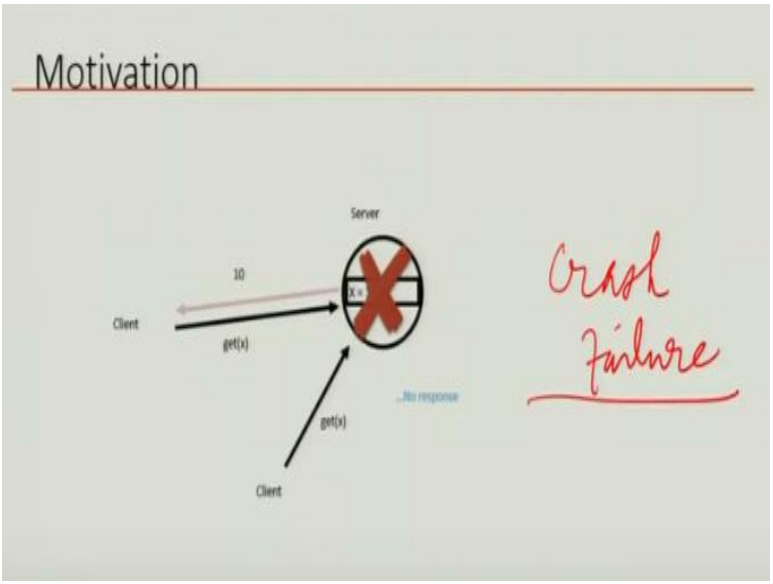
Introduction to Blockchain Technology & Applications
Prof. Sandeep Shukla
Department of Computer Science and Engineering
Indian Institute of Technology-Kanpur

Lecture - 23

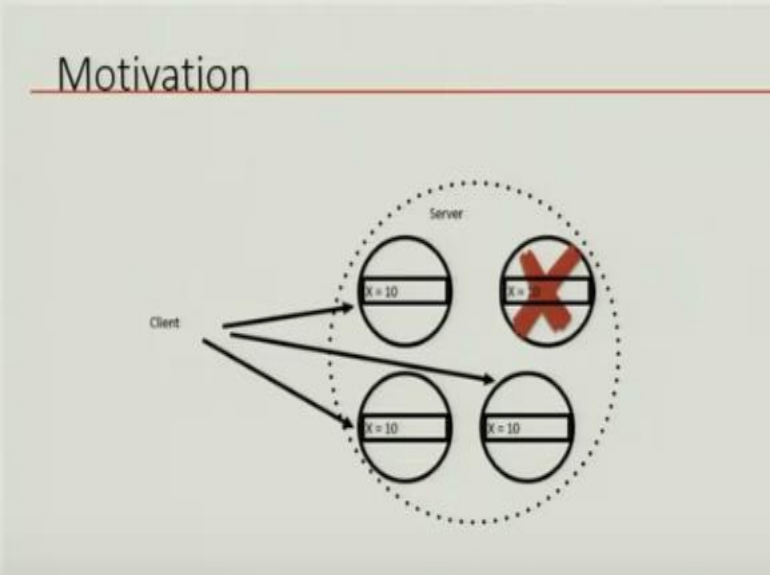
Welcome back. So we were talking about state machine replication. And we wanted to show you some animation to make you understand about state machine replication. So let us say I have a server.

(Refer Slide Time: 00:29)



And the server maintains some data or state. And let us say currently the value of some variable x is 10 in the server. Now suppose a client says what is the value of x now right? So you will tell them that the value is 10. Now let us say the another client says that change the value but at this point, the server crashes. So at this point, if it says give me the value of x, then you will not get any response. So this is what is called a crash failure.

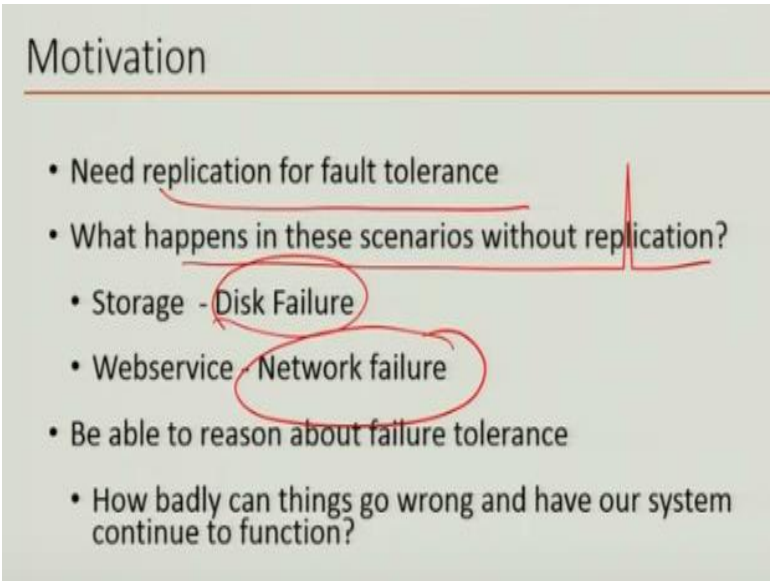
(Refer Slide Time: 01:22)



Now the question now is that in order to avoid such a no response scenario, what people started doing is to have replication of state machine. So instead of one I will have three other which are exactly replicating every state transition of the original one, okay. So you would think that would solve the problem. But it will bring other problems. So let us see what kind of problems it may have.

So let us say one of the four has crashed and three are alive. And then the client asks something. Client does not know how many there are right? So client is asking this is for to the client, this is a service, right. Now whether the service is implemented as a replicated service or as a single service, the client does not care. So the client is asking and it will then be forwarded to all the replicas. In this case, one of the replica is dead. So it cannot be sent to that.

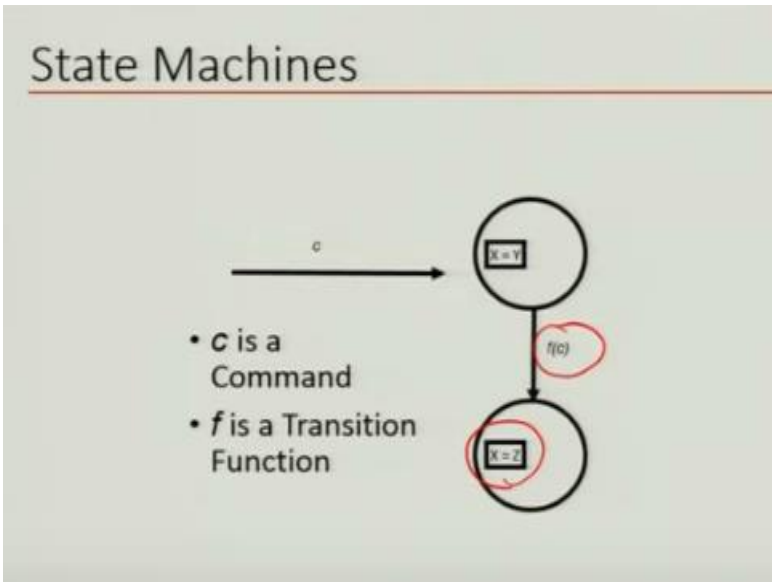
(Refer Slide Time: 02:28)



So this is what is called a replication for fault tolerance. So what kind of you know if you do not have replication, what could have happened? So first of all, you can have disk failure and in case of service is over the network, then you have may have network failure. So you have to be able to reason about failure tolerance, which means that if one of the replica fails or more replica fails when can I guarantee that the service will still work correctly as far as the client is concerned?

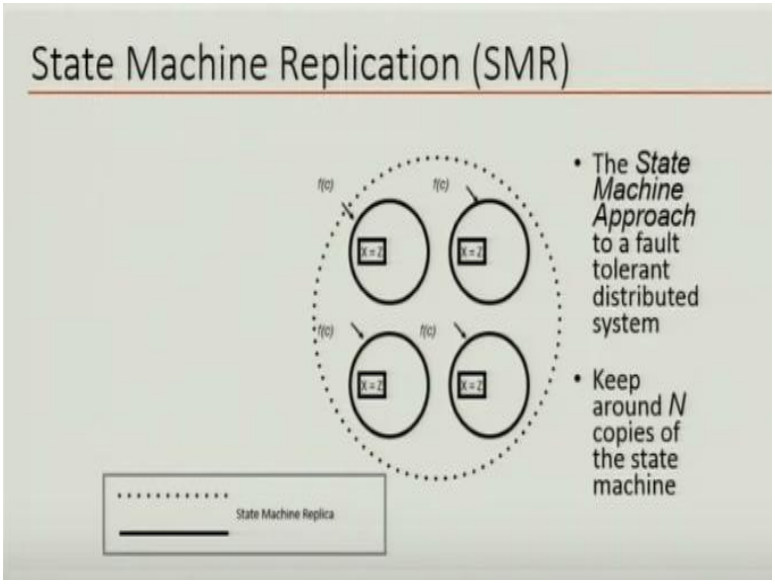
And we will see what can go wrong? And when can we not guarantee that the client will be it will be totally transparent to the client, whether something has failed or not, how many failed etc., can we guarantee that?

(Refer Slide Time: 03:29)



So let us say I have a state machine and I have a client. And client send some command, right? And that command will lead to a state change, right? For example, if this command is basically saying, set the value of x to a new value z, then a state change will occur because now the value of x has now changed from y to z. So it is a new state. So f is the transition function, it does not matter what you call it.

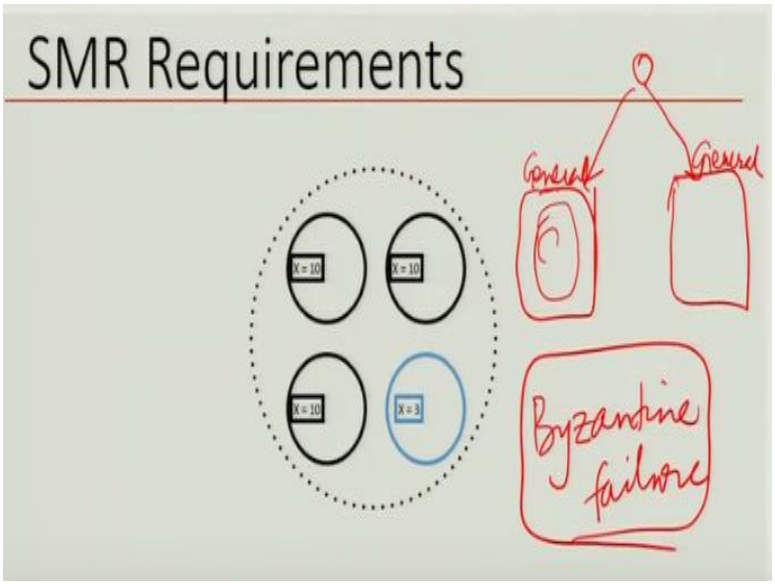
(Refer Slide Time: 04:06)



So now suppose we have this situation. For earlier picture, we saw that we have only one server, and it is changing state. So this is a state machine. Now let us say you have four state machines, and they are replicating each other, which means that they are always having the same, they must be having always the same transition unless one of them or more of them have failed.

Obviously, then they cannot do any transition. So let us say I have four copies. Now let us say I send the command. So it goes to all of them as a state change activity.

(Refer Slide Time: 04:49)



And the state of all of them will change. So now let us look at this scenario where x is three in each of the replica. So they are in the same state. Now you send a command, which says that change the value of x to 10. So if it is replicating state machines, then everybody should change it to 10. So under normal circumstances when nothing fails, etc., that is what you expect. So this is good.

Now let us consider another scenario, when the same situation happens, you want to change x to 10. And that happens. So that is good. Now you want to say x equals 10. And at this time, something goes wrong in the one of the servers, and it either is unable to change the state or it becomes malicious. And that kind of malicious behavior when it acts on its own will, rather than acting as per instruction then we say that, that machine is showing what is called a Byzantine failure.

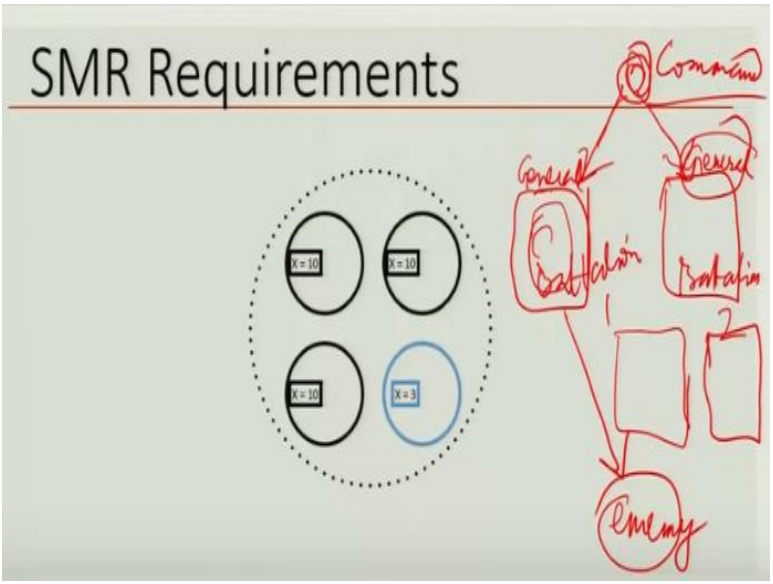
So this is Byzantine failure actually encapsulates many different types of behavior. For example, pretending to be dead could be a Byzantine behavior. Actually dying would could be interpreted by as a Byzantine behavior. Or behaving differently then how it is programmed, see all these state machines have been programmed to follow each other.

But, if one of the machine decides not to follow, then it will be also Byzantine behavior. And then when it does not follow what it will do, we do not know. It can be do arbitrary random things, right. So why is it called a Byzantine behavior is a long

story. So there is this thought experiment. In this thought experiment, you have a commander and you have two sets of army.

And the commander is supposed to send command. And then these set of this, the general here and general here must are supposed to obey the order. For example, if the commander says, both should attack.

(Refer Slide Time: 07:36)



So let us say I have a situation where the enemy is here. And you have battalion 1and battalion 2. And commander is going to decide strategic based on his strategy, his knowledge of enemy position and his knowledge of whether one battalion is enough to you know win over the army or both the battalion is needed and accordingly he will give command.

So let us say the commander decides that both the battalion should go and attack otherwise the one battalion if it goes it will be decimated. So if the commanders sends messenger to both saying that attack. Now if this general is unfaithful, knowing that the order is attack, he might say, I got the order to retreat. So this general will not know that so he will attack and get decimated.

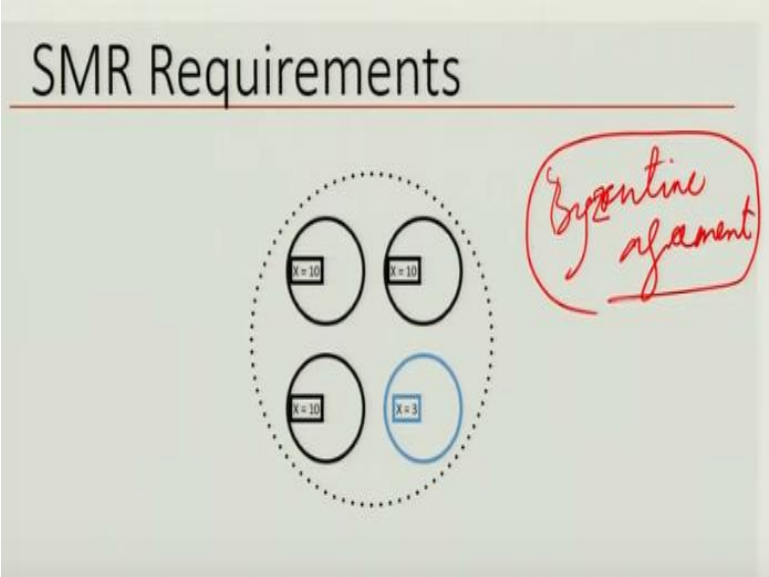
So the question is that how can the commander ensure given that there is a possibility that one of his general would actually not behave properly. How can he assure that there will be enough battalion to actually win, right? So he cannot do this even with

three generals. In fact, if there is at least if he assumes that only one general will defect, even then he will need four battalions and four generals.

Out of which only one can be Byzantinely behavior in a Byzantine way, then only he can ensure that he can win, right. So that thought experiment was done by distributed algorithms people back in 70s to actually formulate this notion of fault tolerance. And they, talked about crash faults, they talked about stop fault. But also there are other faults like message corruption fault and etc., which is a part of the network communication.

But then to encapsulate a behavior that is not crash, but that is downright you know malicious they considered this notion of Byzantine generals, this story of Byzantine generals to formulate this problem. So that is what we mean by Byzantine faults or Byzantine behavior. That is malicious behavior by one of the entities or more entities that we do not know.

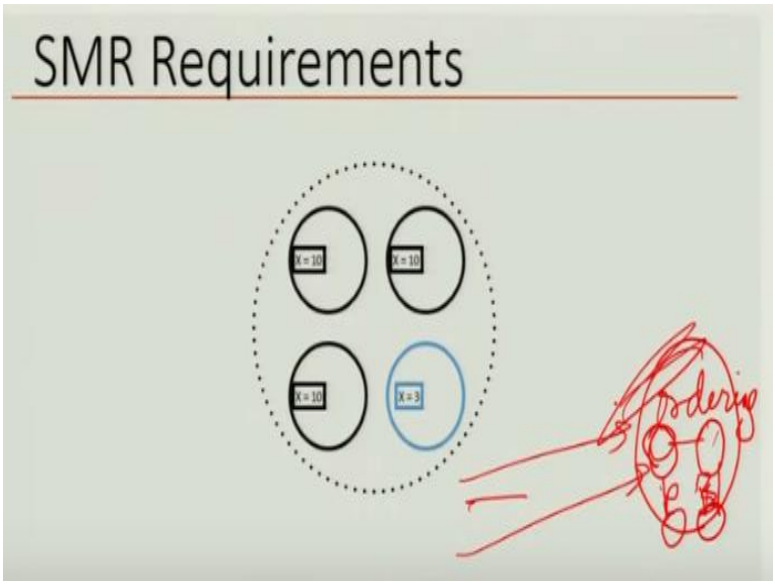
(Refer Slide Time: 10:34)



But let us say I am designing this system, then I have to make some assumption. If all my generals or all my replicas go Byzantine, then I have no way to serve the customer. So I have to question that, at most how many generals or how many replicas can go Byzantine, and I can still guarantee service correct service to the customer. And that is the problem of Byzantine consensus or Byzantine agreement problem.

So here we have a Byzantine agreement problem. Now in today's world, this was done before all this notion of malware and everything was not there. But in today's world, if a malware infects one of the replicas then it can behave Byzantine, right. So this is a very realistic scenario in today's world that some machines may be infected by malware or somebody might do a privilege escalation and takeovers that machine and then not let it do the right thing. So this is quite possible with cyber-attack scenario.

(Refer Slide Time: 11:48)



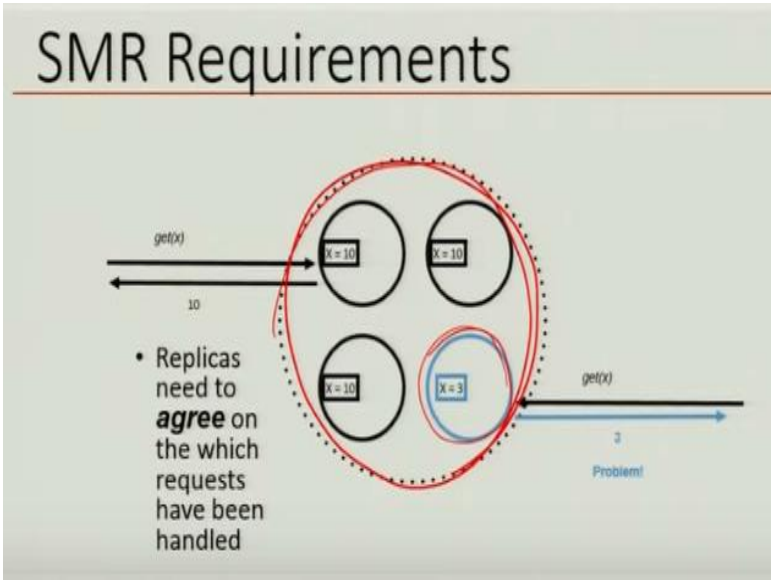
So in case of Hyperledger for example, let us say the transactions are sent to by different clients to this ordering service. So you expect that the ordinary will be done correctly, right. Now this ordering service, let us say you want to make it crash tolerant. To make it crash tolerant you need to do you need to have at least one replica right. So this is primary and backup.

So when a crash happens, you have to assume that only one we crash, both will not crash. Because if both crashes there will be no ordering, the whole thing will halt. But if you assume one crash, then you can do this. But if you assume Byzantine behavior, then as we will discuss, you will need at least four of them in order and that also by assuming only one of them will go malicious.

If you assume the two of them will go malicious, then you will need at least six replicas. So depending on what you are going to assume and how strongly you feel about importance of the service continuation is despite some of the some of them going rogue will determine how much replica you are going to do in the ordering

service. So that is the idea that we are now trying to just give illustrate to you through some examples.

(Refer Slide Time: 13:11)



You may understand it very superficially because there could be an entire class entire course on Byzantine agreement and Byzantine consensus, and so on. So it is not that you know by listening to lecture for an hour or so you will completely master the idea of Byzantine failure, but at least for this class, you should understand why you know ordering service needs to have certain level of fault tolerance.

It should have crash fault tolerance. That is when if the machine that is doing ordering crashes, then at least somebody can take over that is one thing. But if you can assume that if the designer of the entire infrastructure of the blockchain besides that, there is also a possibility of malware attack or cyber-attack, then you might want to assume a Byzantine failure model. So this kind of thing you need to understand.

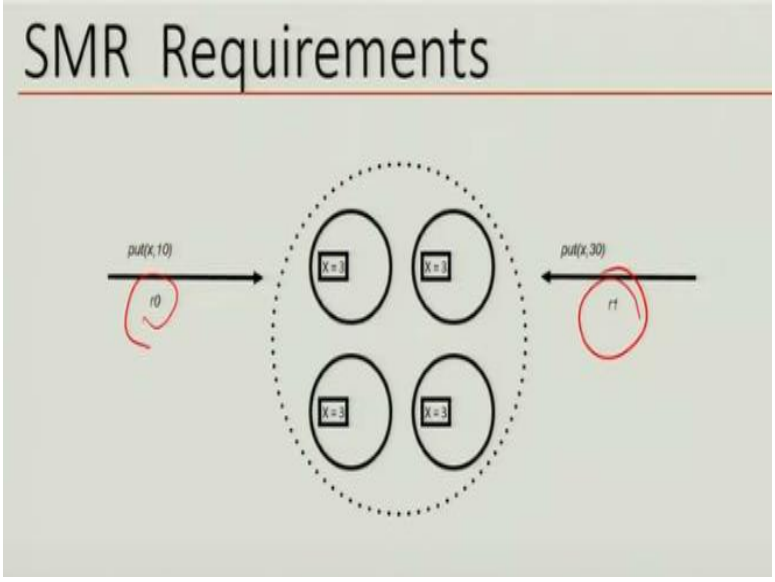
So here in this model, there are four replicas and then you are seeing that one of them is misbehaving. So now a client says give me value of x after the previous transaction where x value was changed. So you will get the right value. Now what is happening, one thing you need to understand is that when a client submits his request, he does not see these four things separately, right. He asks it to the ordering service.

So the ordering service or whatever service it is, will may actually first send this request to whichever server is closer to the client in terms of network sense. So maybe

another client who is there who request, his request goes to this replica. Then this replica will give a wrong value, right? And that is where this Byzantine failure will fail the system.

So the point is that all the replicas need to always agree on the state and all requests must be fulfilled in the same way irrespective of who is requesting that service, when the system is in a certain state.

(Refer Slide Time: 15:39)



So now the other issue is that when you have multiple requests, how that is handled, because earlier we were talking about one is behaving improperly and therefore if the information and the request is going to that particular replica will get this different results, etc. But now we are talking about when multiple different requests are coming.

Now when multiple requests are coming on the same piece of data, multiple transactions on the same piece of data. For example, here we have one transaction from one client, who is saying make x equals 10. The other guy, another client says make x equals 30, right. Now in this case, somebody has to decide in what order these transactions should be executed, because if the order is this r 0 first and r 1 second, then x will first change to 10 and then it will change to 30.

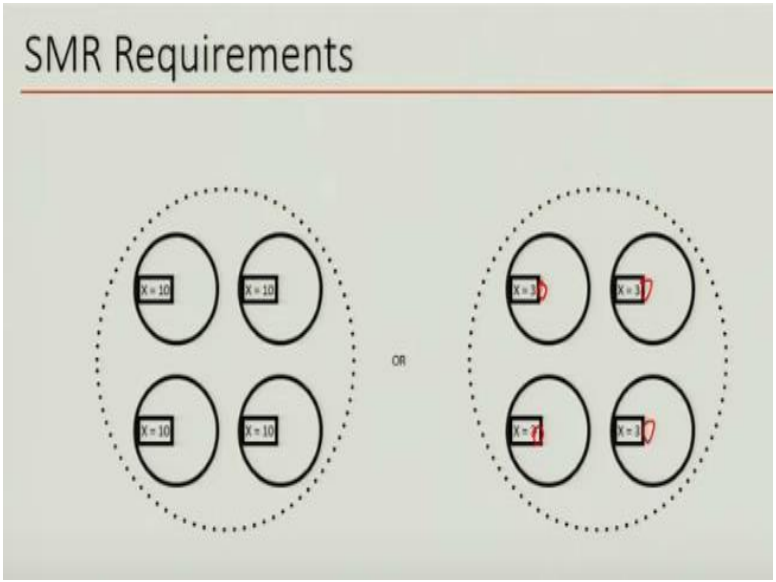
If you if the r 1 is goes first, then x will first change to 30 and then to 10. So after these two transactions, if somebody asks the value of x, what the answer will be will

depend on what would be the what is the in what order the transactions were done. What that also means that all these replicas should do the transactions in the same order. Now as I was saying that when these requests come, they may go to the one that is in closer to that client in the network sense.

So this may go to this guy on the left, and this may go to this guy on the right, right? So if this guy decides that I will do r 1 first and this guy decides I will do r 0 first, and then this guy does r 1 and this guy does r 0 later, then this guy will end up in 30. And this guy will end up in 10, which is no good, because all the replicas should have an agreement on what their state is all the time.

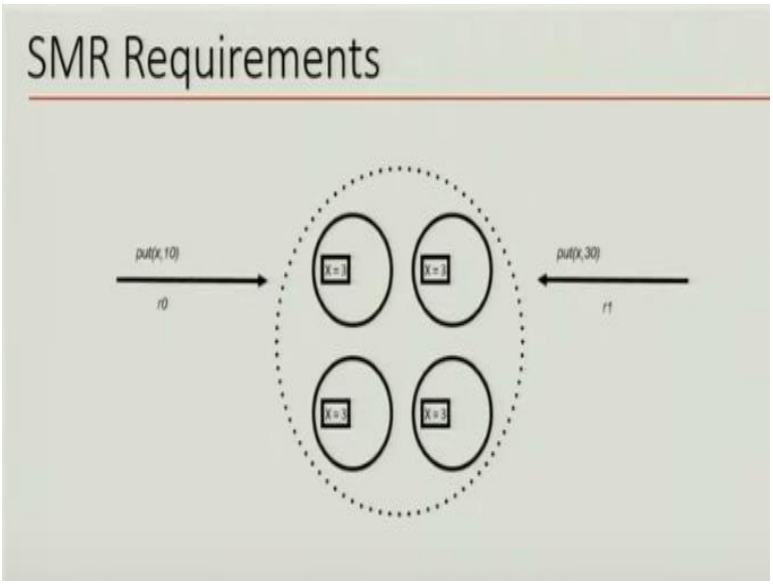
So therefore, we have to make sure that the transactions are ordered correctly.

(Refer Slide Time: 17:54)

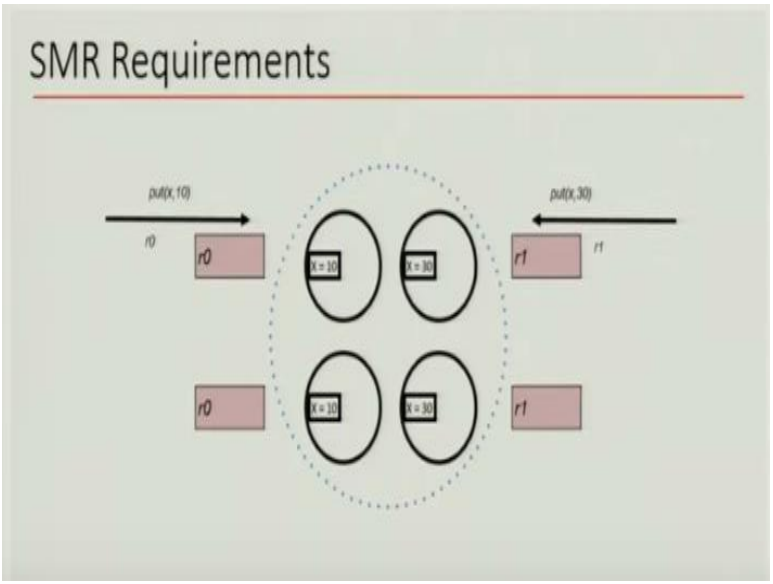


So the result would be should be one of these. Actually, this is this should be 30. But the result should be one of this, then we will say that it is correct. But if result is like half of them have 10 and half of them has 30. That is not correct. So that is what we want to guarantee that it is either after those two transactions, it is either all 10 or all 30.

(Refer Slide Time: 18:22)

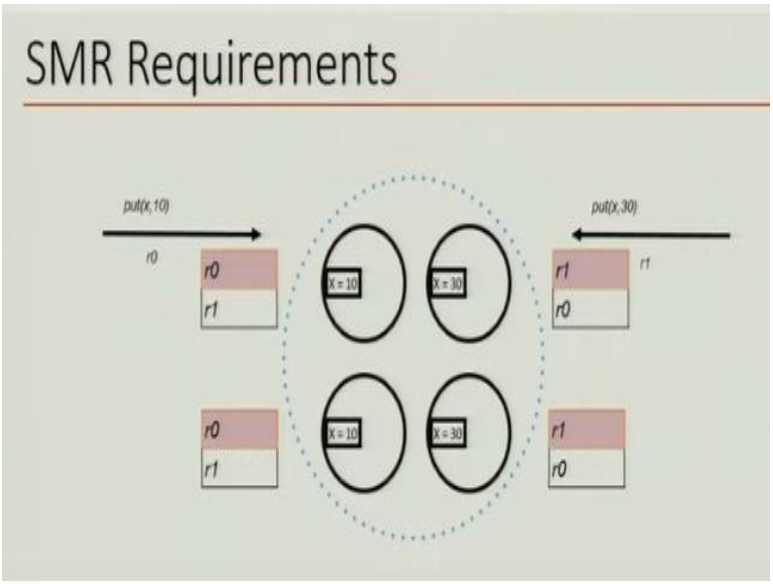


(Refer Slide Time: 18:24)



So therefore, we have to decide how to order the transactions. So there are many algorithms. So we will now see how this is done.

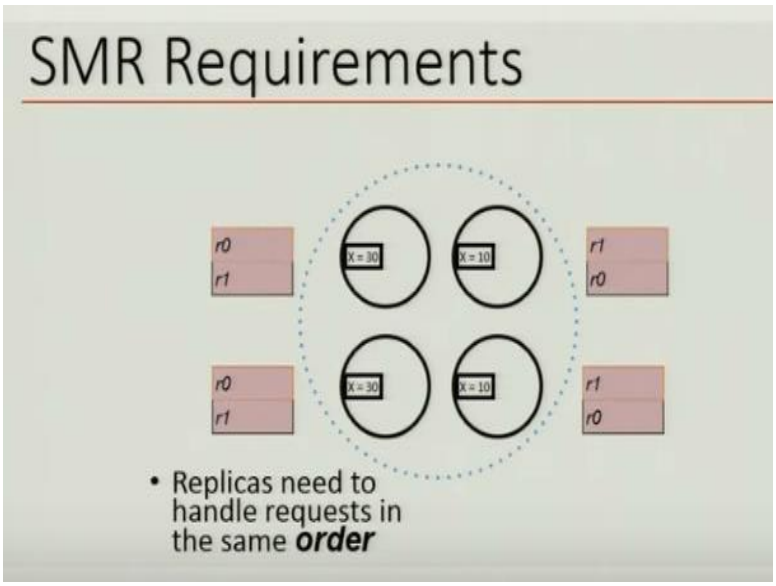
(Refer Slide Time: 18:35)



So everybody, every replica, will get the list of transactions that are under processing right now. But they got them in the different order. For example, this guy has got r 0 first, this guy has got r 1 first, so they will put them here. Now then, everybody has to decide to either do r 0 first r 1 second, or everybody has to decide to do r 1 first r 0 second.

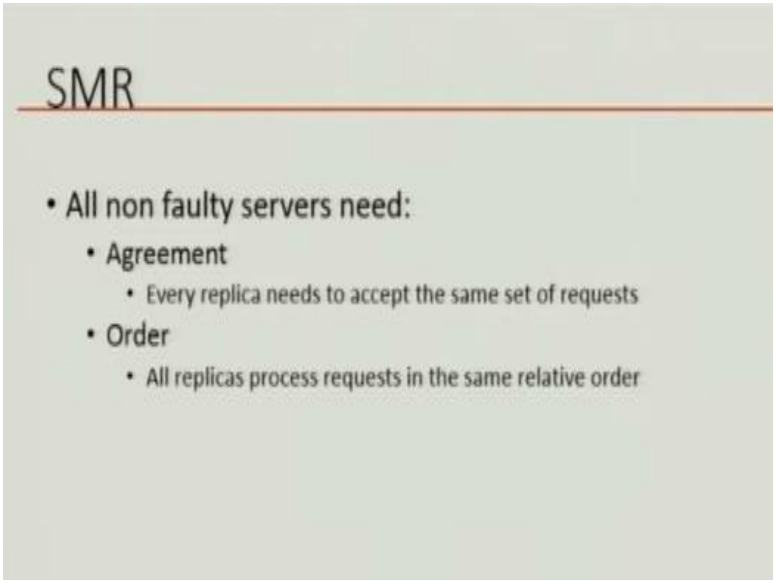
And how will they decide that because they are there, they are different servers and therefore, there has to be some communication between them before this is decided, right. So before doing the execution of the transaction, they must decide on the order of the transactions.

(Refer Slide Time: 19:22)



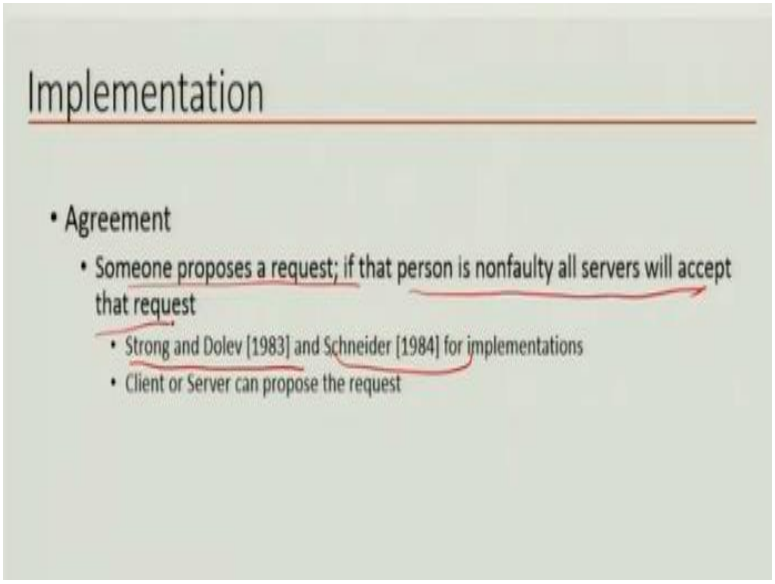
So what they will do, right they will run some kind of an algorithm.

(Refer Slide Time: 19:28)



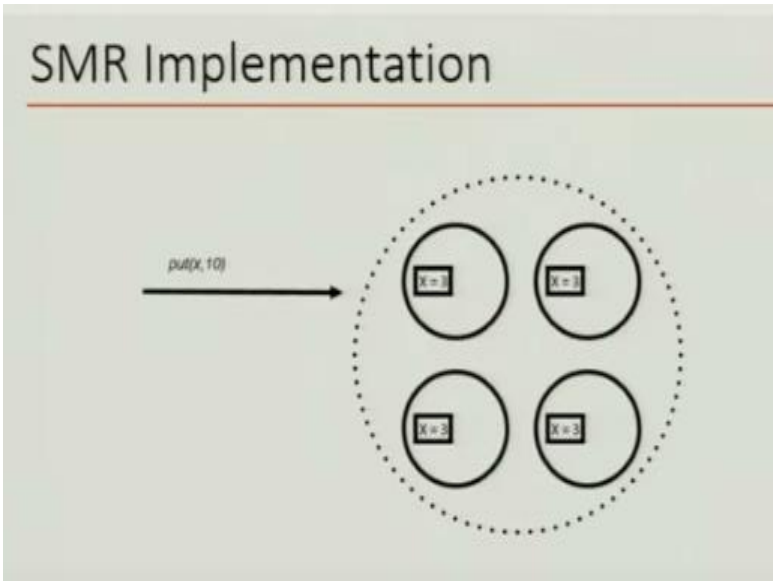
So one such algorithm before that, let us see what we want, right. So we want all non-faulty servers to agree on the order of the transactions and the and then the same set of transactions and the same relative order.

(Refer Slide Time: 19:45)

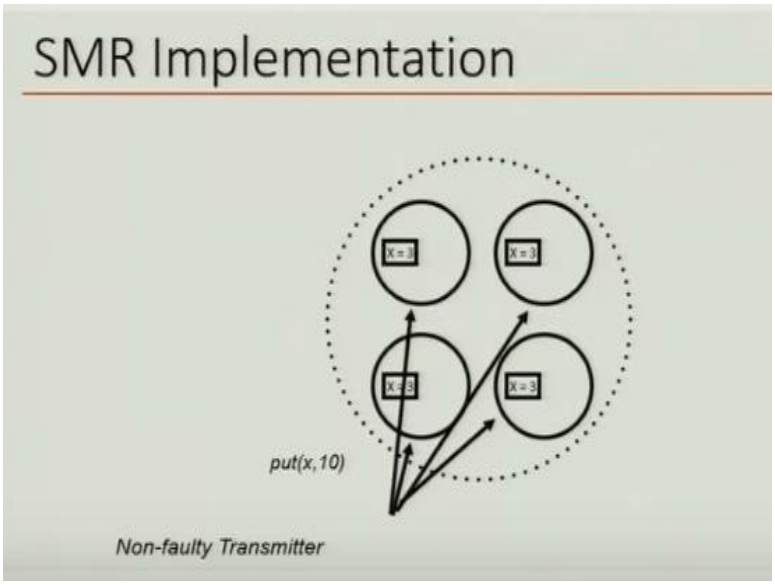


And so there is this algorithm by Strong and Dolev and then Schneider also, Fred Schneider also did some you know improvement on that. So the agreement this guarantees agreement that someone proposes a request and if that person is non faulty all servers will accept that request and any client or server can make a request.

(Refer Slide Time: 20:16)

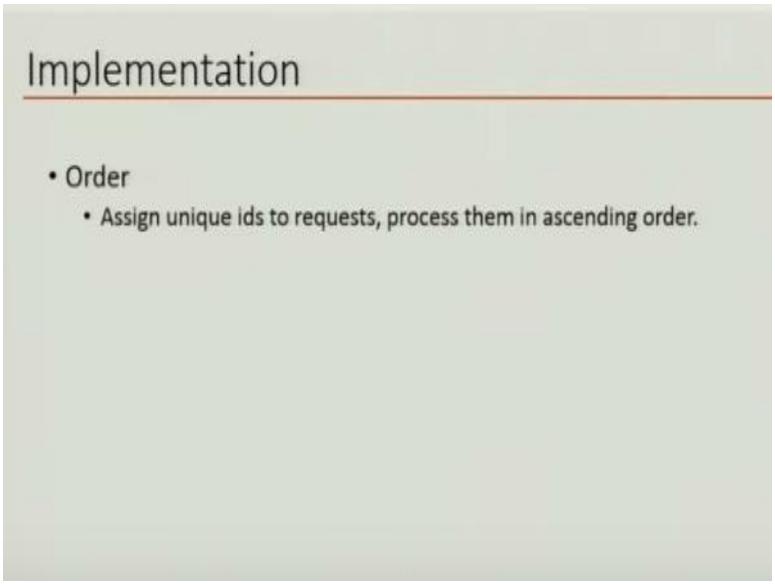


(Refer Slide Time: 20:17)



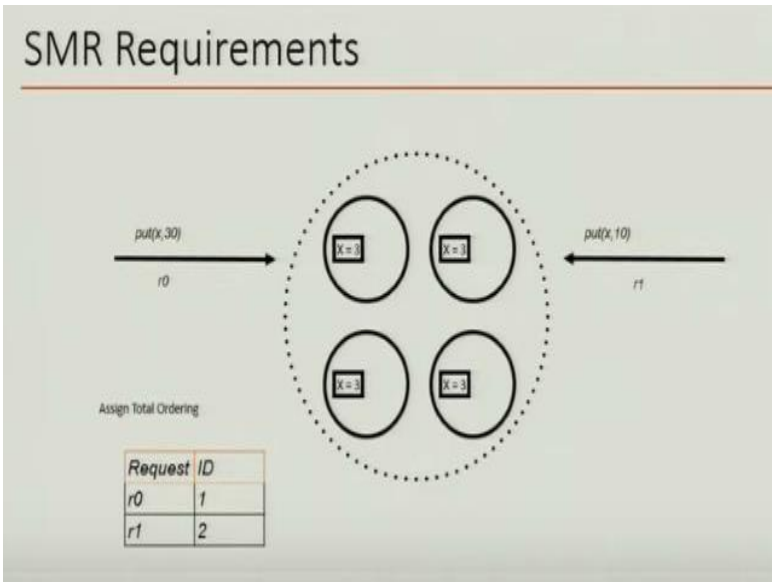
So if the non-faulty transmitter everybody will get the transaction.

(Refer Slide Time: 20:20)



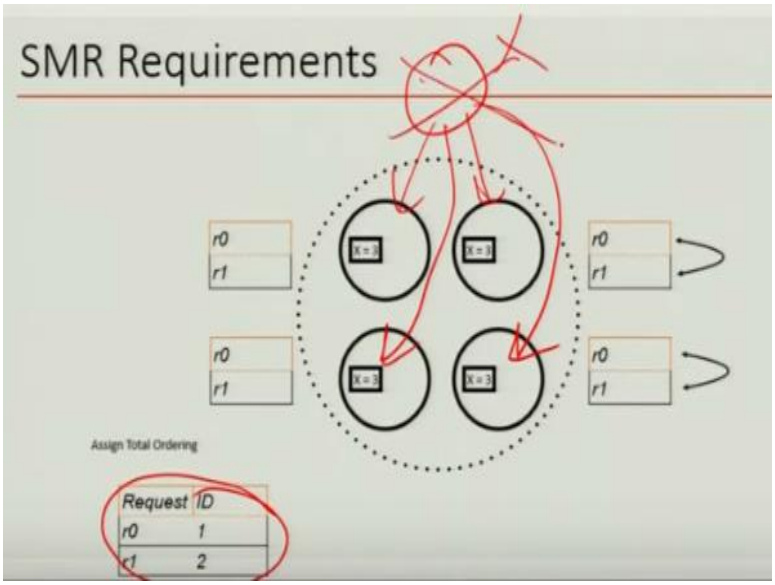
And then the ordering is done by a ID based assigning IDs. So now question is so r 0 may be given an ID and r 1 may be given an ID, but eventually everybody should decide on the same ID right. So that they, the if the IDs are given from an ordered set, then the increasing order will decide what the order of transaction will be.

(Refer Slide Time: 20:53)



So this is what we are going to do. So we want to assign total order. Now you want to assign total order. But if you leave it to each of these replicas to decide the order, then they may decide different orders. So we have to do something.

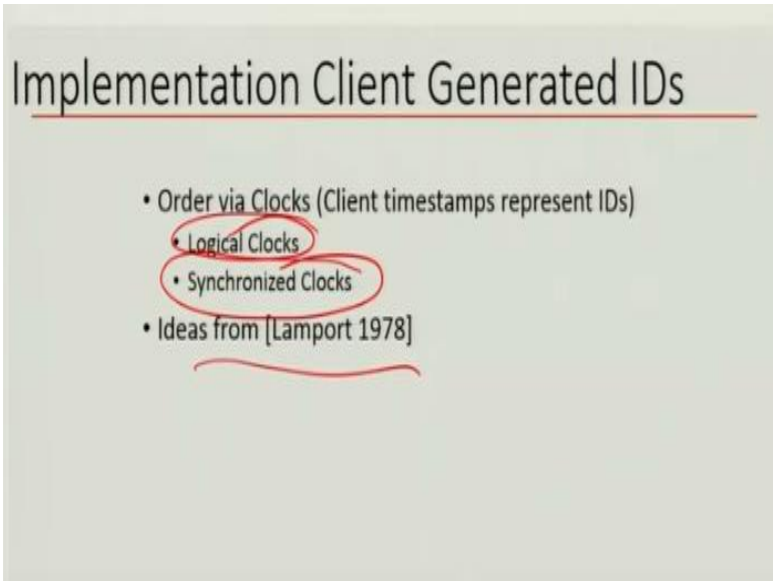
(Refer Slide Time: 21:10)



So what they will do is as follows. So they will first, so they need to reorder. If this is the final ID that we decide for everybody, then they will have to reorder the transactions. These guys are already in the order, these guys have to reorder the transaction. Now if there was a central controller here, who will tell everybody what to do, then it is no longer a distributed system.

And then and if this guy crashes, we still we are back to the same problem of not having any crash tolerance. So this cannot be the case, right. So we cannot have a scenario like this. Therefore, we must have a distributed algorithm in which everybody decides on their own, of course by exchanging some messages, but eventually we can guarantee that everybody who is non-faulty will come to the same conclusion.

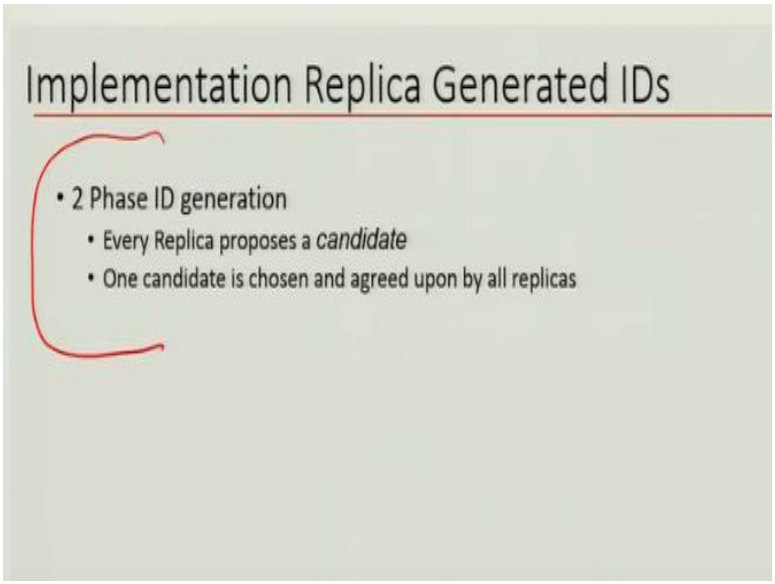
(Refer Slide Time: 22:01)



So this ordering can be done by obviously if everybody had a synchronized clock, that everybody is synchronized to the same clock then they can say okay these this transaction came at 8 pm and this transaction came at 8.01 pm. So this would be the order. But unfortunately in a distributed system clocks of all the machines are not necessarily in synchrony, right. They might have drapes and skews.

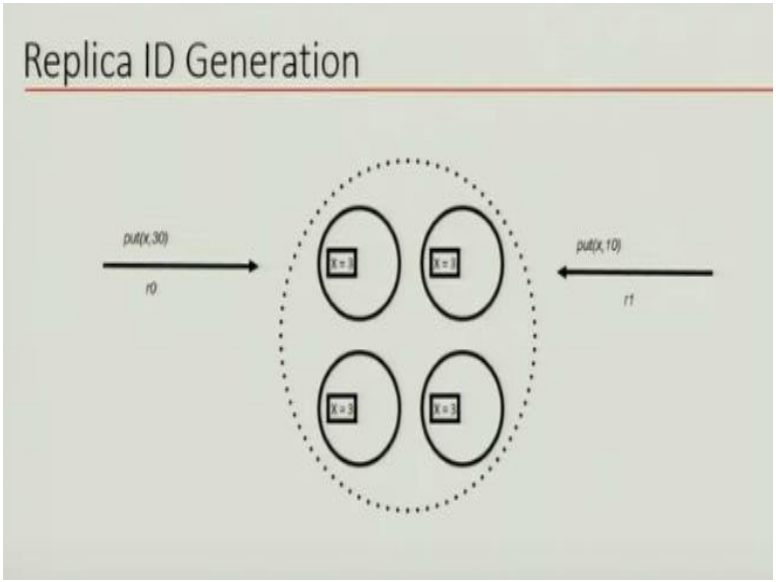
So there are algorithms for synchronizing clocks. So distributed clock synchronization is has an overhead because it requires exchanging of lots of messages. So synchronized clock is not a good solution. Logical clock is another solution. That is that was given by Lamport. But what this problem requires is not that complex.

(Refer Slide Time: 23:02)



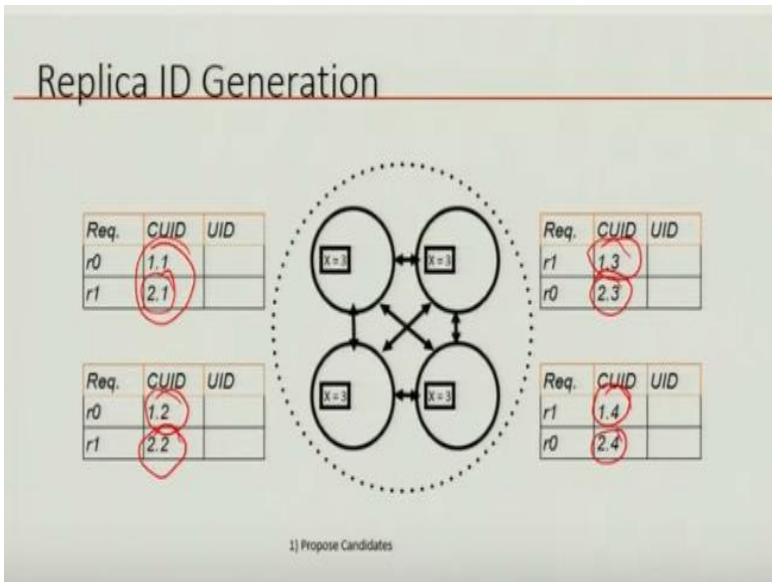
So there is a algorithm, which is called to 2 phase ID generation algorithm that was invented to solve this problem.

(Refer Slide Time: 23:18)



So what you do is that you generate first local IDs, right.

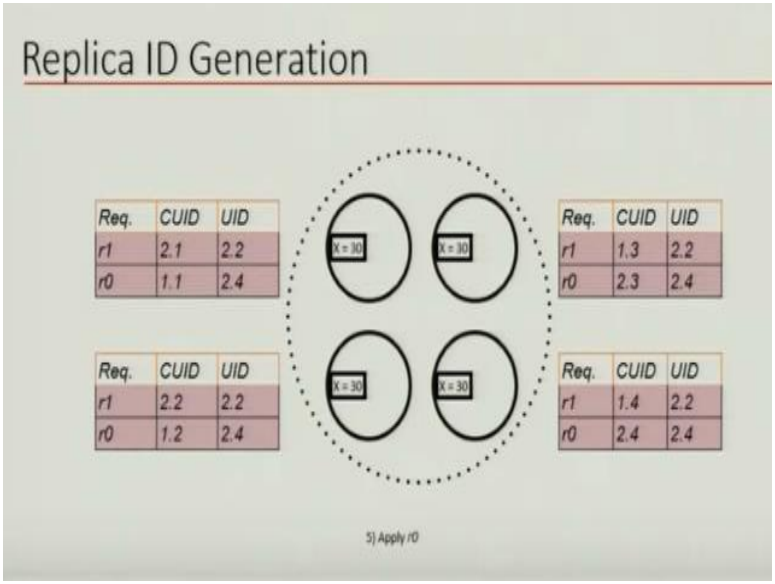
(Refer Slide Time: 23:21)



So these are called candidate IDs, candidate unique IDs. So this guy decides that the you will give r 0 1.1 and r 0 2.1. This guy decides you will give it 1.2 and 2.2. And this guy decides you will give an id 1.3, 2.3. And this will be 1.4, 2.4. This 3, 4, 1, 2 are their own IDs. So actually he is saying this is 1, this is 2. He is saying this is 1, this is 2. He is saying this is 1, this is 2, he is saying this is 1, this is 2.

Now but this is not same in all the places because for this guy, these two guys r 1 is 1 for these two guys r 0 is 1. So some algorithm has to exchange messages between them and after the messages are exchanged, so message exchange in this case will be everybody telling their local IDs for all the transactions.

(Refer Slide Time: 24:23)



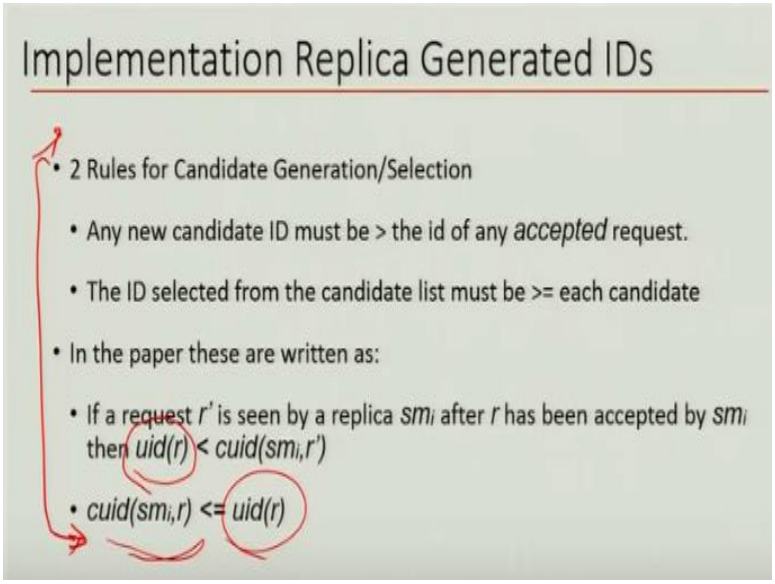
Then the algorithm says that take the highest one. So start with the one of the transaction and give the highest one. So everybody has told everybody what they have assigned to each of these transactions, right. So everybody now knows for every transaction what the id given locally, so this guy knows that r 0 has been given 1.1 by me, my this down south neighbor has given it 1.2.

On my east upwards neighbor has given me given it 2.3 and this guy has given 2.4. So then you will decide that 2.4 is the highest. So I will give 2.4 to this. This guy will also see 2.4 is the highest for r 0, he will also give 2.4. This guy already knows that this is the highest because all the messages are have gone to him. So he will also give 2.4 and this guy will already has given 2.4. Then they will consider r 1.

So this guy will say r 1 has 2.1, 2.2, 1.3 and 1.4. So 2.2 is the highest. So I will give r 1 2.2. And everybody will do the same thing. So at this point 2.2 is smaller than 2.4. So therefore, the ordering will now become r 1 first r 0 second. So that is how, so this guy will now reorder the transactions, and then r 1 will be executed first, which will make x equals 10. And then r 0 will be executed and which will make x equals 30.

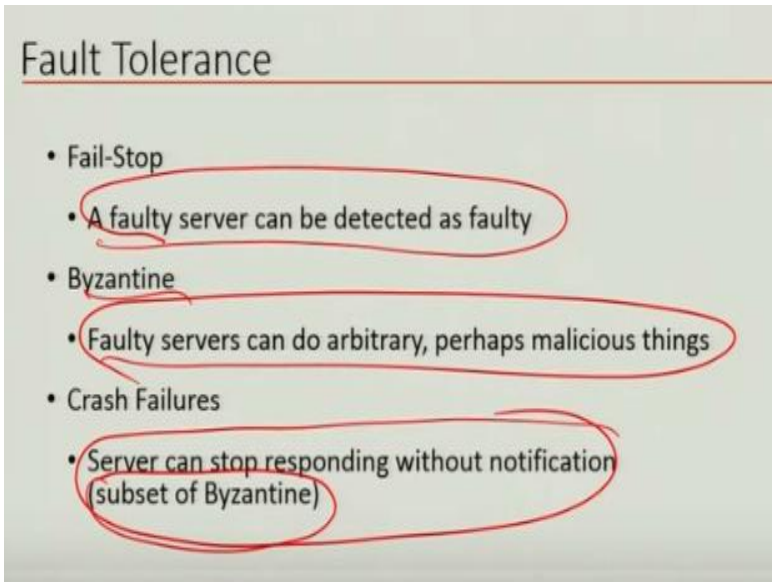
And this will be done at everybody because everybody knows the same ID, global ID for each of the transactions. So ordering problem is solved this way.

(Refer Slide Time: 26:23)



So this is the more formal discussion formal way of expressing what we just described. So you do not need to really worry about this. This basically says that, you know when you give a global ID of a transaction, then you must make sure that it is the less than the candidate ID for that. And then every candidate ID is less than is equal to the UID r so UID r would be an upper bound on all the candidate IDs for the same transaction, same request, etc.

(Refer Slide Time: 27:04)



So now this system, when we describe this system, how this happened, we did not assume any fault. So we assume that all of them are working fine and there has not

been any crash or fail-stop or Byzantine behavior right. So fail-stop means a faulty server can be detected as faulty. So it may actually declare that I am going down So everybody knows that it has gone down.

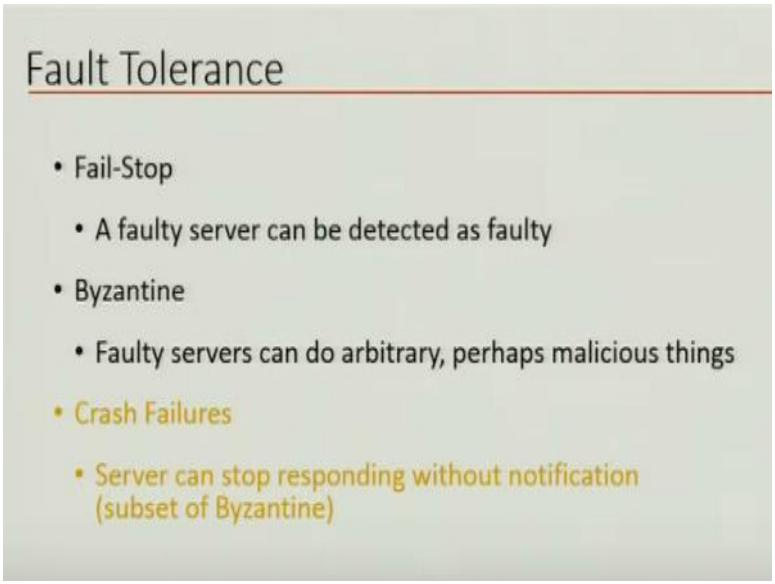
Crash failures on the other hand will be when the server stop responding. So it may be failed or it may be actually behaving, it may actually be highly alive, but it is saying it is not responding intentionally. That is sort of a Byzantine behavior. And the Byzantine behavior which will be more general that is they can do arbitrary behavior. They can send you wrong information.

In the earlier this picture that we were showing there was no wrong behavior. Everybody was honestly sending their CUIDs to everybody and that is how it was simple.

(Refer Slide Time: 28:08)

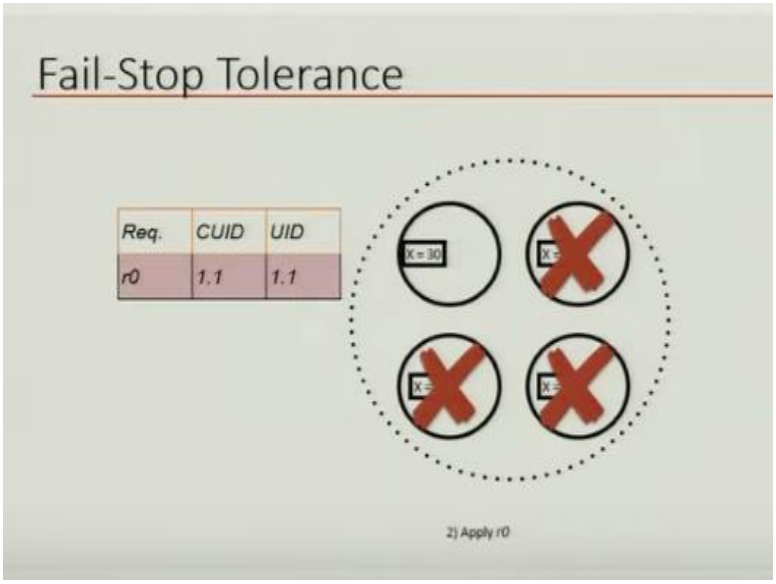
So here, let us say we have a fail-stop, right?

(Refer Slide Time: 28:13)



So fail-stop is when you can detect something is faulty. So you do not wait for yeah, we do not wait for any message from that.

(Refer Slide Time: 28:22)

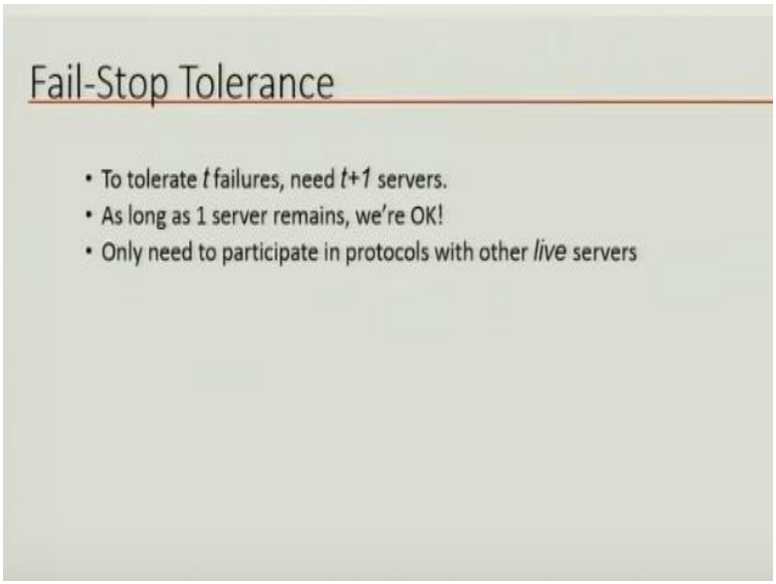


So in fail-stop tolerance, so let us say three of them fails, and you have this transaction. And then this guy is the only one working so you accept r 0 and that is it. And there is no problem.

(Refer Slide Time: 28:44)

So now all the other ones will have the same old value of 3. So now if this guy also crashes, then game is over, right? So fail-stop tolerance will only work at least one replica should survive you know to be self-stop tolerance.

(Refer Slide Time: 29:07)



So to tolerate t failures we need t + 1 servers in fail-stop tolerance. As long as one server remains, we are okay. Only need to participate in protocols with other life servers. It is the easiest to implement. You just have to have extra. If you assume t of them can fail then t + 1 servers is enough. So when we come back, we will talk about Byzantine tolerance for the same setup.