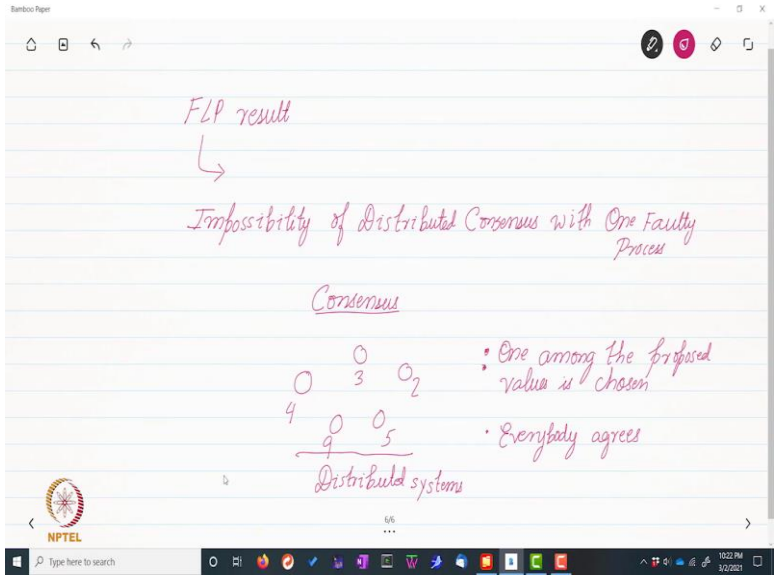
Advanced Distributed Systems
Professor Smruti R Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Lecture No 11
Impossibility of Consensus with one faulty process

(Refer Slide Time: 0:17)



Welcome to the lecture on the famous FLP result. So, the FLP result, the credit goes to three people, and I will tell you in a second, what the result is. But first some honour will give to the people who have arguably discovered the most important result in Distributed Systems. So, the people who deserve the credit are Michael Fisher. So, that explains the F Nancy Lynch for L and Michael S. Patterson for P.

So, the FLP result basically says that to solve the consensus problem in a distributed setting. Even if we have one faulty process, it is not possible. So, the paper is titled Impossibility of Distributed Consensus with one faulty process. So, we will discuss in a second, what exactly Distributed Consensus is and why exactly are we so bothered about it? So, let we first discuss what is consensus and then what is distributed consensus?

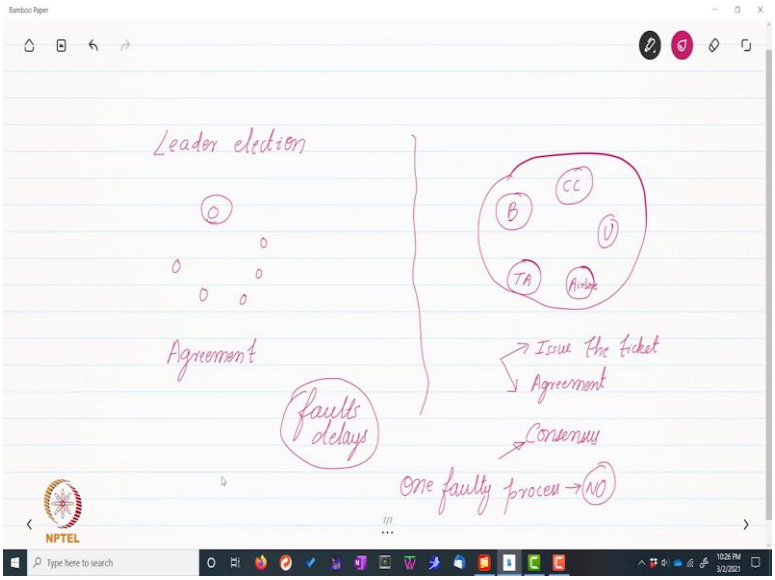
So, consensus basically means that there are a set of process and each process proposes a value. So, let us say that they propose an Integer. So, one process, one of these circles, which is a process, one of them proposes 4, then 9, let us say 3, 2 and 5. So, consensus basically says there is two properties. So, the first property would be one among the proposed values.

So, one value among all the proposed values is chosen. So, it cannot be a separate value. So, one among them is chosen. That is, and second one is that everybody agrees. So, this can be thought off as an agreement problem as well something that everybody agrees.

So, why is this problem so important? Well, the problem is important basically, because if we consider each process to be a distributed process and we consider all of this to be a distributed system, then what it basically means is that we want to make a set of distributed processes, agree on some common thing on something of common interest and will in a second, see what they are, but basically it is an agreement problem where there is no centralized entity. All that we can do is we can send messages between processes, the way that we have been doing.

And we also know that messages may get delayed and that to get delayed indefinitely for a very long time. So, in such a noisy environment, we want to ensure that consensus holds, which means one among the proposed values is chosen and everybody agrees. So, why is this such a big deal?

(Refer Slide Time: 04:09)



So let us look at two examples of why this is such a big deal. So, let us go back to one of our problems, which was leader election. So, leader election, what were we doing? We had a set of nodes. Each node said that it is the leader, or at least wanted to be the leader. Finally, we had an election between them basically by sending messages between the nodes.

And ultimately one node was chosen as the leader. And everybody else agreed that this node should be their leader. So, this is in a sense exactly the consensus problem, that one among the proposed values is chosen and everybody agrees. So, as you can see, there is complete

agreement. So, agreement is there no doubt. Furthermore, one among the values is chosen and leader election is definitely an instance, a specialization of the general consensus problem.

So, let me give another example. So, let us say that, I am the user and I make a credit card transaction to buy an airline ticket. So, that automatically involves the bank as well. That automatically involves the travel agency, the site that I am using to buy the ticket. And that involves the airline as well. So, the airline as well is involved.

So, the point is that all of them, all these five entities need to agree on one thing either they agree to issue the ticket all five of them. This basically means that if my money is deducted, then the ticket should be issued and I should have the ticket with me. It should never be the case that money has been deducted from my bank account or my credit card account. But my ticket is not there with me that should not happen.

And furthermore, it should also not be the case at all, five of these entities, which means me the credit card, the bank, the travel agency, and the airline, all five of us agree that I should be given the ticket, but ultimately it is not issued.

That is also not allowed because a consensus is to choose one value among the set of proposed values. You cannot parachute a value from somewhere else. So, which means that, a broad agreement, consensus like agreement is required. So, these are also called transaction commit systems, which we will study later, which form the core technology of most banking, finance, almost everything to do with Distributed Systems. This basic transaction processing system, even in large databases is required.

Again, what is the core problem? The core problem is a consensus problem. So, similarly it has been identified that a large number of problems in Distributed Systems and Concurrent Systems are essentially specializations of the mother problem, which is the consensus problem. As a result, the consensus problem has a very special place.

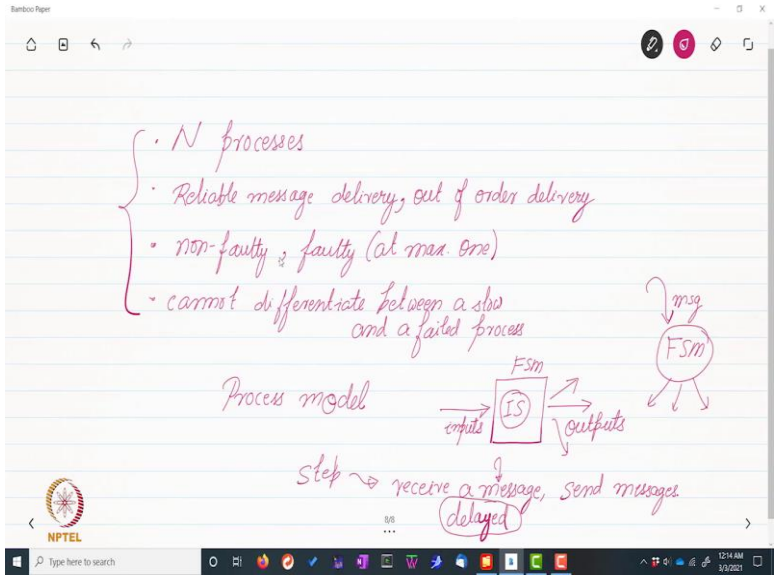
Now, when we want to achieve a consensus, some sort of an agreement in a real-world scenario, we will have faults in the sense we might have network links that die. We might have nodes that die. We might have indefinite delays, or we just might have slow machines. We just might have a machine, which for some reason is congested is taking too long to reply.

So, in such a scenario, the question is, can we achieve consensus? Can we achieve an agreement? Can node decide on something? So, the answer is, it is an unfortunate answer, but the thing is even if you have one faulty process. So, we will formally define what a faulty

process is. But the idea is that even if you have one faulty process, the answer is no, it is not possible.

And this is the famous FLP result, which is kind of the underpinning of most consensus systems as of today. And the sad part is that still it is not doable. Nevertheless, a lot is done to kind of get close, which we will discuss in subsequent lectures, but we are starting from this summer note over here.

(Refer Slide Time: 08:42)



So, given the fact that we have described the consensus problem, let us now describe our system model that under what kind of assumptions are we trying to solve the consensus problem? So of course, we assume that we have N processes. Between any pair of processes see any pair of processes can communicate. So, there is a channel between them.

So, we do assume reliable message delivery in the sense messages are delivered correctly. Their contents are not modified, but it is just that messages may get indefinitely delayed. Nevertheless, no message is ever dropped. I mean, unless the node is faulty, so, we will come to that. So, the issue is that we have in our system among the processes, we have regular processes, normal processes, or non-faulty processes.

So, non-faulty process, what it can do is that it can take an infinite number of steps in the sense that as long as messages keep coming, it can process them as compared to that, a faulty process can take a finite number of steps and then it will simply stop responding. So, you can say that it is slow or it is dead. So, we have no way of differentiating. So, we cannot differentiate between a slow process and between a slow and failed process, a process, which has failed.

So, it is not possible for us to differentiate between them. So, as far as we are concerned, the process is slow, but whether it is slow or whether it has actually failed, that we do not know. So, given the fact that we have this, and then of course we have another thing about the channel the NOC is that we can have out of order delivery in the sense that we do not have the FIFO property.

So, we have out of order delivery of messages in the sense that if one message was sent, it could be delayed for a long time. So, that assumption is something that we make. So, these are standard assumptions where we have N processes and we have faulty processes, but, at max one at the most let us call it at max one faulty process, the rest being non-faulty, we do have reliable message delivery in a sense, messages are not corrupted. It is just that they can get delayed.

And for a certain process, the faulty one, we cannot say that it is slow or it is dead or something. And the same holds for others as well, that there is no time base. So, we cannot say, if a process is just slow or slow to respond or dead or it is not possible for us to say, so given that we have that, we should now look at the process model. So, the process model in a distributed system looks something like this. So, every process in a distributed system gets a set of inputs. And then it produces outputs.

So, it has an internal state. So, as far as we are concerned, what the process basically does is, it is kind of like a finite state machine. So, the finite state machine essentially takes in a message, changes the internal state. And then in response, it can send messages to a few, a few other nodes. And so, the outputs are basically output messages that are sent to other nodes.

So, then the process goes back to sleep. So, it is basically a finite state machine that is woken up by a message. In response, the internal state changes and other messages are sent out to other processes. So, that is basically the model that we have. And then of course we define the notion of a step in such a process.

So, the notion of a step basically means that we can either receive a message from the network in a single step, or we can send an arbitrary number of messages so we can receive a message and is assume that we instantaneously update the state or we can send messages. So, messages can be sent.

So, when messages are being sent, the atomic broadcast capability is assumed. So, the atomic broadcast basically says that if one non fault process receives the message, then all non-faulty

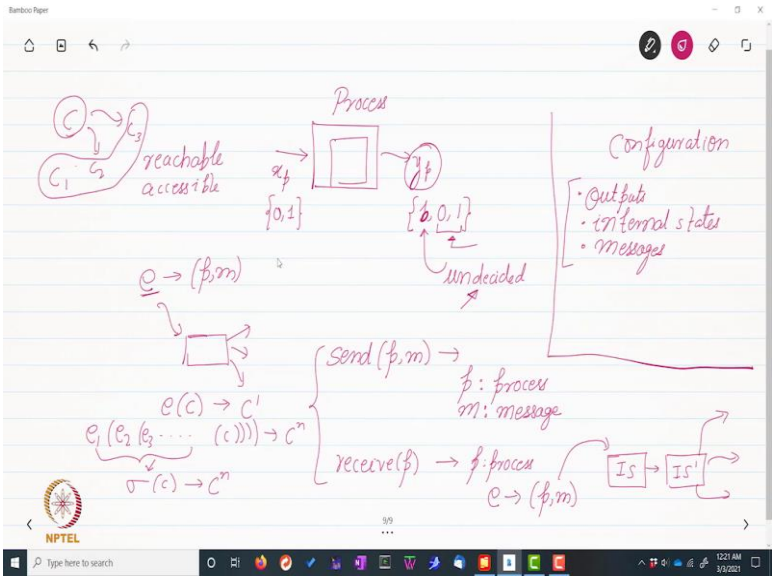
processes will ultimately receive the message. It is just that we are talking of eventual delivery of the messages. And we are not really saying that they will be delivered by this time.

So, messages needless to see can be delayed. So, they can be delayed for sure. And furthermore, these delays also can be arbitrarily long and these can cause the entire algorithm to wait indefinitely in the sense, wait for a very long period.

So, given the fact that we have seen this fair message delivery times, and even no response times are not deterministic are not bounded. Here as I said, out of N processes, one of them will be faulty. So, faulty basically means that within the scope of the algorithm, or let us say in the protocol for all possible inputs, it will only take a finite number of steps. And at one point it will stop. It will just not take steps to take steps basically means either receiving a message or sending a message.

So, receiving a message automatically includes processing it and changing the internal state. So, that is the reason I am not talking about changing the internal state as a separate step, but essentially the step is receiving a message from the network from some other node and sending messages. So, both of these things might get delayed. So, that is the reason a node may appear to be unresponsive to the rest of the system.

(Refer Slide Time: 16:19)



So, given that, that is happening, let us now look at our consensus protocol. So, in the consensus protocol for every process, this is a model that we assume. So, we assume that it is input is a single bit register Xp. So, the single bit input can either be 0 or 1, which pretty much corresponds to the value that is being proposed. So, the value that is being proposed is 0, this could be 0, or it could be 1. Then of course we have some internal state, which can be as much as you want. And then there is a output.

So, in this case, this output is a final output. It is not the messages that we send. So, the final output is what we decided. So, this is essentially a process as a part of the consensus algorithm, what it does. It takes in an input since as many messages as you want. Finally, the output has three possible values. The first one being the most interesting. So, b means undecided.

So, undecided means that I have not decided on 0 or 1. So, as far as I am concerned, the consensus has not been reached and 0 or 1 are the regular 0 and 1, they mean that a consensus has been reached. So, what our aim will be or what we want to show that if there is one faulty process, which takes a finite number of steps, that we have defined it and then appears to fail the, all the processes, at least the non-faulty ones, they will appear to be undecided in the sense they will not be able to make a decision.

So, what we further say is that we need not. So, also the other thing is this is the right ones register in the sense that we cannot update its value. So, now given this process model where we know what the inputs are and what the possible outputs are, let us look at how exactly processes will communicate. So, processes will communicate by sending two kinds of

messages between them. One is send (p,m). So, in this case, p is a process is a process ID rather, and m is a message.

And similarly, we have received p that is other function that processes can call. So, this basically means that we receive a message from process p say again, p is the process ID. So, these are the two basic functions that processes will call. And then let us say that after receiving, we get what is called an event, an event is basically from a given process and a message.

So, it is basically a process ID and a message pair. So, this event is applied to the internal state of the process based on this, the internal state changes to a new internal state. And furthermore, this can lead to messages being actually sent.

So given that we have seen this, we are now in a position to define something called the configuration and the configuration is a key point in our algorithm. So, it is one of the most important inputs that we will use. So, configuration is basically a configuration of the entire system and what the configuration essentially contains is that it contains the following. So, it essentially contains all the outputs, a union of the internal state of the system, for all the processes.

So, the internal states, and along with that, all the messages in flight. So, basically all the messages. So, this to us is a configuration. And as we have seen, every process has an input, has an output and an internal state. If I just take a union of this across all the processes. So, that would include their outputs. If they have decided something or not their internal states and whatever messages that are currently in flight that are yet to be delivered. If you look at that, we will call this configuration.

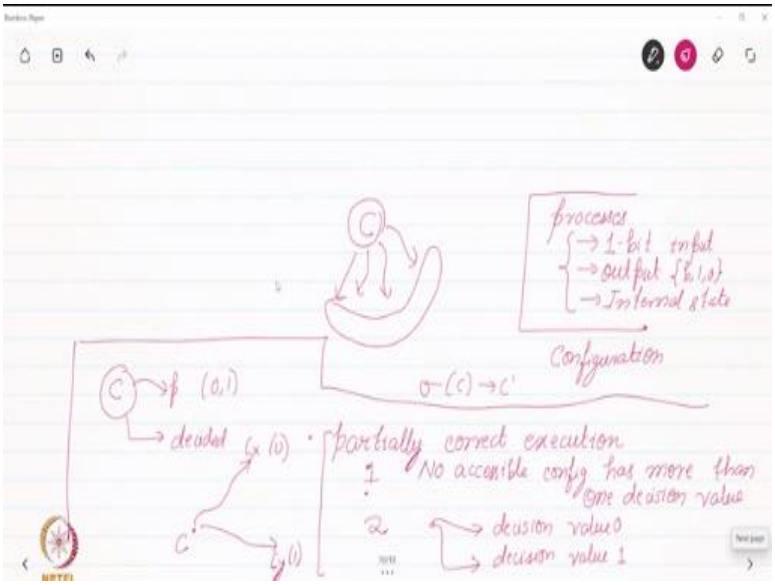
So, every time a certain event, and event is the same as essentially a message with the process ID, along with it, every time that this is appeared, this is applied to a node, which means a node. This message is sent to a node, the node will of course, update its internal state and send messages. And this event will also cause the configuration to change the global configuration to change, the way that we have defined.

So, when I look at the global configuration, what I can see is that the configuration is at, as it is, but once let us say a message is sent to it, so this message is this event over here, then I will apply this event to the existing configuration to get new configuration C'. So, this is known as applying. And if I apply several events in the sense I present these events to the processes, so then it will be some structure like this.

And finally, we will have some configuration, let us say Cn. So, the important thing is when I am applying a series of events in sequence, we can call this an event schedule is essentially a finite sequence of events. So, we can replace this sequence over here with the term Sigma. And we can just say, sigma C is Cn. So, basically Sigma is essentially a sequence of events that is applied to a configuration to get a new configuration.

So, as we can see, given a configuration for different combinations of events, we can arrive at all kinds of new configurations. So, all of these configurations that are arrivals from C are set to be reachable configurations. So, reachable or alternatively they are also called accessible configurations.

(Refer Slide Time: 23:27)



So, this basically means that I take one configuration C of the entire system, and then I apply all kinds of events to it. Whatever I can reach is my reachable set or accessible set. So, given that I have this, we are pretty much at a position to make to define our final correctness criteria. So, at this point, what we can do is, so, what have we looked at up until now, if I were to summarize, we have looked at process. So, specifically we have looked at their one-bit input, the output, which also includes the undecided state that is very important.

The internal state, and of course, a combination of all of these, essentially a union of all of these across all the processes, which leads to a configuration and the way in which if you can apply events to a configuration, it will become another configuration to apply the event sequence Sigma, to see so it will become some other configuration C'.

So, now we are in a position, as I said, to provide some definitions that will take us to our final notion of correctness. So, the first thing that I would like to say is that a configuration C has a decision value V if some process has already made its decision. So, let us say within the configuration, there is some process p whose output is already a 0 or 1. It is not undecided, and it has finalized the output.

So, finalized output, meaning that is a result of the consensus. See if this has been finalized, we say that the configuration C has decided and whatever output any one process has decided that will be the decision value of the consensus, of that configuration as well. And clearly multiple processes cannot decide different things.

So, now let us use this fact to define what is called a partially correct execution. So, partially correct execution has two properties. The first property is that no accessible configuration has more than one decision value which means that if I start from an initial state, then no accessible configuration has more than one. So, this is something that we just said.

So, what we basically said is that, look, if let us say a configuration has more than one decision value, then there is clearly no consensus. So, that is something that is not allowed. So, this is property 1 and a second property is like this, that let us say, if I start from the, from an initial configuration, then there is some accessible configuration that has decision value, 0. And there is some accessible configuration that is decision value, 1.

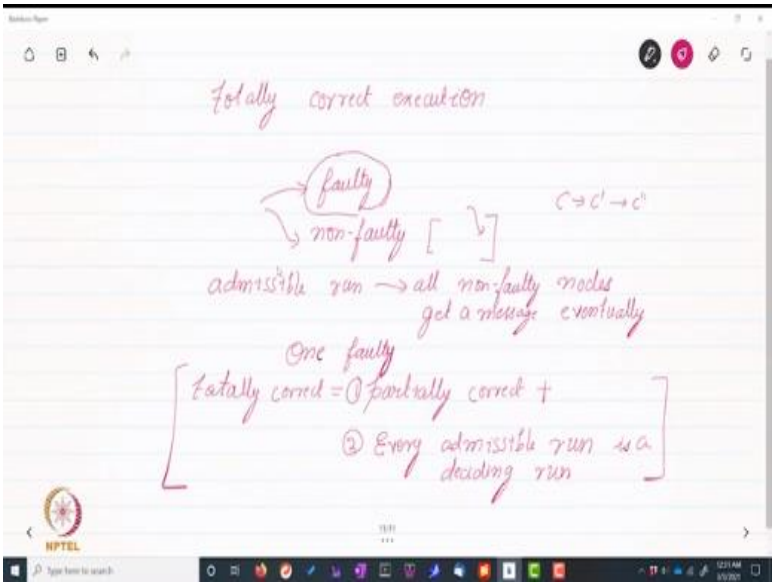
So, this basically means that it is not a fixed match in the sense that when I am starting, I have access. Some configuration is reachable, which is going to decide 0. And something is reachable, which is deciding 1, which basically means that I am starting with an open mind. I am not committing myself to any particular outcome. I am simply saying that, look, this is where I start, this is my initial value.

And essentially all configurations are accessible, accessible in the sense I have path of events exists that can take me from the initial configuration, let us say C to some other configuration, let us say Cx, which decides 0 and some other configuration, let us say Cy, which decides 1 in a sense, both are possible.

So, both of these are essentially saying the first one is a clear-cut consensus condition that in multiple things cannot be decided. And the second one is more like a sanity check that it is not a, it is not like a fixed match. So, when you begin, you start with an open mind that either you

can decide 0, or you can decide 1. So, given that we have this, we can extend a partially correct execution to what is called, totally correct execution, which is something that we are after.

(Refer Slide Time: 29:24)



Let me write it down. So, totally correct execution does take in the notion of a faulty process and a non-faulty process. So, as we said, a process is faulty, if it takes a finite number of steps, in the sense it, it does something, something, something, and then abruptly stops or let us say stops for an infinite amount of time or for a very large amount of time, such that everybody else concludes that it is virtually dead.

So, we will see what exactly this means in the context of our proof. But essentially the idea is that for a finite number of times, it takes steps and then it basically stop. Whereas, a non-faulty process, every time you send it an event, it will take a step in a sense, receive it and then send additional messages or something. So, it will take a step the way that we have defined.

Furthermore, we define the notion of an admissible run. A run is basically a sequence of configurations where we just present messages to them. And then we move from configuration A to B, to C to D and so on. And so, a run is the same as let us say the run of a program where you start the protocol and the protocol runs. And the way that the protocol will run is that it will jump from one configuration to the next, to the next, as we are generating more events, the nodes are consuming them and changing their internal states would be moving from configuration to configuration and because messages are being sent and received. So, this is called a run.

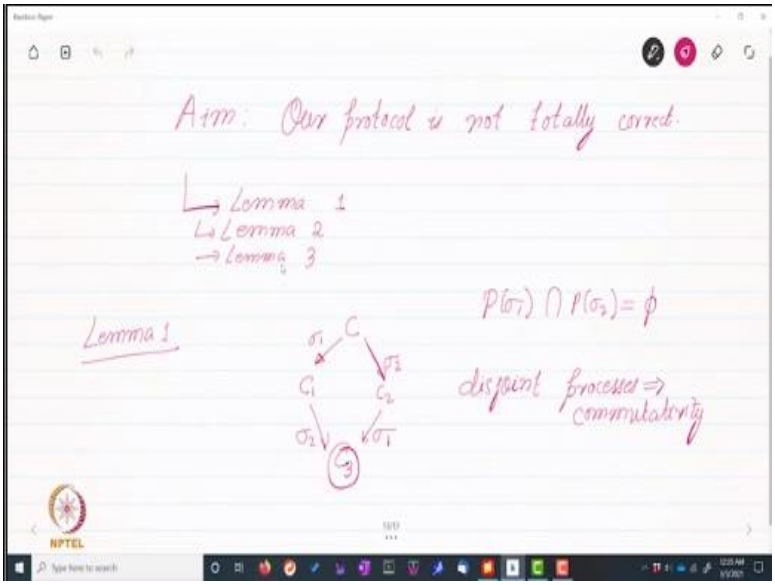
So, the admissible run is when basically all non-faulty nodes, get the message, get the message eventually. So, of course a message may be delayed, but eventually all non-faulty, nodes will get the message. So, for them, no message is dropped. So, this is an admissible run. So, what it totally correct protocol is, is that totally correct is essentially partially correct plus little bit more. Totally correct is basically partially correct. Partially correct means that it makes sense in the sense that consensus is achieved as we have seen over here.

And furthermore, we start with an open mind. So, we do not commit ourselves to one outcome, 0 or 1, and furthermore, every admissible. So, this is, let us say first clause in the second clause that every admissible run. So, admissible run is essentially a run of the protocol where all non-faulty nodes will get a message. So, for them, no messages dropped. So, every admissible run is a deciding run.

So, what this basically means is that if for every admissible run of the protocol, we finally reach a configuration that decides either decides 0 or 1, we do not really care, but finally, we do reach a configuration that decides the outcome, whatever it may be.

So, in the presence of one faulty process, this is what a totally correct execution to us means that the basic principles of consensus do hold, basic sanity checks hold, and furthermore, every admissible run is a deciding run in a sense that every time we run the protocol, we early ultimately end up with a decision. So, what we would like to prove is that our protocol is not totally correct in the sense every admissible run is not a deciding run, which means that even if you run it for an infinite number of steps, ultimately we will not be able to decide.

(Refer Slide Time: 34:10)



So, our aim, if you would look at it in this proof is basically to prove that our protocol is not totally correct, because if something is totally correct, then it means that it is deciding on every single run, it is totally decisive, but we want to prove the reverse that our protocol is indecisive. So, we want to say that our protocol is not totally correct. That is essentially what we would like to prove.

So, for this, we will prove a sequence of three lemmas, which will say that look, if you start from an indecisive state, regardless of how many messages you exchange, you will always have an in indecisive states in the sense your protocol. It is possible that for certain runs, regardless of how much you run you will never be able to make a decision. So, you will never be able to computer consensus, in the presence of one faulty process, which basically means that a protocol is not totally correct, because in this case, every admissible run is not a deciding run. So, ultimately, we are not deciding anything.

So, to proof this, we will essentially divide the proof into threes lemmas, lemma 1, lemma 2, and lemma 3. They can take a look at the paper. So, this is exactly what we are going to do, that we look at two simple lemmas and then a third one. So, the first lemma is like this, and it is fairly straightforward. So, here is what it says. So, what it says is let us start from some configuration C and apply to schedule the events? So, let us apply Sigma 1 or let us alternatively apply Sigma 2.

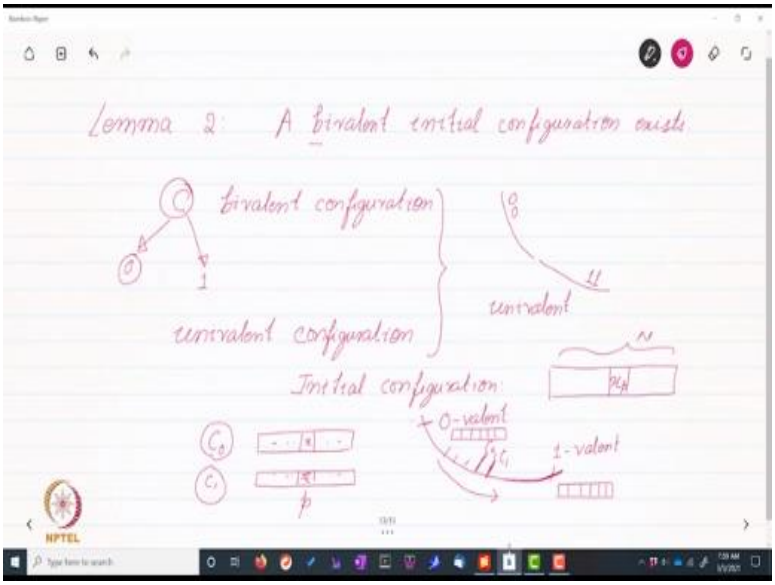
So, let these lead to configurations, C1 and C2, respectively, furthermore, all the processes that take steps in Sigma 1 and all the processes that take steps in Sigma 2, let them be mutually disjoined, or let us say the function P $(\sigma_1) \cap$ P $(\sigma_2) = \phi$. So, essentially what it means is Sigma

1 works on one set of processes and Sigma 2 works on a different set of processes. So, if that is the case, then what can be done is that we can apply Sigma 2 to C1 and we can apply Sigma 1 to C2. They will ultimately reach the same configuration C3.

Because as far as C3 is concerned, the relative order of Sigma 1 and Sigma 2 does not matter because the processes themselves are disjoined. So, it does not matter. There is no exchange of information between them. So, that is the reason here this appears to be commutative in the sense you either apply Sigma 1 first and then Sigma 2, or you apply Sigma 2 first and then Sigma 1. It does not matter. You ultimately reach the same point. So, this basically means that the processes are disjoined.

So, disjoint processes in a sense imply commutativity of operations. So, the operations commute, they can be replaced. So, we will remember this, this is a very important result and we will take it forward.

(Refer Slide Time: 37:58)



So, now let us come to the second lemma which is Lemma 2. So, prior to describing the lemma, let me describe, or let me initiate the definition of one more term. So, this is a simple term. So, let us say that C is a configuration. And let us say from this configuration, we can reach a configuration which decides 0. And we can reach a configuration that decides 1 in the sense that this is an open-minded configuration. So, we call this a bivalent configuration.

So, this means that starting from here, I can reach either a state that decides 0 or that decides 1. So, this is called bivalent. In comparison, we can define a univalent configuration where, regardless of the configuration transitions that you make, it is guaranteed that you will always

decide either 0 or 1 in the sense a decision has already been made and the decision is not going to change. So, that is a univalent configuration.

So, lemma 2 basically says that any protocol, a bivalent initial configuration exists, initial configuration exists. So, let us assume to the contrary that this is not the case. So, if this is not the case, what it means that from the definition of partial correctness, all the initial states are either 0 valent or 1 valent . So, if you do not have a bivalent state, then all so amusing state and configuration interchangeably.

So, it basically means that when you are just starting the protocol, what we said from partial correctness is that it should be possible in some runs to decide 0, in some runs to decide 1, in a sense, it should not be a fixed match that we always decide 0 or always decide 1. So, then of course there are two ways that this can be interpreted, one way is that our initial configurations itself are bivalent, in a sense, depending upon who sends what messages we can either proceed towards ultimately deciding 0 or deciding 1, that is one way of looking at it.

But let us say that in this case, this is what Lemma 2 is trying to prove that bivalent configuration exists, but less assume that it does not exist. Then another way of looking at partial correctness is that the initial configurations itself are either 0 valent or are 1 valent. Fair enough. So, if initial configurations itself are 0 valent and 1 valent. So, this will basically mean since we are assuming to the contrary and we want to prove by contradiction. This will essentially mean that from partial correctness, that all the initial configurations are univalent.

And furthermore, there has to be at least one such configuration, which decides 0 and at least one such configuration that decides 1 in the sense, all of them cannot be deciding 0 or cannot be deciding 1. So, given this let us look at what an initial configuration is. So, when we are starting the protocol, we are not sent any messages. There is no internal state, nothing. So, the only thing that we have in the initial configuration is basically the inputs. That is the only thing that we have.

So, if we have n processes, so, we can say that the initial configuration comprises an n input, array where the Pth entry is xp. So, if you recall, every process takes a one bit input and this can be the value that it will propose. So, basically if you have n processes for each process, we have a one-bit input and that is all. So, that is the only thing we have in our initial configuration.

So, now let us say that, let us look at a hypothetical line of all our configurations. So, let us say that at one end, we have a 0 valent configuration initial configuration. And on the other side,

we have a 1 valent. So, what does this basically mean? This basically means that there is a vector associated with this off the initial input values. And there is one more vector associated with this of the initial input values. The vectors are not the same. And because of this vector, the contents of this vector, we ultimately decide 0. And because of the contents of this vector, we ultimately decide 1.

So, let us do one thing. Let us approach this configuration from this by gradually flipping one, one bit each. So, then let us say the hamming distance is K. We need to flip K bits to reach this configuration from this 1. So, as we keep on flipping bit after bit after bit, ultimately, we will reach two configurations where we make the 0 to 1 transition.

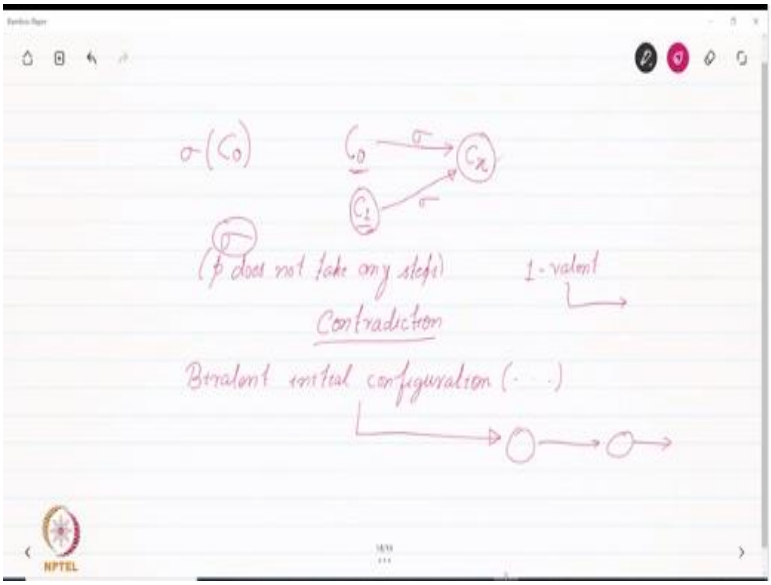
So, let us say that we reach a configuration C0 that is 0 valent. And we reach a configuration C1. That is 1 valent. And the only difference between C0 and C1 in terms of their input vectors will be a process P. And the rest of the bits will be the same. The only difference will be in a given location, let us say for process P.

So, let us say this has some value, and this has a compliment of that value, but the rest of the bits will exactly be the same. So, if this is the case, and there are two such configurations that are essentially one bit apart and basically for process P the inputs are different. So, let us use this as a starting point.

So, what do we have? So, let me again, retrace our journey. So, the lemma 2, we had to prove that a bivalent initial configuration exists. We assume that to the contrary, let us assume that it does not exist. If it does not exist, then by partial correctness, which holds all our initial configurations are now univalent, but by partial correctness, some have to decide 0 and some have to decide 1.

Furthermore, what we said is that in the initial state, we just have 1 bit per process. So, the initial configuration is just an n bit vector. And as we move from one configuration that is 0 valent to a configuration, that is 1 valent will always arrive at a C0, C1 pair, which are essentially neighbours that differ by just one bit for the Pth process where C0 is 0 valent and C1 is 1 valent. So, given the fact that we have come to this point, here is what we can do.

(Refer Slide Time: 46:05)



So, let us start at C0 and consider an admissible run. And so, let the schedule of events be Sigma C0. So, which means that from C0, if I apply the set of event Sigma I will reach some state. Let us say, Cx. Similarly, what I can do is I can take C1 and apply the exact same set of events, but there is something special about this set of events in this set of events. Process P does not take any steps the way that we have defined. So, in a sense, process P is quiet.

So, as we had discussed a faulty process, when the time comes, it betrays you. So, it basically does not take any steps. So, let us say that process P betrays us, and it does not take any steps. See, even though it is inputs are different, since it does not take any steps, the fact that his inputs are different in both the initial configuration C0 and C1, it is not visible to other processes.

So, now when we apply a set of transitions, which are essentially a set of events are applied, but in this set of events, P is not taking any steps. So, regardless of where we start from C0 or C1, we are bound to reach the same finance configuration, which is Cx primarily because process P was only different. And in this case, it did not take any steps.

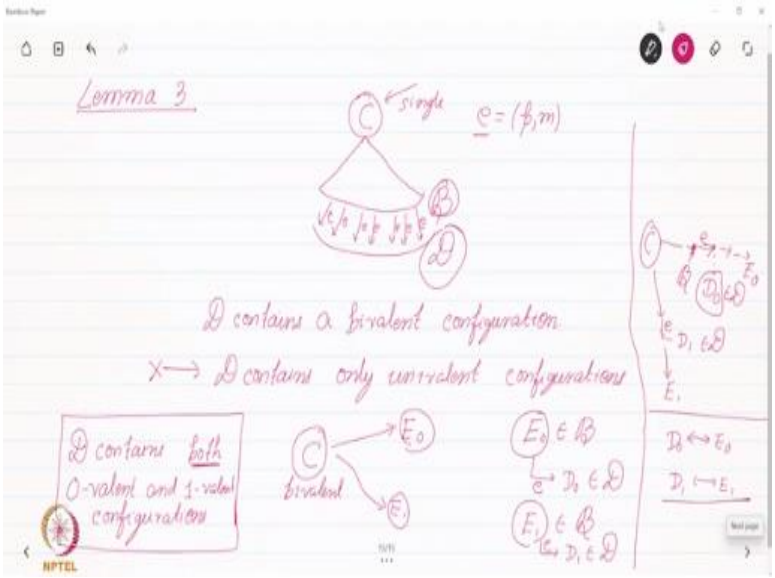
So, given that we are reaching the same configuration, there is a problem. This was 0 valent. This was 1 valent. If this configuration is 0 valent, then this state cannot be 1 valent. Because if this configuration deciding 0, then this configuration could not be deciding 1 because once a state is, let us say 1 valent, regardless of the transitions that are made, so regardless of whatever transitions we make, all reachable configurations will be 1 valent, but that is not happening over here.

So, which basically means that for C0 and C1, which decide different values, there can never be a common configuration that is reachable from both. But given the fact that we have a common configuration that is reachable from both, it essentially indicates that there is a contradiction because this is not possible. So, if this decide 0, this one has to decide 0, but then C1 cannot point to it because C1 would want this to decide to 1. So, there is clearly a contradiction and the contradiction tells us that we will at least have 1 initial state, which is bivalent fair enough, lemma, prove we are happy.

So, now what we will prove is that we will essentially provide an inductive argument. So, we have a bivalent initial configuration. So, bivalent basically means that it is open-minded in the sense it has not made up its mind, whether it is going to decide 0 or 1. So, then it is an open mind configuration it is undecided. So, we will now prove that given an initial state is undecided.

It is possible that after we apply a set of events, we will still arrive at an undecided state. Again, we apply a set of events, we will still arrive at an undecided state, so on and so forth. We can continue till infinity will always have a state, which is undecided, which means that our protocol will never be able to make a decision 0 or 1, which is exactly what we set out to prove. Which means that it is not totally correct. This means that every admissible run is not deciding or in common man terms. If we want to solve a problem, it is possible that we might run an infinite number of steps and still not be able to solve it.

(Refer Slide Time: 50:30)



So, what will help us in doing this is Lemma 3. So, what does Lemma 3 say? So, lemma 3 says let us C be a bivalent configuration. Furthermore, let us consider an event of the type p comma

m which means there an event destined for process p with message. So, let this be applicable to C.

Then let the set B all the states, all the configurations that are reachable from C without applying e. so, essentially what I do is I apply all the other events, but I do not apply e. so let B be all the configurations that are reachable from C without applying E and furthermore, let the set D be all the configurations from B, where all that I do is I take a configuration here and I just apply e.

So, given the fact that e was applicable to C, which is the initial configuration. So, then we did not apply it. So, we apply the rest of the event. So, that is okay. So, given that it was originally applicable, and of course, let us assume that the message got delayed, which means the event D got delayed, it will still be applicable. So, let us say after that, we reach a bunch of states B, and we can apply e to any one of them. So, we will reach a set of configurations D.

So, mind you, C was a single configuration, which is bivalent. But B and D are sets. So, B is the set of all configurations that are reachable without applying e and then I apply e and then apply e to every single configuration of B. And then I reach the set D. So, what we, I want to prove is the statement of the lemma is that D contains a bivalent configuration. That is what I would like to prove. Now D contains a bivalent configuration.

So, again, let us assume to the contrary that this is not the case. See if this is not the case, what is the proof by contradiction assumption that D contains only univalent configurations. So, this is what we are setting out to disprove. So, let us say that, given that C is bivalent so, we will start from this point. So, since C is bivalent, let us look at two configurations, E0 and E1, where E0 is univalent it decides 0 and E1 is univalent. It decides 1.

So, let us now look at the different cases that are possible. And of course, E0 and E1 will exist because C is bivalent. So, then E0 and E1 will exist because it would ultimately take you towards two states, depending upon the transitions that are applied, which are E0 and E1. So, let us consider, let us say the case of E0, if E0 is an element of the set B, it means that E has not been applied to it. So, let us do one thing, let us apply E to it. So, then we will reach, let us say a state called D0, which is an element of set D and this follows from definition.

So, same holds for, let us say the set E1, E1 is element of set B in the sense it is reachable without applying E, then all that I do is I apply E to it, and then I will reach another state D1,

which is an element of D. So, now let us consider the other case, which I am taking out some real estate over here, and I will write it over here.

So, let us assume that we reach the state E0 after applying E. so, which means from C, we come here, then we reach some configuration. We apply E so whatever is this configuration is now a member is now a part of D. So, let us call this configuration as D 0 and there, after a set of transitions, we reach E0.

So, basically this state over here has to be an element of the set D. So, this follows by definition because this state, this state is reachable from C. So, this state is an element of set B, and then we apply the event D so that by definition, this becomes an element of set D and given the fact that this state is not bivalent, so this follows from our assumption, and this ultimately leads to E0. So, this state has to be D0, which is univalent. So, using a similar argument, you can also say that, look, if we apply E before reaching E1, then this state over here is D1, which is an element of D.

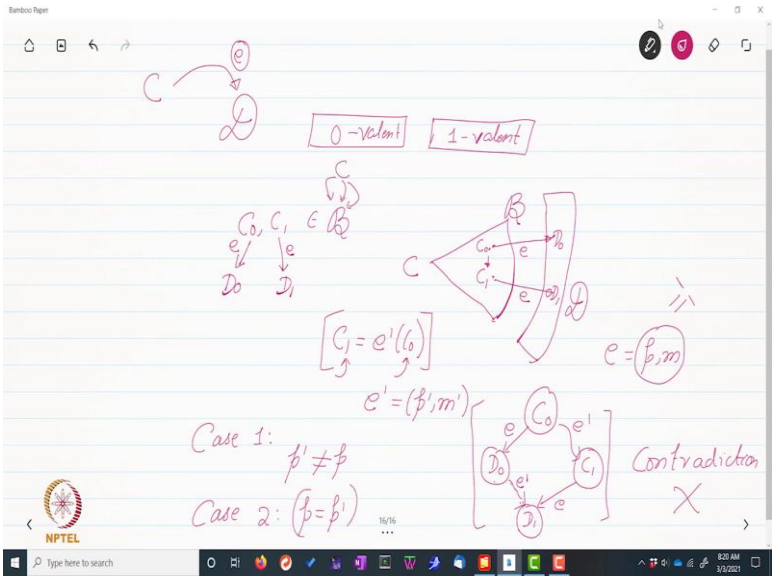
See, if I look at both these cases, what will they essentially tell us? What they will essentially tell us is that either there is a path from D0 to E0, it does not matter. Or there is a path from E0 to D0. So, one of these is true. Similarly, the path from D1 to E1, or from E1 to D1, which is fine. But then what is the key conclusion? Why did we do all of this? So again, this is the last piece of real estate that I will use on this slide.

The reason that we did all of this is to say is to prove something very simple. What we proved is that D contains both 0 valent and 1 valent. That is what we were able to prove that the set D it contains both 0 valent as well as 1 valent configurations. And how are we able to prove that?

So let me again, go back. So, we set that look, state C is bivalent. It this means that two states E0 and E1 would be reachable from C, but E0 decides 0 and E1 decides 1. So, from our definition, let us we took two cases. So, let us assume that to each E0, we did not apply E then there is a state D0 and D and a state D1 and D per D0, decides 0 and D1 decides 1. And then we looked at the other case, and let us say, we apply D to come to E0. Then also, there are two states in D where one state decides is 0 valent. And one state is 1 valent.

So, this means that D contains both 0 valent and 1 valent configurations. It is not the case, even though D we are assuming, does not contain the bivalent configuration. It is not the case that D contains only 0 valent configurations, and only 1 valent configurations. It contains both. This is what we are able to prove.

(Refer Slide Time: 59:32)



Now, what we are going to do is we will play the same trick that we played in lemma 2, where we are saying that, look, we have the set D. So, from C, we have come to the set D by applying event e, basically, and this contains two kinds of configurations, one kind at 0 valent and another kind 1 valent.

So, this is where we stand. So, now let us do one thing. Let us consider two configurations, C0, and C1, both an element of the set B. So, recall that what is a set B? It is a set of all the configurations that are reachable from C without applying event D. Furthermore, it is possible to prove the same way as we did in lemma 2, that C0 and C1 are essentially one step away. And so they are essentially one step away. And, and furthermore, the step D0 comes from applying e to C0, and the step D1 comes by applying e to C1.

So, let me rephrase this. We are saying a lot of things. This is complicated. So, I will say this again. So, what was the set B? The set B was essentially a set of all the configurations that are reachable from set from configuration C without applying event D.

No problem. Then when I apply event D, I come to a much bigger set. So, let us say this is the set. And this is set D and set D basically every single point within set D is generated by taking a point from within set B and applying event D. And what we were just able to prove that set D contains both 0 valent and 1 valent configurations. And by our assumption, we are saying it does not contain a bivalent configuration.

So, now what I am saying is that there will definitely be 2 configurations, C0, and C1. So, I start, they lead to D0 and D1 after applying event D, after applying event D where D0 decides

0 and D1 decides 1. So, that has to be the case because given the fact that you have D0 and D1, they will have images and set B, and these would have been created by applying event D so this follows by definition.

So, there is nothing non obvious over here. But if you look at all the states in set B, you will definitely arrive at one such pair where this relationship holds that C1 = e'(C0). So, this is similar to lemma 2, where we said that, look, if we consider all the configurations, that definitely one point will arrive. The two configurations are essentially one step away. One decides 0 one decides 1.

In this case, of course, C0 and C1. It is not the case that C0 decides 0, but C0 after applying an event goes to D0, which decides 0 and C1 decides, and D1 decides 1. But the point is that by lemma 2, well, let us say by a reasoning, similar to lemma 2 we can argue that, look, we have a large number of events, and all of them are essentially reachable from C via a large number of paths.

So, then if I look at it, so let us see if I, if let us say this is C1, and I just look at it, I go back, back, back all the way up to C. So, there will definitely be 1 event where there is a boundary in the sense that it is one step away where it is images in D, one decides 0 and one of course decides 1 because all the states in D are univalent and there will be at least one such event pay, which are 1 step away, where C0 would lead to D0 and C1 would lead to D1.

So, let us assume that this is what they are called. And the single step is essentially the event e' = (p', m'). So, given that we have been able to prove this, we will look now, look at two cases and look at and see what happens. So, recall that we are looking at two events over here. We are looking at event e and event e dash. So, event e applies to process p and e' applies to the transition that e' applies to process p', which leads to the transition from C0 to C1.

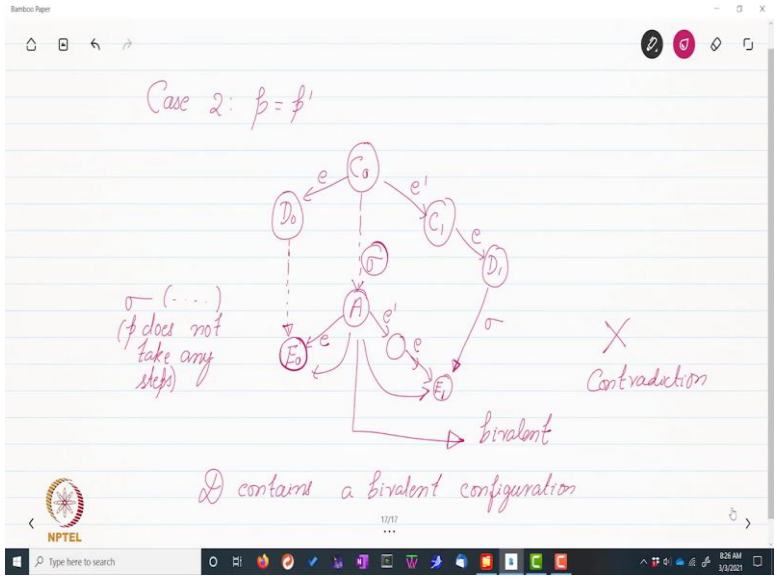
So, the case 1, which is kind of simple is that p' ≠ p. So, if p' ≠ p. So, then what does this mean? So, what this means is that we should draw another simple diagram and a simple diagram would look like this, that we had states C0 from there, we apply e'. So, we come to configuration C1. Then what we know is that from C0, we apply event e. So, we come to D0. And similarly, from here, we apply event e and we come to D1.

So now here is the fun part, given that e and e', they are disjoined. So, e is working on process p and e' is working on process p'. So, by lemma 1 commutative would hold because the process sets are disjoined, which basically means that if I apply e' to D0, then I should be able to come

to D1. And here in lies, a contradiction, because any transition from a 0 valance state will never take us to a 1 valance state, but it, this appears to be happening given the fact that this appears to be happening. This is a contradiction and this is not possible.

So, we clearly have a contradiction over here, and this is definitely not possible. Say if this is not possible, what we shall do is we shall consider the other case in which is case 2, where p = p'.

(Refer Slide Time: 66:44)



So, the other case where p = p', let me write it over here. So, again, let us draw a few diagrams. So, we have C0. So, we apply e' to it. We come to C1. Then we apply e to it. So, we come to D1. Similarly, what I know is that I apply e to C0, and I come to D0. So, this is the information that I have. And furthermore, I know that both e and e' are actually working on the same process.

So, let us do one thing. So, this is where, again, the notion of our faulty process will come in. So, let us consider a run, a run Sigma where p does not take any steps. So, as I said, the faulty process, we trace us at the right time. So, let us assume that p does not take any steps.

And so, let us consider one such run. So, in this run, given that p does not take any runs. So, let this be a deciding run, because the point is that we do not know for how long p is not going to take steps. And since we have assumed that our system is totally correct. So, basically for every state, we should have a, we should have an admissible and deciding run. So, let us assume that we have one such deciding run that P is not taking any steps.

So, P is the faulty process in this case, and it has betrayed us. So, in this case, we will apply the transition sigma where p is not taking any steps. And so, let us say that we arrive at state A. So, given the fact that we arrive at state A. So, now things appear to be possibly under our control.

So, let us now apply event E to state A. So, then if we do that, we will reach some state, but here is the fun part e and Sigma are mutually disjoint. So, it basically means in e only process p takes steps. And in Sigma process, p does not take any steps. So, then mutually is joined. So, nothing stops us from applying Sigma to D0 as well. And if we do that, given that D0 is univalent, the state here will also be univalent.

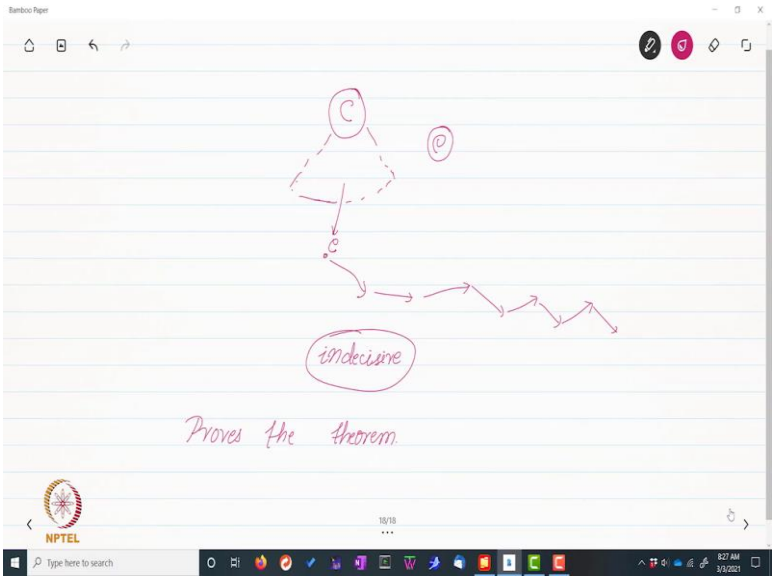
And so, basically this state will be E0. So, no problem. Let us do the same thing on the other side. So, let us first apply e' And then let us apply e. So, if I apply e' and e in quick succession, then I will arrive at one state and this state, let us call it E1.

The reason is like this, that e' and e are again on the same process p but process p is not there in Sigma. So, nothing stops us from applying Sigma to D1. And with that, we arrive at state E1 and E1 has to be 1 valent because D1 is 1 valent and they can see by lemma 1, this construction is allowed. So, just take a look at this once again, what we have done is that Sigma, as far as we are concerned is an admissible decidable run. And this exists because we have assumed that our protocol is totally correct. And from every configuration and admissible deciding run exists.

So, then the thing is that if that is the case, then we can do this construction. And what we see is that from state A, we can reach a 0-valance state and we can reach a 1 valance state. This means that state A is bivalent and. So, this means that state A is bivalent, but this is not possible since the run to A is deciding. So, since Sigma is a deciding run, A cannot be a bivalent state. Consequently, we have a contradiction over here because this goes against our assumptions.

So, given the fact that this goes against our assumptions, lemma 3, as we have written, that stands proven where we said that D also contains a bivalent configuration. Because we looked at two cases and the two cases are over here, one case is over here and one case is over here in both the cases we arrived at a contradiction. So, this essentially means that as for lemma 3, D contains a bivalent configuration. So, now we are almost done. There is nothing much that we need to do other than spend some amount of time on the next slide.

(Refer Slide Time: 72:03)

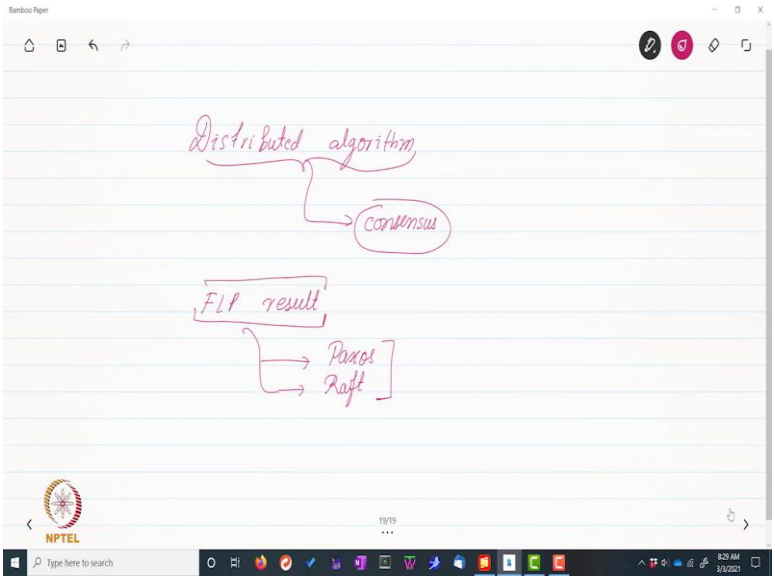


So, here, what did we do? We, as we said that, look at the beginning, you will have a bivalent configuration. So, think of this as the base case of the induction. Then we said that, let us say if we consider an event D, which is applied to process p. So, what you do is that you drain the rest of the events that you have, you will reach a large number of states. From here then again, you apply e. so again, so it is possible to reach a bivalent configuration.

And you just keep on doing that. You keep on applying events. You will keep on reaching bivalent configurations, the way that we have shown. So, it is possible that you keep on applying events and keep doing so infinitely, and you will still be reaching bivalent configurations, in the sense, you will still not be able to make a decision.

So, this means that your consensus will not happen because your protocol is indecisive. And this basically means that you will simply not arrive at a state at a configuration where a decision has been made. So, this pretty much proves the theorem. So, what is the implication of the theorem? The implication of the theorem basically means that whenever I want to do any kind of a solve any kind of a distributed algorithm.

(Refer Slide Time: 73:26)



So, it turns out that most distributed algorithms can be mapped to some or other kind, a consensus problem, some or other kind, an agreement problem. So, it might not be able to efficiently map, but it can be mapped to a consensus problem. And given the fact that this consensus problem cannot be solved.

Even with one faulty process, this means that if you consider faults in our system, our system will be severely challenged. So, it is simply not possible to arrive at any form of agreement. Even if one single process is faulty and this to us represents a significant issue. So, we will keep this in mind. We will keep referring to this as the FLP result. So, we will keep this in mind and subsequently, we will discuss a host of consensus protocols.

So, we will discuss back source in later slides of the lecture set. We will discuss wrapped in all of them. We will say that look, yes, we know this result. It basically means that we will not be able to achieve consensus all the time, but at least let us aim to do this most of the time, that most of the time, let us try to achieve some degree of consensus and otherwise let us rely on some form of time out mechanisms and some form of timer because in the real world, nothing is really indefinite. I mean, it can be 1 second to 1 hour to 1 day, but with some additional timing mechanism we can do better.

So, nothing is theoretically a synchronous. So, but of course the FLP result kinds of gives us a hard, lower bound of what is doable or rather what is not doable.