretty much snooping on the channel.

So what happens is that if let us say one of them has sent an ACK, the rest do not have to set an ACK. Furthermore, it is possible that if a receiver has not gotten the original message, but it sees an acknowledgement for it, then it sends a separate request to the sender for the original message, which it gets back.

So this is kind of a simple way of doing this on a local area network, but the issue is that this requires some access to the internals of the network because in general, user processes are not allowed to snoop the Ethernet channel. So, so this is a good starting point, but we need to have a protocol that completely works at the level, at the application layer.

(Refer Slide Time: 12:22)

Virtually Synchronous Multicast

- Processes can **fail** at any time
- Hence, we need to **change** our **definitions**
- Virtually synchronous multicast
 - A message is **delivered** to all **non-faulty** members of the group
 - All the members **agree** on the current **group** membership
- View $V \subseteq RCV(c) \cup SND(c)$
 - Processes are **added** or **deleted** by **view changes**
 - In a **stable state**, all the processes agree on the current **view**
 - All non-faulty processes see all view changes in the **same order**



view
atomic
and
reliable
delivery
guarantees

7

So this is where the notion of virtually synchronous networks or a virtually synchronous multicast comes in. So the assumptions are as follows, the processes can of course fail at any time with or without information. So we define a virtually synchronous multicast, which is like this, that a message is delivered to all non-faulty members of the group. So it is atomic, is either delivered, or it is not delivered. But if it is delivered, it is delivered to all the non-faulty members of the group. That is point number 1

Furthermore, all the members of the group agree on the current group membership. So which means that the group membership, all the processes that are a part of the group membership, all the members agree on it. So that is the idea that a message is delivered to all the members, all the non-faulty members. And furthermore, all the members agree on the current group membership. And then either, we have an atomic delivery.

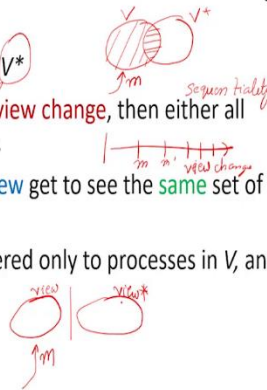
So we define the notion of a view, just change it to a pointer. We define the notion of a view over here which is the subset of the received set and a sent set. So the processes are added or deleted by, via a view change. In a stable state, all the processes agree on the current view, and pretty much the way a view is defined, a message is either delivered to all the processes in the view or to none.

And of course, the view is considered to be a set of working processes that are a part of non-faulty processes. That will clearly be a subset of the receivers and the senders. The good thing about the view in a virtually synchronous system is that a view is essentially a set of non-faulty processes, where you have atomic and reliable and reliable delivery guarantees. Furthermore, all the non-faulty processes see all the view changes in the same order, which means that there is a global consensus on the way that the view changes.

(Refer Slide Time: 15:20)

Virtually Synchronous Multicast – View Changes

- Let us say that view V changes to view V^*
- If a message m is sent to V before the view change, then either all $p \in (V \cap V^*)$, receive m , or none do.
- All non-faulty processes in the same view get to see the same set of multicast messages.
- A message sent to view V can be delivered only to processes in V , and not to successive views.



So, let us say that we have a given view, which means we have a set of processes. We have a process group. So view is essentially being used as a synonym for a process group. And the assumption here is that the process group is a subset of the global senders and receivers. And all the servers in this process group are non-faulty, at a given point in time.

So, let us assume that V changes to V^* . So if a message, if a message m is sent to V before the view change, then either all the $P \in (V \cap V^*)$ receive m or none do. So the important point is that, let us define a view, V , and let us define another view V^* . Let us say that before the view actually changes, I send a message m to view V .

So, one thing that is clear is that all the rest of the processes that do not change, they will get the message, should get the message, but consider this set, which lies at the intersection. So what we claim is that all the processes in this set also, also get the message m . So there

is an inherent sequentiality in the view change, which means before a view changes, all the messages are delivered to the overview.

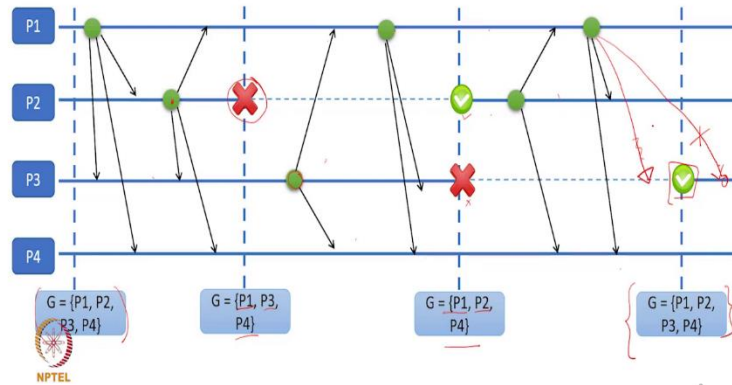
Then in a sense, so, if I were to consider a linear timeline, these are all the message deliveries of m , m' and so on. Then there is a view change. And after that, all the messages get delivered to the new view, with the same guarantees of atomicity and reliability.

So, the other interesting thing is that a message sent to view V can be delivered only to processes in V and not to successive views. So I will tell you the overall crux of this discussion. The overall crux of this discussion is like this, that we are defining process group of non-faulty processes called a view.

So, the view is essentially the current set of receivers. So if I were to send a message, it is either delivered to everybody or to nobody. And if the view changes, that is considered to be an atomic event. Just before the event, all the messages get delivered to the previous view. Just after the event, all the messages get delivered to the new view. So this is a sequentially happening thing.

(Refer Slide Time: 18:18)

Example of Virtual Synchrony



Virtually Synchronous Multicast – View Changes

- Let us say that view V **changes** to view V^*
- If a message m is sent to V before the **view change**, then either all $p \in (V \cap V^*)$, receive m , or none do.
- All non-faulty processes in the **same view** get to see the **same** set of multicast messages.
- A message sent to **view V** can be delivered only to processes in V , and not to **successive** views.



8

So let me give an example of virtual synchrony. Let us say that the view here would be P1, P2, P3, P4. So any message will have to be sent to all of them. And so similarly, if I were to consider this message of P2 sending the message, it will have to deliver the message to P1, P3 and P4, which means the rest of the processes.

Now let us assume that there is a failure with process P2. So the view changes to P1, P3 and P4. So in this case, any subsequent message transmission and message receiving that would only go to the processes in a new view, which are P1, P3 and P4. In this case, P1 also sends, it sends to P3 and P4.

Now assume that P2 comes up and P3 goes down. So the new view becomes P1, P2, P4, and any subsequent message transmission is only sent to the members of the new view, which are P1, P2, and P4. Finally, it is possible that at one point in time, denoted by this dotted line, the view will finally change to P1, P2, P3, P4.

So here, if I were to draw a line like this, this would be wrong. And because of, why is that? Well, that is because nothing can be delivered to this process. It is faulty. And also if I were to draw a line like this, that also would be wrong because this message was fundamentally meant for the processes of the older view. So it cannot be delivered to a process in the newer view, which is something that is being said here.

A message sent to view V can only be delivered to processes in V , and not to processes in successive views, which is why this particular message transfer would be illegal. So we consider a view a closed world of its own of where atomicity and reliability guarantees are provided. And once the view changes, all old messages are like all dead. So our time starts from when the view changes.

(Refer Slide Time: 20:53)

Few more assumptions

- A **sender** to a view V should be a member of $view(V)$
 - Many people define virtual synchrony by **relaxing** this assumption
- If a **sender** $s \in V$ crashes
 - First we **flush** its multicast message (if possible)
 - Then **remove** s from V
 - As long as s is a **member** of V , all the assumptions of **virtual synchrony** continue to hold
- If a **receiver** $r \in V$ crashes
 - We can either **deliver** the message **later** (rest of the **processes** in the view have a copy)
 - Or **remove** the receiver from the view.



process groups \rightarrow views
Virtual synchrony is independent of the order of message delivery.

So a few more assumptions. Well, so people in the literature have made two kinds of assumptions. So should a sender be a member of view V , well, many people say yes, many

say no, need not be. So the assumption that we are following is that the sender should be a part of view V .

So now, assume that if a sender, that is part of a view, the sender crashes. So if the sender crashes, well what happens then? What happens is that we need to flush all of its multicast messages that are in flight, if possible, because we wish to provide atomicity guarantees.

So, this will be elaborated. It is a fairly complex process. So I have just summarized multiple slides worth of material into a single line, but this is a reasonably complex process. So this will be elaborated quite a bit in subsequent slides. So first is that we take care of its messages, which means that either messages get delivered to all the processes in the view, or whatever messages are there, they are flushed from the system.

Then we remove s from V . So then of course, we go in for a view change. So as long as $S \in V$, all the assumptions of virtual synchrony continue to hold. And after this is removed, we go for a view change. If a receiver crashes, we can either deliver the message later because rest of the processes in the view have a copy.

So if let us say, it comes back up, we then deliver it later if we are willing to tolerate transient fault or the other is that we remove the $r \in v$, the latter being more common where the receiver is removed and we go for a view change.

So one important point is that here we are simply talking of delivery, and we are not talking about the consistency requirements like causal or sequential consistency. So virtual synchrony as such is independent of the order or message delivery. The only thing that virtual synchrony cares about is essentially these process groups, which we are also calling as views.

So it basically says that message is either delivered to everybody in the current view or to nobody. And the view changes appear to happen to everybody in the same global sequential order.

(Refer Slide Time: 23:40)

Implementation

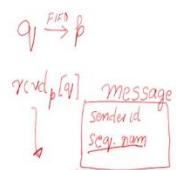
- We **define** $view(p)$ for process p as follows
 - If $p \in view(q)$ then $q \in view(p)$
- Messages **received** by p are queued in $queue(p)$
- If p **fails**:
 - First **flush** the messages sent by p if they are not **delivered** to any process in the view. Ensure all the **outstanding messages** are delivered.
 - Then **remove** p from the **group (view)**
- **Sending a message**
 - Attach a **timestamp** with each message (**increases** by 1 with every send)
 - Assume **FIFO channels** ✓
 - Highest numbered message from q that is **received** by p is stored in $rcvd_p[q]$



11

Implementation

- We **define** $view(p)$ for process p as follows
 - If $p \in view(q)$ then $q \in view(p)$
- Messages **received** by p are queued in $queue(p)$
- If p **fails**:
 - First **flush** the messages sent by p if they are not **delivered** to any process in the view. Ensure all the **outstanding messages** are delivered.
 - Then **remove** p from the **group (view)**
- **Sending a message**
 - Attach a **timestamp** with each message (**increases** by 1 with every send)
 - Assume **FIFO channels**
 - Highest numbered message from q that is **received** by p is stored in $rcvd_p[q]$



11

So how would implement a virtually synchronous system? Well, that is easy to do. And in fact, it is required in any practical distributed system. There is a requirement, and that too a reasonably strong requirement to implement virtual synchrony. So this is how we implement. So first we start with defining view p , for process p as follows.

So if $p \in view(q)$, then $q \in view(p)$, which means the views are identical. So if a given process is in q 's view, then $q \in view$ of that process. So any message that is received by process p is, at the moment, is temporarily kept in a buffer called the queue of, so consider this to be an array, which points to this buffer.

So, any message that is received by p is stored in this temporary buffer. Now assume that p fails. So if p fails, we first flush the messages sent by p . If they are not delivered to any process in the view. So if let us say p has failed, and let us say, p has some of the, some messages that intends to send, and they have still not been delivered to any process, then the messages are flushed because if any, if a single process gets it, all the processes need to get it.

Then we need to ensure that all the outstanding messages, which means those messages that some subset of processes in the view have gotten, they are ultimately delivered to all. Once that has happened, we remove p from the view and we go in for a view change. So the important aspect that we need to consider over here is that if any process p fails, then we need to look at all the messages that it has sent or is intending to send.

So if a message is still there in its output queue, and it has not been sent, the message is flushed. Otherwise, we wait till the message is delivered to everybody. And then we remove it from the group. So sending a message, what we do is we attach a timestamp with each message. It increases by one with every send. So we essentially attach a timestamp and the ID of the sender.

So let us provide a solution to this assuming FIFO channels. So for this, we introduce the data structure received p , q . So the highest numbered message from q that is received by p , let it be stored in this array, which essentially, so, as we have just described, every message has a sequence number. So the structure of a message would look like this.

So if q sends a message to p over a FIFO channel, p maintains a data structure, which is the message received from q with the highest sequence number. The latest, the sequence number of the latest message received from q . So we will see that this helps us ensure enforce virtual synchrony.

(Refer Slide Time: 27:56)

Implementation - II

$q \rightarrow p$

- p periodically sends $rcvd_p[]$ to all the processes in its view
- Each process p records $rcvd_q$ (from any other process q) in an array $remote_p[][]$.
 - $remote_p[q][i]$ indicates what p knows about message arrival in node q
- Consider a sample $remote[][]$ array

Find the minimum in each column

	P_1	P_2	P_3	P_4	
P_1	2	3	1	3	$rcvd_p$
P_2	2	3	1	4	$rcvd_q$
P_3	1	1	1	5	$rcvd_3$
P_4	2	2	1	5	$rcvd_4$
min	1	1	1	4	



12

Implementation - II

- p periodically sends $rcvd_p[]$ to all the processes in its view
- Each process p records $rcvd_q$ (from any other process q) in an array $remote_p[][]$.
 - $remote_p[q][i]$ indicates what p knows about message arrival in node q
- Consider a sample $remote[][]$ array

Find the minimum in each column

	P_1	P_2	P_3	P_4	
P_1	2	3	1	3	
P_2	2	3	1	4	
P_3	1	1	1	5	
P_4	2	2	1	5	
min	1	1	1	4	

msg 2 from P_1 unstable
msg 1 from P_3 stable



12

So, let us now look at the second aspect of the implementation. So p periodically sends a received p , the received p array to all the processes in its view. So recall that the array was defined over here where the received p array pretty much says, what is each column of this array, let us say the q th column, pretty much stores what is the sequence number of the latest message that it has gotten from process q ?

So, p periodically sends this array to all the processes in its view. Each process p records received q , where we are assuming q is a generic process that can send messages to p . So

this is stored in an array remote p. So the structure of the `rcvdp[q]` that it has rows and columns. So if I were to consider the qth row, which in this case is for P2, so it stores a vector of four values, and this is pretty much the received q array that came from q.

So, each element of this row stores the sequence number of the latest message received from the corresponding process, of that column. So let us say that, let us give a practical example of a remote area. So, consider this row over here, which is the row for Process 2. So in Process 2, what this is saying is that Process 2 has received a message with sequence number 2 from Process 1, it has received a message with sequence number 3, from Process 2, it has received a message with sequence number 1 from Process 3, and finally, it has received a message with sequence number 4 from Process 4.

So this is the state. So, so let this remote area be from the point of view of Process 1, where Process 1 = p. From p's point of view this is what P2 has been seen, because this is the last state of the received q array that was sent to p. So it has simply stored it in the corresponding row of its, of the 2d array remote. Similarly, it has something about, P1 has something about P3, and P1 has something about P4.

And so, this would basically be the received 3 array that it got from P3, and this would be the received 4 array that it got from P4. And of course the row that is corresponding to itself is pretty much its own received array. So, this is received p. So, the important point to consider over here are actually these columns, where the columns are telling us something.

So, consider the first column. So this is what all the processes know about messages from P1. So clearly, P1 will have the latest information about its message because it also sends to itself. So, in the column the message with Sequence 2 has been received by P1, by P2, and by P4. However, P3 has not gotten that message. The latest message that it has gotten has a sequence number with 1.

That is the reason when we compute the minimum, the minimum does not match all the entries. There is a diversion. So this message, which is message 2 from P1, this is considered to be unstable. That is because it has not reached all the processes. So it is still

in flight. However, if I consider this message, which is message 1 from P3, so it has reached P1, it has reached P2, it has reached P3, it has reached P4, and the minimum is equal to all the elements of the column.

So message 1 from P3 is considered stable. So message 2 from P1 is unstable, we can clearly see why. And message 1 from P3 is stable, we can also see why, because everybody has gotten it.

(Refer Slide Time: 33:09)

Stability and Flushing of Messages [Details]

- A message is **stable** if it has been received by all the processes in the **view** (refer to the **min. vector**)
- The message can be delivered to the process's **next layer** *(applied to the fsm)*
- To **remove** a process a failure-detecting process needs to multicast a **flush** message to all the processes in the view.
 - All the processes **stop** sending new messages.
 - Processes **send** their **rcvd** arrays to other processes. *(remote)*
 - They also **elect a leader** (coordinator). Once a **process** finds that its messages have all been received, and it has received all the messages it sends a **flush_ok** message to the coordinator. Otherwise, after a timeout it sends its list of **unstable messages**.
 - If the coordinator does not get all the **flush_ok** messages in a given interval, it collects all the **unstable messages**, and multicasts them again.
- After getting **acknowledgements**, it sends a **view_change** message.



13

Implementation - II

- p **periodically** sends $rcvd_p[]$ to all the processes in its view
- Each process p **records** $rcvd_q$ (from any other process q) in an array $remote_p[][]$.
 - $remote_p[q][i]$ indicates what p knows about message arrival in node q
- Consider a **sample** $remote[][]$ array *(rcvd_q[q])*

P1	2	3	1	3
P2	2	3	1	4
P3	1	1	1	5
P4	2	2	1	5
min	1	1	1	4

Find the minimum in each column

rcvd₁
msg 2 from P1 unstable
rcvd₂
msg 1 from P3 stable



12

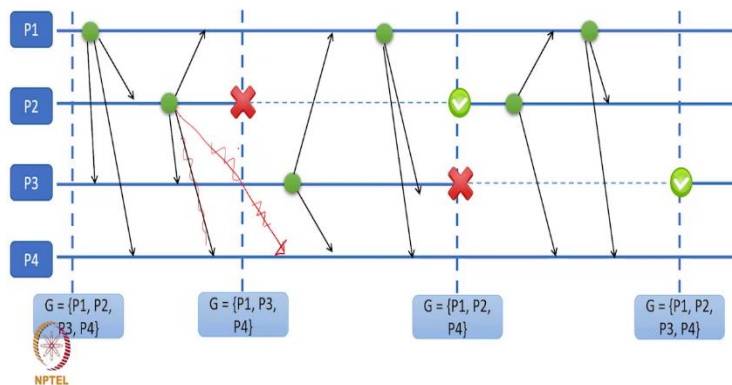
Stability and Flushing of Messages [Details]

- A message is **stable** if it has been received by all the processes in the **view** (refer to the **min. vector**)
- The message can be delivered to the process's **next layer** *(applied to the fsm)*
- To **remove** a process a failure-detecting process needs to multicast a **flush** message to all the processes in the view.
 - All the processes **stop** sending new messages.
 - Processes **send** their **rcvd** arrays to other processes. *(remote)*
 - They also **elect a leader** (coordinator). Once a **process** finds that its messages have all been received, and it has received all the messages, it sends a **flush_ok** message to the coordinator. Otherwise, after a timeout it sends its **list of unstable messages**, *→ coordinator*
 - If the coordinator does not get all the **flush_ok** messages in a given interval, it collects all the **unstable messages**, and multicasts them again.
- After getting **acknowledgements**, it sends a **view_change** message. *(VUV)*

NPTEL

13

Example of Virtual Synchrony



NPTEL

9

So now, let us look at the flush process and the view change process in some more detail. So, till now we have slightly been looking at it superficially, but given that we have introduced these tools, we can look at it in slightly more detail. So, a given message, a message m is stable, if it has been received by all the processes in the current view, refer to the minimum vector.

So, if the minimum matches all the injuries in the column, the message is stable for the remote array. So, the message can be delivered to the process's next layer. So of course, a remote area, the 2D remote area from the point of view of a single process only, but even if a single process knows that all the processes in its view have gotten the message, this

means it is stable. So it can safely be delivered to that process's next layer, or what is called in the terminology of typical consensus papers, is applied to the FSM.

So it can be applied to its, to the FSM it maintains, and a change can be made. Now, let us look at some of the complicated cases over here. So let us say that we have a failure detecting process, and it detects that a process has failed, it needs to be removed. So it multicasts a flush message to all the processes in the view. And a flush message indicates the possibility of a view change.

So the moment that a process receives a flush message, the first thing it does is that it stops sending new messages. So it enters a pause state. The processes send their received arrays to other processes, such that all the processes can update their remote arrays. So all the processes can update the remote arrays and figure out which message is stable and which message is not stable.

So, the important thing to notice over here, and we have also seen this in the Chandy-Lamport algorithm, that the moment a process receives a flush message, it simply stops sending and creating new messages, because now a major change is going to happen. So only the next view will, process will actually be sent. Then, the processes elect a leader or a coordinator.

So, what happens is once a process finds that its messages have all been received, which basically means its latest message is stable. So this is important. So once the process finds that all of its messages have been received by the rest of the processes in the view, which is stand amount to saying latest message is stable, and it too has received all the messages.

So that is why, so, so why is that the case? Well, that is the case because at least the sender always knows or always has the latest message. So just, just look at this, just look at the diagonal elements in this figure, and it will be clear what I am trying to see. So just take a look at the diagonal messages over here.

So, this message is what Process 1 knows about messages it has received, which means that, so this is based on the fact that when Process 1 creates a message, it immediately receives it. So this is the latest state about Process 1, which it keeps. And so similarly, if

Process 2 is saying, is sending a sequence number of 3, it basically means the moment at which Process 2 generated this array received 2, 3 was the sequence number of the last message that it created.

And the same holds for Process 2, the same holds for Process 4. So after a flush message is sent, we know that no new messages are being created. So any received array gotten, any received array that a process gets, which is this array over here, this array over here, that a process gets after processes stop creating new messages, one of the entries, which is the i th entry of the, let us say received q is being sent, then its q th entry is going to be the latest that q has generated.

So this, viewers need to convince themselves of this fact that the q th entry of received q will be the latest because q will at least know what is the latest message it has created. And if it is not creating anymore, so this would be the latest message. And so that is how a process can know if it has received the latest messages generated by all the processes or not.

So coming back to this point, every process would know via exchanging these received arrays whether all of its messages have been received, which means its latest message is stable, and whether it has received all the messages. If both hold true, then it sends a flush okay message to the coordinators, which means that from the point of view of the current view, it is done. It has gotten everything, it has sent everything, and whatever it has sent is stable.

Otherwise, the coordinator waits for some time and the processes, individual processes wait for some time. And if their messages are not becoming stable, then they send a list of unstable messages to the coordinator. So the coordinator takes upon itself, the owner is tasked of ensuring that all the processes sent to a view reach everybody. That would be the atomicity requirement.

So if the coordinator does not get all the flush okay messages, then it will collect unstable messages by wearing the individual processes, adding multicasts all of them once again to all the servers that have not gotten them. And after getting acknowledgements, which

means after ensuring that every message sent in the view has reached everybody, it is time to change the view.

So it send a view, change message to all the servers in $V \cup V^*$, where, V^* is the new view. So the old ones, either remove themselves from the view and the new ones add them to the view. So it sends a view change message and the view is changed. So this ensures the correctness.

So if I were to go back to this figure, so this basically ensures that none of these arrows, none of these solid arrows cross this dotted line arrow. For example, if I were to delete this and I would do this, this would be wrong, this would violate virtual synchrony. And the reason is that this was sent in a previous view and it is being received in the next view.

So this is clearly not allowed. And our method of essentially flushing all the messages and ensuring all of them have been received by the receivers before changing their view would precisely stop this situation from happening. So this was virtual synchrony for us.

(Refer Slide Time: 41:40)

- Virtual Synchrony
- Reliable Multicast

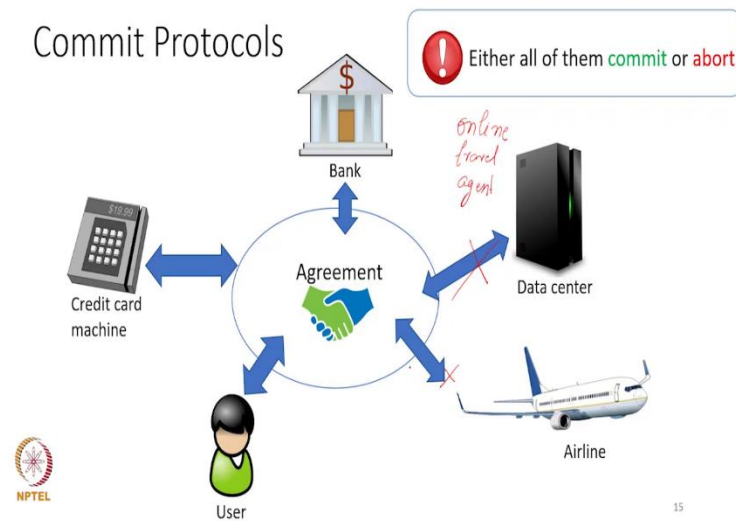
Commit Protocols



So now assume that I have been able to achieve virtual synchrony, which means I can create a stable process group. And I have been able to achieve a reliable multicast, which means that I can send a message to a group of processors using a virtual synchrony kind of

approach. So what is it that I can do with it? Well, I can use this as the baseline substrate to design what are called commit protocols.

(Refer Slide Time: 42:22)



So, consider a typical problem which all of us have faced, and that too it is a very nasty problem. So we have a credit card machine, a user, of course, swipes the card on the card machine, then a bank which provides the credit card functionality, a data center something like a major, like make MakeMyTrip, Expedia which provides a travel, a ticket booking facility, essentially an online travel agent, and then of course the airline.

So, we want to buy an air ticket. So unless there is a complete agreement among these five entities, what is going to happen is that I might charge the card, the money might be debited from my credit card account, the data center might do some processing, but this is where things may fail, and I'll never get the ticket, even though the money has been debited.

So, this is a very common problem, it is a very nasty and very annoying problem. So what most of us do in this case is that we call up the office to the online travel agent, we are lucky if somebody picks up the phone and then after sometimes after 10 minutes, sometimes after half an hour, sometimes after three days, we get the money back. In some cases, we get the ticket, in some cases we do not get the ticket. So that also destabilizes our travel plan, and it is clearly not an optimal situation.

So, what we want is that we want all of them, all of these entities to come into an agreement. Either my ticket is booked or it is not booked. So in that sense, I do not mind if my ticket

is not booked. So I will at least immediately get to know. So I can try with the same travel agent or a different travel agent. I am talking of an online travel agent, a website any travel booking site. I can try 5 minutes later, 10 minutes later, I am okay with that.

What I am not okay with is if my money is debited and I do not have a ticket or I have a ticket, but I do not have an e-ticket, I do not have an email, so I do not have any paper proof or a digital proof of my ticket. So, I do not want that. I am okay to be told, look, our site is down, we cannot book a ticket right now, but I am not okay with anything else.

(Refer Slide Time: 44:57)

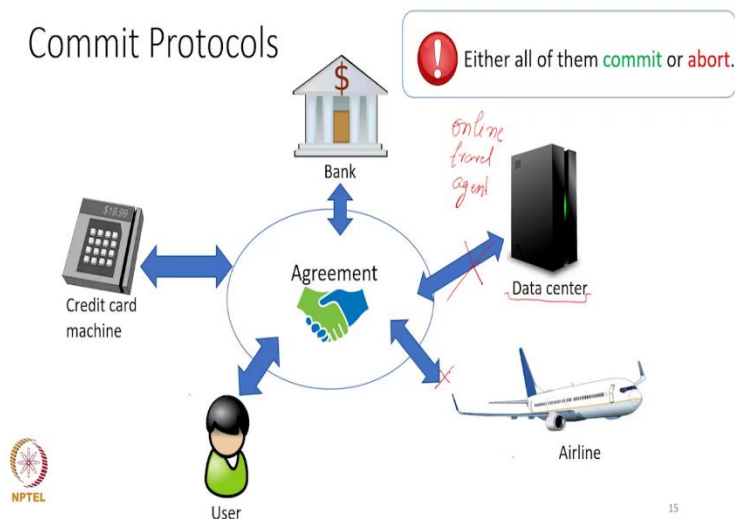
2-Phase Commit

- The nodes elect a coordinator
- **Phase 1a:** Coordinator sends a Vote-request message to all participants.
- **Phase 1b:** A participant returns either Vote-commit or Vote-abort.
- **Phase 2a:** Coordinator collects all the votes. If all are Vote-commit it sends a Global-commit message, otherwise it sends a Global-abort message to all.
- **Phase 2b:** Each participant waits for the messages from Phase 2a. It acts accordingly.



16

Commit Protocols



15

So, the first algorithm in this space is called 2-Phase Commit. We will see it as some problems. So the nodes elect a coordinator, they elect leader using any of the leader election algorithms that we have seen. The coordinator in this case, asks all the nodes to vote. So, so the nodes would be each one of these, the user card machine bank, data center, and airline. Of course the user is being represented by the command running on his BS, on this browser.

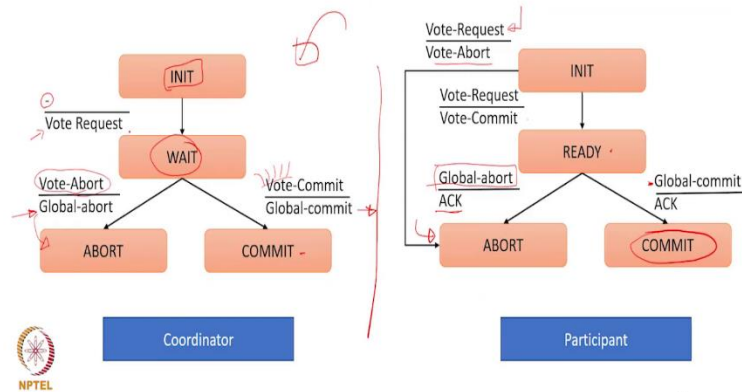
So, the entities either say Vote-commit or Vote-abort. Vote-commit means we are okay with the transaction. So for a user, it would mean pressing the submit button, for a credit card machine it would mean that the card details have been read correctly. For the bank, it would mean there is sufficient money in the credit card account, for the data center, it would mean that they have been able to process the details, and for the airline, it would mean that they have been able to book the ticket.

Once all of them say yes, that would mean all of them are voting Vote-commit. If one of them has a problem, let us say the airline does not have tickets, it simply sends a Vote-abort, and then the entire process gets aborted. So the coordinator collects all the votes. If all of them are Vote-commit, which means the ticket can be booked, it sends a Global-commit message to all the nodes and tells them, okay, go ahead and commit. Commit means issue the ticket.

Otherwise, it sends a Global-abort message, which basically means that you just decline this transaction, and the user is informed accordingly that the ticket could not be booked. Each participant, participant waits for the messages from Phase 2a. It acts accordingly.

(Refer Slide Time: 46:58)

2-Phase Commit



So, the 2-Phase Commit state diagram would look like this for, so this part is for the coordinator. So coordinator starts at an INIT state. So initially nothing happens. It just sends a vote request. Once the participant that also starts in its INIT state, once that gets, once the participant gets a vote request, the participant decides what needs to be done.

If it decides to abort the transaction, it sends back a Vote-abort message, and it moves to the ABORT state. Otherwise, if it decides it needs to commit, it sends a COMMIT message, and it goes to the READY state. So mind, in the READY state, it is not aware of what the other participants have sent. So, it waits.

So, the coordinator also waits once it gets all the messages, if one of them is Vote-abort, it goes to the ABORT state, and it sends a Global-abort message to all the participants. So, which means that the participant gets a Global-abort message. So the participants just abort. So this means that any of the nodes over here has a veto power, and it can simply say no, and the entire process gets aborted. So, as I said, we are fine with that as a business model.

And if all the participants Vote-commit, so once the coordinator sees that and all are saying Vote-commit, it sends a Global-commit message and it moves to the COMMIT state. Once the participant gets the Global-commit message, well, it sends an acknowledgement and it moves to the COMMIT state, which means it finishes or commits its transaction.

So this is a fairly simple state machine over here, and this works in most cases, but the reason that it is not used is basically when we consider failures. So after all, why did we do this? The reason we did this is basically because any of the nodes in here might have an issue with the transaction. So it would then send it would then send either a COMMIT or an ABORT notification.

And if there is a single ABORT, the entire transaction would get cancelled. So, the reason we have problems, particularly with this is because any of these entities can fail. So failures are something we have to consider. And most often, we have network failures, we have time-outs, all kinds of things.

(Refer Slide Time: 49:53)

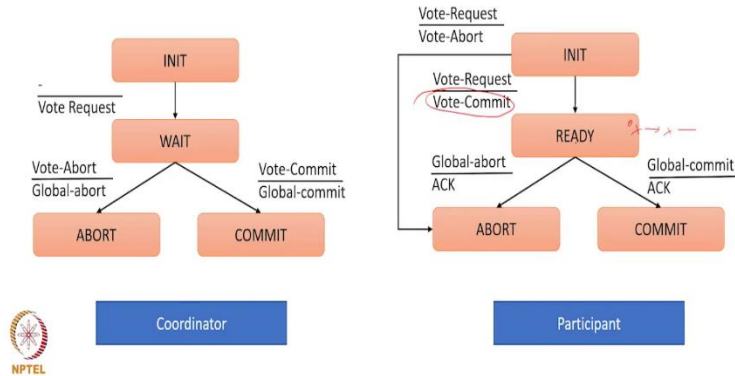
Analysis of 2-Phase Commit

READY → X → READY

- Let's say a participant fails, recovers and then restores its old state.
- **INIT state**: No problem. It just aborts.
- **READY state**: It needs to know the global decision (**commit** or **abort**) after it recovers. Ask the coordinator or other participants.
 - This is very **problematic**.
 - The coordinator might have **failed**. Other processes might have **committed** or **aborted**. They might have **erased** all the history of the transaction. **Blocked**
 - We will never **know** what decision the coordinator took.
- **ABORT state**: Complete the abort process.
- **COMMIT state**: Complete the commit process.



2-Phase Commit



So in this case, let us say a participant fails, and there is some amount of non-volatile memory. So the participant recovers and restores itself to the old state. So the participant recovers and restores itself to the INIT state with no problem, it just aborts because from this state, nothing really has been sent. So which means that I was there in the INIT state, and I just, and then I recovered in INIT state, which means I have not sent any message.

So you have not sent any message, the coordinator also cannot proceed. So then I can just abort the entire transaction. That is simple. The main problem is the READY state. So in the READY state, what happens is I was there in the ready state, which means I sent either ABORT or COMMIT decision, and then I crashed and then I woke up and I woke up to be in the READY state.

So if I had, earlier, if I had sent the ABORT message, well, that is fine. Then the coordinator would have aborted, so I can abort. But if I had sent a COMMIT message, a Vote-commit message, it kind of becomes problematic. So what I would do is that I would not know what happened to the system after I crashed. And I might wake up like 2 minutes later, which is a long time in the computer scale.

So then I would ask the coordinator, and if the coordinator says we have all aborted or all committed, I would do the same. But the problem is the coordinator itself might have failed,

or the coordinator might just have finished the transaction and erased its state. Furthermore, even other, so then what I would do is I would ask other processors.

Other processes might have committed or aborted. They also might have erased all the history of the transaction. So I would not know what to do. So I would kind of get blocked. Because the thing is, I am recovering in the READY state. So kindly take a look at this. So I am recovering in the READY state. So when I am recovering in the READY state, I have already sent a Vote-commit message.

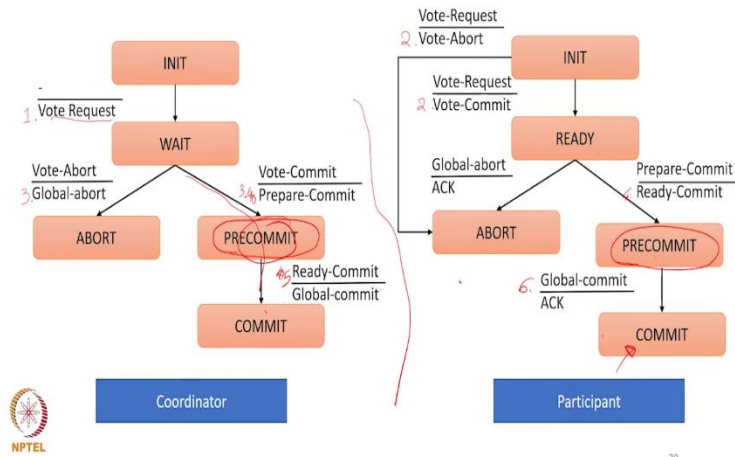
And then, well, I do not know what is to be done because I crash and then I wake up. And then I do not know what the others decided. Of course from INIT if I had sent a Vote-abort, then that is okay, I would have simply gone to the ABORT state or if any state, any intermediate sub-state, I would have recognized that everybody is aborting, I would have aborted.

But in the READY state, I really do not know what is to be done because everything depends on what the others have voted. And if I am not able to contact the coordinator or another process, then I really do not know what happened to the system. So I will keep on, I will remain blocked forever.

So this is just called a blocking protocol, and it does not give us a deterministic answer, which is why the 2-Phase Commit protocol is not actually used. And of course, if I even fail in ABORT or COMMIT state, that is not an issue because I already know the state of the system. So to fix this problem, the problem of crashing and recovering in the READY state, what I do is that I propose a new protocol where I can show that diagram at first, which is called 3-Phase Commit.

(Refer Slide Time: 53:52)

3-Phase Commit



So instead of two phases, I have three phases. So in this case, I add an extra PRECOMMIT state between READY and COMMIT. So between READY and COMMIT, I add an extra PRECOMMIT state, which in a sense increases my distance from, increases the distance from the wait to the COMMIT state, which we will see is important.

So in this case, what we are actually doing is that if I were to compare this diagram with the previous one for 2-Phase Commit, there is an extra state in the middle. We call this the PRECOMMIT state, and we claim that by adding this PRECOMMIT state, many of our problems have been solved. So let us first go through the protocol and then we will discuss its advantages.

So, it starts in exactly the same way, the coordinator sends a vote request message, which is step 1. Each of the processes gets the vote request message, so they either send a Vote,-abort, which immediately takes us to the ABORT state. So this part remains the same. Or, the, we, so this is also a time step 2, the participants send a Vote-commit message and they move to the READY state.

So what happens is this part is the same that they either vote a Vote-abort or commit, and if they Vote-commit, they move to the READY state. So when it is in the READY state, the participant keeps on waiting for messages. If it gets a Vote-abort, then of course it sends

a Global-abort. Or, if it sends, or if it gets all the replies and all of them are a COMMIT, then instead of directly global committing, it sends a Prepare-commit message.

So this should also be time step 3. So it is a sends a Prepare-commit message and it transitions to the PRECOMMIT state. So once a participant gets a pre, Prepare-commit message, so then it sends a Ready-commit message, which means, it is saying that I am ready to commit. That is mainly because that it is already voted that it will commit, so it cannot change its mind. So it will send a Ready-commit message, it will go to the PRECOMMIT state.

Once all the Ready-commit messages have arrived, this should be times step 5. Once all the Ready-commit messages have arrived from all the participants, then the coordinator knows that all the participants are ready to commit. And so then it sends a Global-commit message. And the Global-commit message, once it is received, all the participants commit and send an acknowledgement as well.

So the important point to note over here is just this additional state in this path, which is a Prepare-commit. And when the coordinator receives all the Vote-commits, instead of directly committing, it sends a Prepare-commit message, which also takes all the participants to the intermediate state called PRECOMMIT. And from there, once they receive a Global-commit message, they finally commit.

So this is the algorithm that each of the participants has to follow. And so this part of the algorithm was for the coordinator, this is for a participant, they look more or less similar, just with this extra state.

(Refer Slide Time: 58:03)

3-Phase Commit

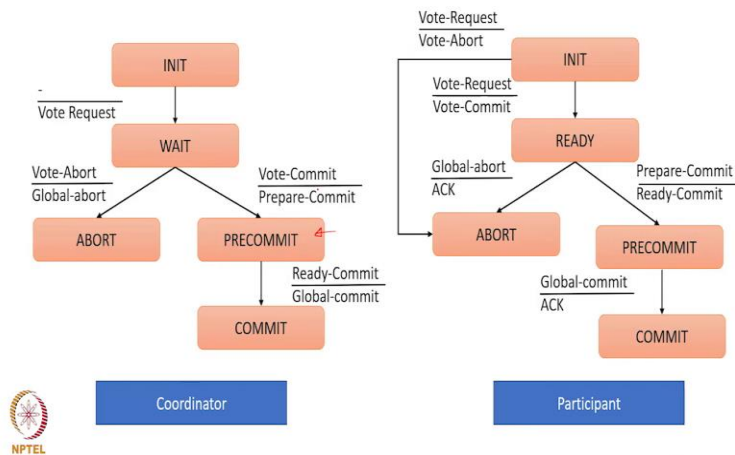
- The nodes elect a **coordinator**
- **Phase 1a**: Coordinator sends a **Vote-request** message to all participants.
- **Phase 1b**: A participant returns either **Vote-commit** or **Vote-abort**
- **Phase 2a**: Coordinator collects all the votes. If all are **Vote-commit** it sends a **Prepare-commit** message, otherwise it sends a **Global-abort** message to all.
- **Phase 2b**: Each participant waits for the messages from Phase 2a. If it gets a **Prepare-commit** message, it proceeds to send a **Ready-commit** message, else it aborts.
- **Phase 3a**: After the coordinator gets all the **Ready-commit** messages it sends a **Global-commit** message to all the participants.



- **Phase 3b**: The participants **wait** for the **Global-commit** message.

19

3-Phase Commit



20

This is summarized in this slide so I am not going over it once again. So, this is something that I just described. The operative part of it is that we actually send a Prepare-commit message that takes us to the PRECOMMIT stage, and then after a Prepare-commit message, we move to the final COMMIT, after the PRECOMMIT state, we move to the final COMMIT state. So this extra state over here is what we clean does all the magic. So how does it do that? Well, let us see.

(Refer Slide Time: 58:53)

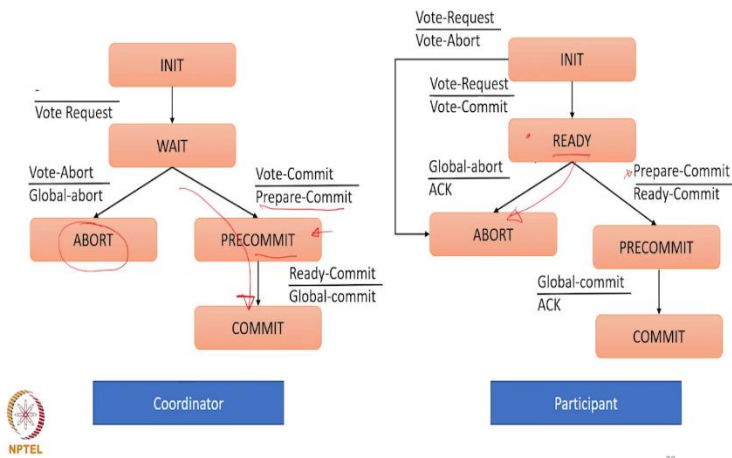
Analysis

- Consider the **READY** and **PRECOMMIT** states.
- If a **participant** fails in the
 - **READY** state: It can ask the **coordinator**. If the coordinator has **failed**, then we can elect a new coordinator. If any process has gotten a **PRECOMMIT** message, then they proceed towards a **global commit**. Else all processes **abort**.
 - **PRECOMMIT** state: Elect a new coordinator that will send a **Global-commit** message.
- Coordinator fails in the
 - **WAIT** state: Participants time out in the **READY** state
 - **PRECOMMIT** state: Participants time out in the **PRECOMMIT** state



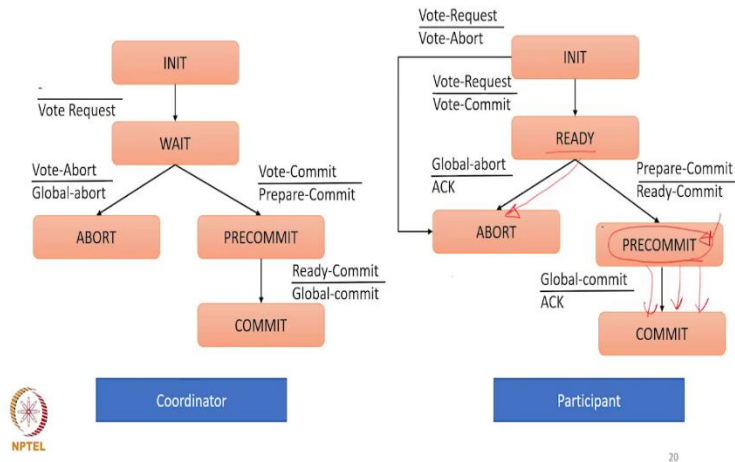
21

3-Phase Commit

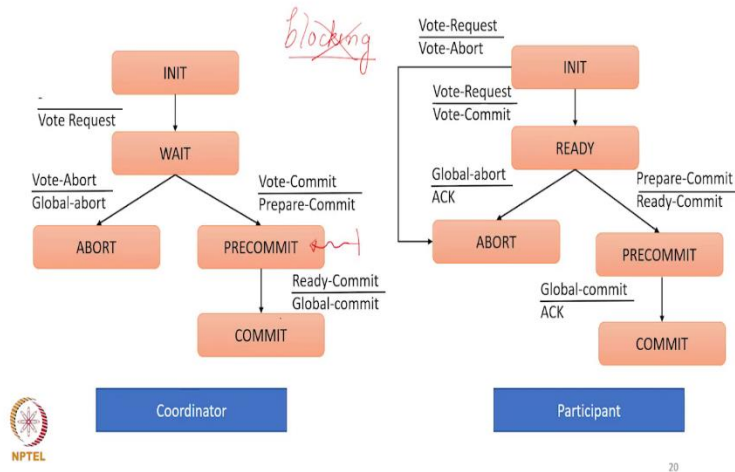


20

3-Phase Commit



3-Phase Commit



So consider, let us do a similar failure analysis where the participant fails and then it recovers to the same state. So, it is assumed that the participant maintains some degree of non-volatile memory, which allows it to recover to the same state. So consider a participant's failure in the **READY** state. So if it fails in this state, so it knows what it has voted for, it has clearly voted for **COMMIT**.

So then it asks the coordinator, once it gets up, what really happened? Did you **ABORT** or did you **PRECOMMIT**? So if the coordinator has failed, then we can elect a new coordinator and the new coordinator can check that if any process has gotten a **PRECOMMIT** message, then it will proceed towards a **Global-commit** via this route.

But, if let us say after querying all the processes, nobody has gotten a PRECOMMIT, which basically means that no process has, so the important point to keep in mind, so this is the most important point that in the previous case, what would happen is that if I woke up in the READY state, the rest of the processes would have either aborted or committed. So essentially, they would be done.

In this case, only the processes that would have aborted would have been done, but the rest of the processes would actually be there in the system, they would not have completed because the thing is that after READY, a Prepare-commit message has to be sent, which has not been sent. So those processes would still be there. this is what makes the entire difference.

Since those processes are still there, they can be queried about their current state. And if all of them are in their READY states, none of them is in their PRECOMMIT state, then it can safely be assumed that the coordinator crashed before sending the Prepare-commit message. And then all the processes can move from the READY state to the ABORT state. So they can just be safely aborted.

So this is the main advantage of this protocol that even if the coordinator fails, even if a process fails we will still not have a situation where we need to block. We can always make a decision and the decision in this case will be correct. So let me go over this, this is the prime primary advantage of the 3-Phase protocol, which is that as compared to the 2-Phase protocol, where if I woke up in the ready state.

There is a possibility that the coordinator would have failed and the rest of the processes would have just completed their execution. They would have either aborted or they would have committed so that, like the protocol would be over and I might not get what is the current state. In this case, only those processes, so only the processes that would have aborted would be out of the system, if there are any, otherwise, all the other processes, if I am moving towards the COMMIT would be there in the system.

So they can always be asked for their current state. And if all of them are READY, we can safely assume that the coordinator crashed before sending a Prepare-commit message. So

then we can simply abort all the processes and take them to the ABORT state. And that would be correct. So, so what again is the key, key, most important insight?

The most important insight is that a PRECOMMIT state over here stops a process from actually leaving the system. So the processes will still be there in the system. And because they are there in the system, we can query their state and figure out what the coordinator must have done.

If I fail, and then I, so I being a process, fails, and it wakes up in the PRECOMMIT state, well, no problem. This means that it must have sent a Prepare-commit message from READY and gotten a Ready-commit in return. So in this case, all that needs to be done is we need to elect a new coordinator that will send a global commit message and commit all the processes.

So, they do not move automatically. It is still done via coordinator to ensure that everybody commits. And the coordinator sends the Global-commit message to all, and ensures that all the processes commit. And in this case, the reason we can safely commit is because all the processes have already agreed that they want to commit. That is why they have come via this route.

So all the processes have agreed that they have a desire to commit, and we have also verified that all of them have a desire to commit. So that is the reason the process entered the PRECOMMIT state in the first phase. So that is the reason we just need to commit all the processes, which can easily be done.

Now, the coordinator fails in the WAIT state, so the coordinator would fail in the WAIT state. This means that it would have not gotten all the Vote-commit or Vote-abort messages. So the processes will timeout in the READY state and ultimately they will elect a new coordinator, which will take a decision accordingly.

If the coordinator times-out in the PRECOMMIT state, well, that also means that the coordinator is in this state. So which means it has sent all the Prepare-commit messages, so all the participants will also be there in the PRECOMIT state, they will timeout, they

will elect a new coordinator, which will commit all the processes. So as we can see this algorithm has inherent advantages, the 3-Phase Commit algorithm.

And it stops the, the biggest problem that it stops is that it stops the blocking problem of 2-Phase Commit. So 2-Phase Commit had a problem of blocking. So this particular problem is not there in the 3-Phase Commit protocol. So, this does not have that problem. It is always possible to make some decision, which is correct. And the entire advantage is accruing from one additional state that we added between WAIT and COMMIT.

(Refer Slide Time: 1:06:00)

References

1. Tanenbaum, Andrew S., and Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.
[Thanks to Prof. Martin Steen for allowing us to adapt his original slides.]
<https://www.distributed-systems.net/index.php/books/ds2/>



So this completes the lecture. So the original slides were in Andrew Tanenbaum Maartenn Steen's book, and they have been very grateful to allow me to present the slides in my class. And you can always find the original slides on this link. So Virtual Synchrony and 2-Phase and 3-Phase Commit protocols are otherwise also very popular. So the viewers will get a lot of resources on the web that describe these protocols in detail, and also the Cornell University has a toolkit that implements Virtual Synchrony. So that toolkit can also be used for physically, running a system with Virtual synchrony.