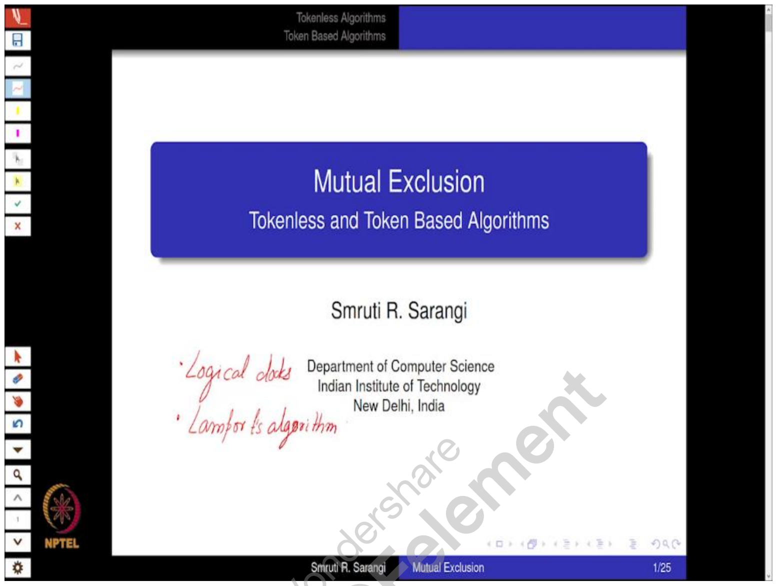
Advanced Distributed Systems
Professor Smruti R. Sarangi
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
Lecture 8
Distributed Mutual Exclusion Algorithms

(Refer Slide Time: 00:18)



Tokenless Algorithms
Token Based Algorithms

Mutual Exclusion
Tokenless and Token Based Algorithms

Smruti R. Sarangi

Department of Computer Science
Indian Institute of Technology
New Delhi, India

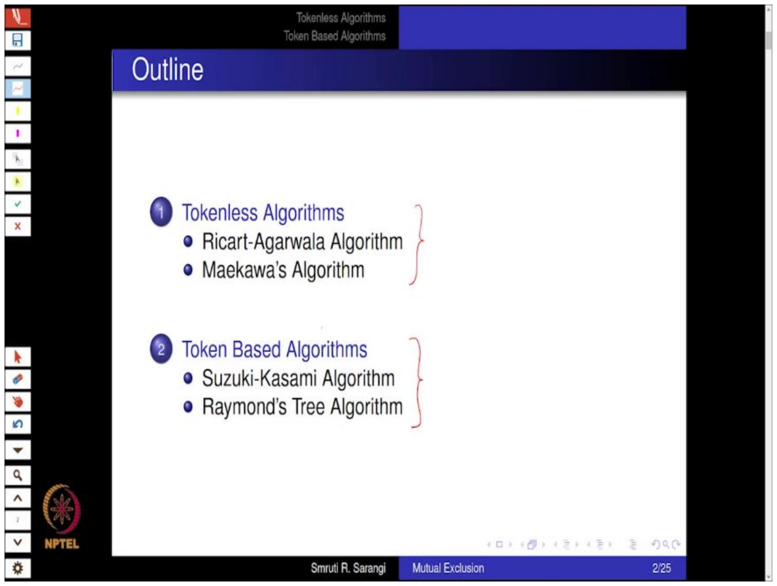
· Logical clocks
· Lamport's algorithm

Smruti R. Sarangi Mutual Exclusion 1/25

So in this lecture, we will discuss mutual exclusion algorithms. So, this will be our full-fledged lecture in which we look at distributed algorithms per se. So, we will extend the basic lecture that we had discussed last time.

So, the key prerequisites would be logical clocks, so, Lamport clocks, both scalar Lamport clocks as well as vector clocks. The next important prerequisite here would be Lamport's algorithm for mutual exclusion, of course, assuming our asynchronous setup where we do not have clock synchrony.

(Refer Slide Time: 01:15)

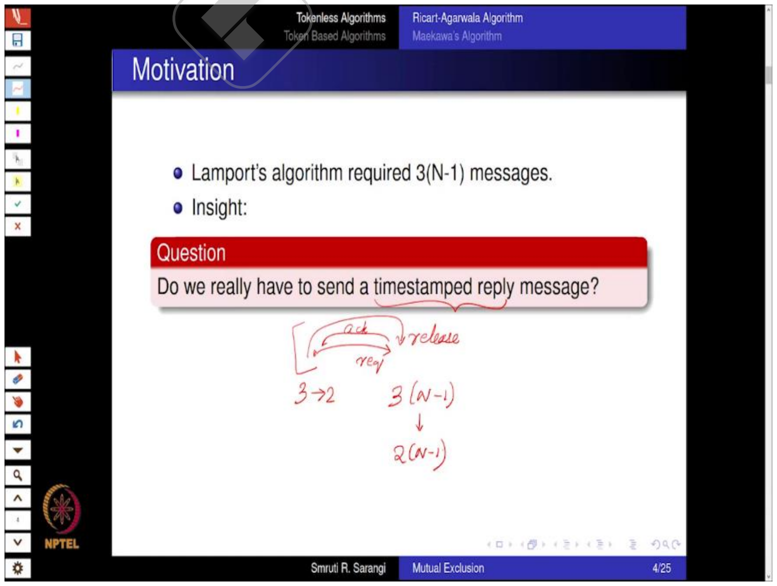


Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

All so, in this lecture we will discuss two kinds of algorithms. One is without tokens, so we will discuss the Ricart-Agarwala Algorithm, and the Maekawa’s Algorithm. Then, we will discuss token-based algorithms as a part of this slide set, where we will discuss two algorithms, again, the Suzuki-Kasami Algorithm and the Raymond's Tree Algorithm.

(Refer Slide Time: 01:41)



Motivation

- Lamport's algorithm required 3(N-1) messages.
- Insight:

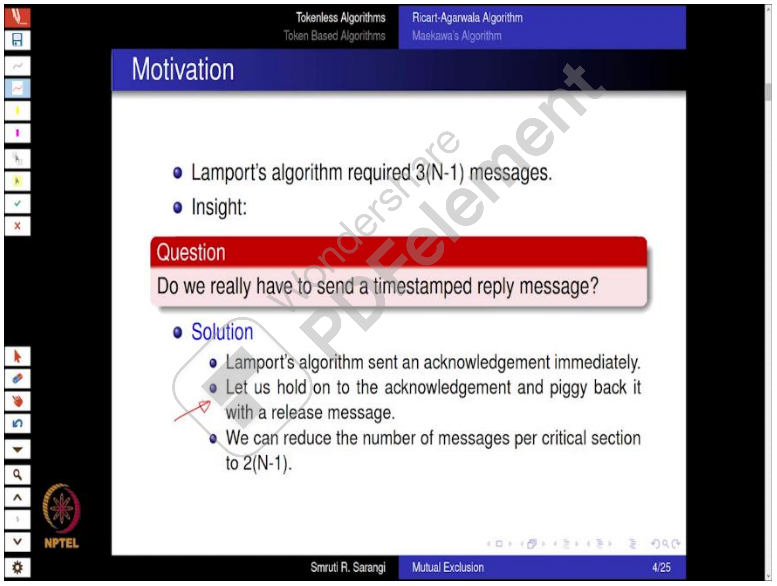
Question

Do we really have to send a timestamped reply message?

So coming to the Ricart-Agarwala Algorithm, so again the key prereq is to understand the Lamport's Algorithm that was a part of the previous lecture. So in that algorithm we said that we would require 3(N – 1) messages for achieving mutual exclusion, and we had proved that it is correct. So now I am just extending from there, and of course, the prereq is that that algorithm is completely understood.

So, that algorithm has three rounds. So, where we send a request, then we send an ack, and then we send a release. So, the question is that is a time stamp to reply or an acknowledgement necessary. So, that is the key question. And, so, the main idea is that instead of these 3 messages that I will send to every single node, can I make 3 to 2? Then instead of 3(N – 1) messages, I will have to send 2(N - 1) messages, which is an improvement.

(Refer Slide Time: 02:52)


Tokenless Algorithms
Token Based Algorithms
Ricart-Agarwala Algorithm
Maekawa's Algorithm

Motivation

- Lamport's algorithm required 3(N-1) messages.
- Insight:

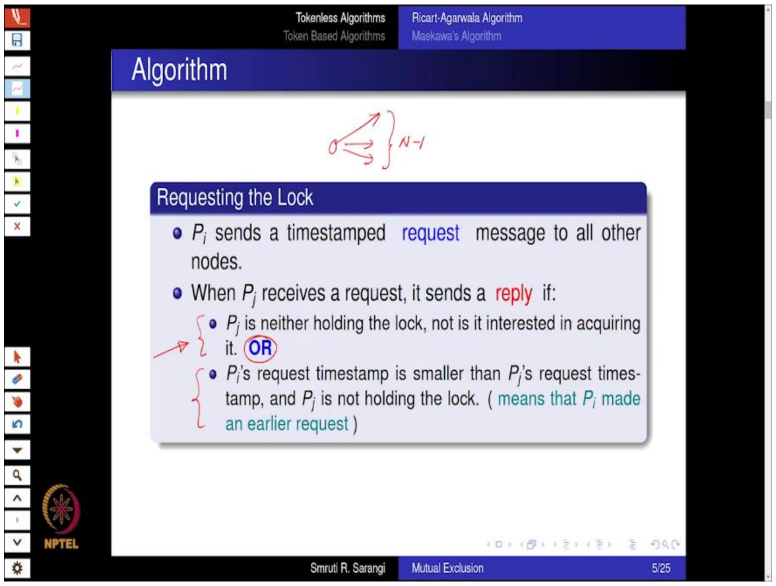
Question
Do we really have to send a timestamped reply message?

- Solution
  - Lamport's algorithm sent an acknowledgement immediately.
  - Let us hold on to the acknowledgement and piggy back it with a release message.
  - We can reduce the number of messages per critical section to 2(N-1).

Smruti R. Sarangi Mutual Exclusion 4/25

So, the idea was that in the Lamport's algorithm, we send an acknowledgement immediately because we have to show that another node has recorded the message and it is sending an acknowledgement with a greater timestamp. So, the key insight over here is that we hold on to the acknowledgement and we piggyback the acknowledgement along with a release message. We will see how. So, this will help us reduce the number of messages from 3N to 2N.

(Refer Slide Time: 03:22)



Algorithm

Requesting the Lock

- P_i sends a timestamped request message to all other nodes.
- When P_j receives a request, it sends a reply if:
  - P_j is neither holding the lock, not is it interested in acquiring it. OR
  - P_i's request timestamp is smaller than P_j's request timestamp, and P_j is not holding the lock. (means that P_i made an earlier request)

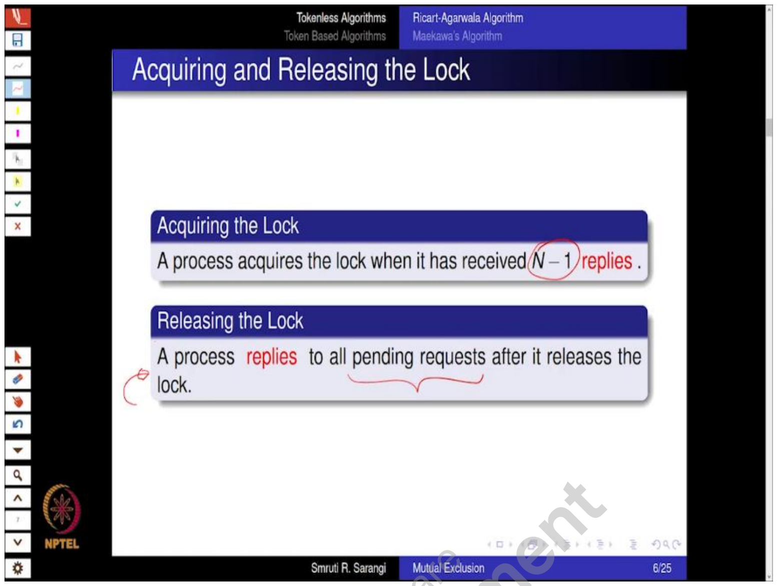
So, here is algorithm. Process i sends a timestamp request message to all other nodes. So, we had seen this. So, this part is the same. To all the other nodes, the rest of the other nodes, it sends a timestamp request message. When Process j receives a request, it sends a reply if. So this part changes.

The first is P j is neither holding the lock, nor is it interested in acquiring it. So, if it is not interested at that point of time in acquiring it or if it is not holding it, it just sends a reply and the reply would follow the same rule for scalar Lamport clock, so its timestamp would be higher than the request.

Or second condition, P i’s request timestamp is smaller than P j's request timestamp, and P j is not holding the lock. So this basically means that P i made an earlier request. So even though P j is desirous of acquiring the lock, so, lock means the shared resource. I will often use lock in place of shared resource. So this, kindly understand, because in the concurrent systems world any shared record, any shared resource is actually called a lock.

So coming back to our discussion, the thing is that look if P j is not holding the lock or not interested, it will send a reply, or if it is interested, it has send out a request but P i’s request happens to be smaller, the way we defined it in the previous lecture, than P j’s request, then also it will send a timestamped reply. So this part changes, and this part is crucial. So this key idea is that P j will send a reply only if both of these conditions hold.

(Refer Slide Time: 05:29)



Tokenless Algorithms | Ricart-Agarwala Algorithm
Token Based Algorithms | Maekawa's Algorithm

Acquiring and Releasing the Lock

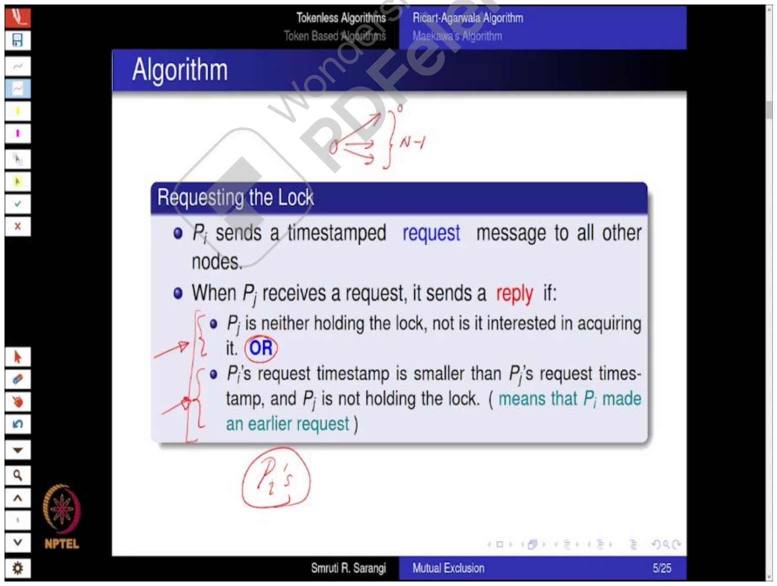
Acquiring the Lock

A process acquires the lock when it has received $(N-1)$ replies.

Releasing the Lock

A process replies to all pending requests after it releases the lock.

Smruti R. Sarangi | Mutual Exclusion | 6/25



Tokenless Algorithms | Ricart-Agarwala Algorithm
Token Based Algorithms | Maekawa's Algorithm

Algorithm

Requesting the Lock

- P_i sends a timestamped request message to all other nodes.
- When P_j receives a request, it sends a reply if:
  - P_j is neither holding the lock, not is it interested in acquiring it. OR
  - P_i's request timestamp is smaller than P_j's request timestamp, and P_j is not holding the lock. (means that P_i made an earlier request)

Smruti R. Sarangi | Mutual Exclusion | 5/25

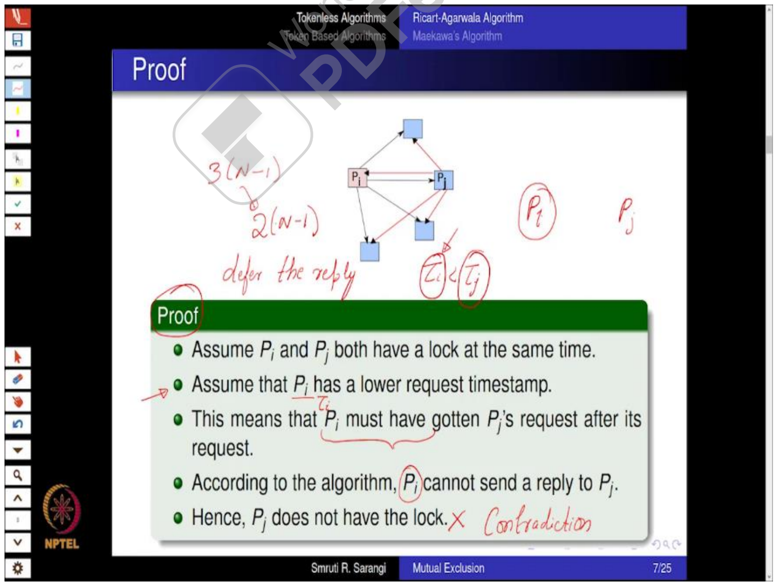
So now, when you acquire a lock, well you acquire the lock when a process has received (N – 1) replies. So let me just go back. So, what would happen is that P j will not send a reply as long as these conditions hold. So, let me put it in a different way. It is not that P j is going to junk the request from P i. Process j is not going to do that. What Process j will do? P j will do, is that it will hang on to P i’s request until one of these conditions is true.

Subsequently, it will send a reply and once a process P i has received the (N – 1) replies, it knows it can acquire the lock, so it will acquire the lock and finish its job. And after, so releasing the lock basically would mean doing the same, which means replying to all the pending requests which the request it has kept pending, it has not replied to, it will reply to them. After it releases the lock, it will reply to all the pending requests.

So, one thing that is obvious over here is that instead of sending three messages to every node, we are sending two. So the release is not there. The ack and the release, we are fusing into one message. So instead of sending three, we are sending two. So the key idea over here is that I send a request, other nodes simply hang on to my request, keep it pending until one of these conditions holds true. Once they hold true, it holds true, they send a reply and they get rid of my request. So then cleaning up the state is not an issue.

And once I get all my replies, I enter, and subsequently, if I have kept, if Process i, which is me, if I have kept other requests pending, I will reply to them.

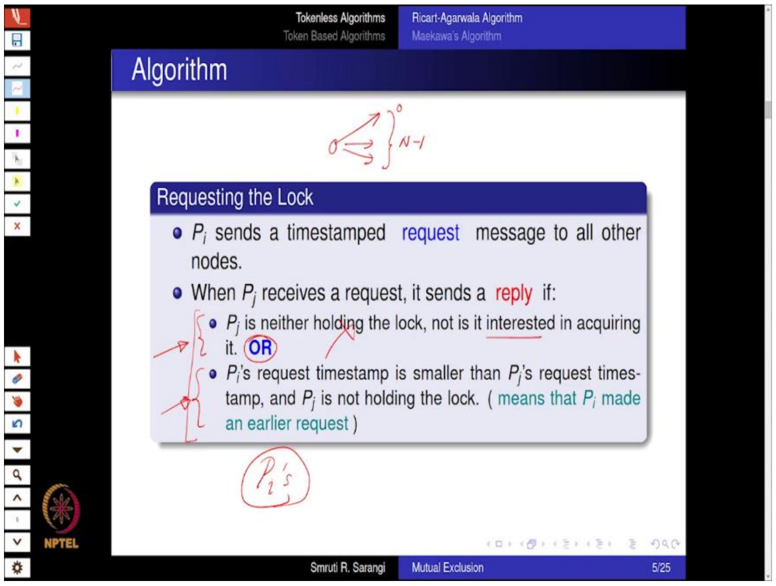
(Refer Slide Time: 07:30)



Proof

Proof

- Assume P_i and P_j both have a lock at the same time.
- Assume that P_i has a lower request timestamp.
- This means that P_j must have gotten P_i's request after its request.
- According to the algorithm, P_j cannot send a reply to P_i.
- Hence, P_i does not have the lock. ✗ Contradiction



Algorithm

Requesting the Lock

- P_i sends a timestamped request message to all other nodes.
- When P_j receives a request, it sends a reply if:
  - P_j is neither holding the lock, not is it interested in acquiring it. OR
  - P_i's request timestamp is smaller than P_j's request timestamp, and P_j is not holding the lock. (means that P_i made an earlier request)

P_i's

So, this is a simple algorithm. What about the proof? So, hopefully, the algorithm and understanding is 2(N – 1) has not been an issue. But again, I would like to say that the Lamport's algorithm is a prerequisite. Kindly not go through this unless you have understood that thoroughly.

So again, let us prove by contradictions. So assume that processes P i and P j both have acquired the lock at the same time. With no loss of generality, same approach, assume P i has a lower request timestamp. This means that P i must have gotten P j's request after its request. That is obvious.

Why? Because the thing is that if let us say P i’s timestamp is T i, then it could not have gotten P j's request, because otherwise, T i would have had a higher timestamp and this relationship would not have held. Given the fact that this relationship is holding, it means that P i must have sent its request, must have gotten P j's request after it sent its own.

So, then this means, according to this algorithm, the moment Process i got the request from Process j, it had already sent out its request. Furthermore, its request timestamp was less than the timestamp of Request j. Consequently, it is not possible via this condition. First, it was interested in acquiring the lock and second, of course, this condition does not hold but for this condition, when it was interested, the request timestamp of P j was higher than its request timestamp, hence it could not have sent a reply.

Given the fact that P i could not have sent a reply to P j, P j would have definitely not gotten a reply from i because i was the one with the earlier request and the one with the lower timestamp. Consequently, P i did not send a reply to P j. So clearly P j’s conditions that it would have gotten a reply from everybody would not have held. Hence, P j could not have acquired the lock. Given that P j could not have acquired the lock, we have a straightforward contradiction over here.

So, I am not explaining this proof once again because this proof, if you think about it, is extremely simple. But I would request the viewers to keep replaying this part of the video where I explain the proof until they understand it thoroughly. Let me explain it in another way. The key idea over here is that unlike the Lamport's algorithm where we send a reply immediately, here, we hang on to the reply.

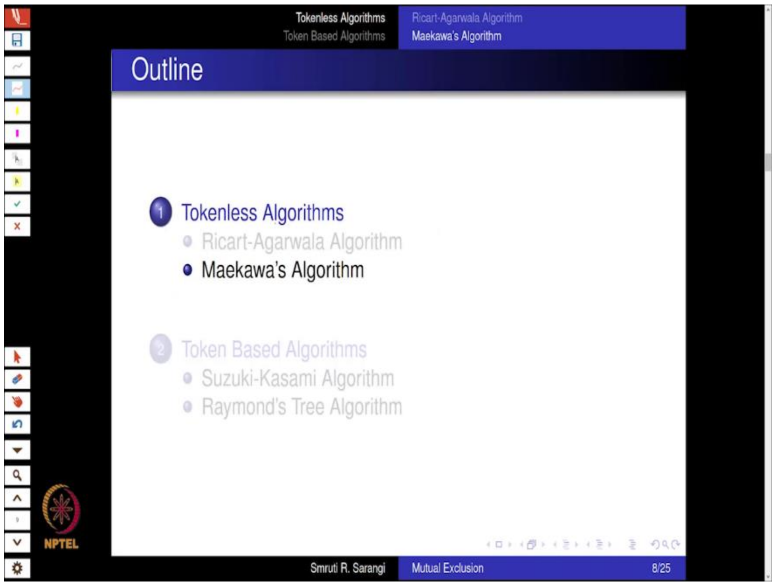
So, the argument that is being made, the same argument was made over there also that there is no way that i would have received, Process i would have received Process j's request before it sent its request. Otherwise, by the law of scalar clocks, it would have had a higher timestamp. That is not the case, which means, it receives after it.

And if it receives after it, it already has its own request in its queue, which has a lower timestamp. So reply could not have been sent. If a reply could not have been sent, there is no way that P j could have entered, could have acquired the lock, I was about to say enter the critical section, but again that is also not wrong because that is also another way that we refer to mutually exclusive objects in concurrent systems, but I would prefer not using that.

So, but the key idea is, P j could not have concurrently entered the critical section or acquired the lock, same thing. So, there is a contradiction here, which means proof by contradiction, the algorithm is correct. And as opposed to 3(N – 1) all, we have an approach which is 2(N – 1). So clearly, there is a 50 % benefit. And the key insight is that defer the reply. That is the key insight.

So every algorithm has to give us an insight. So the key insight over here is that we simply defer the reply. So this deferral is reducing 3 to 2.

(Refer Slide Time: 12:25)

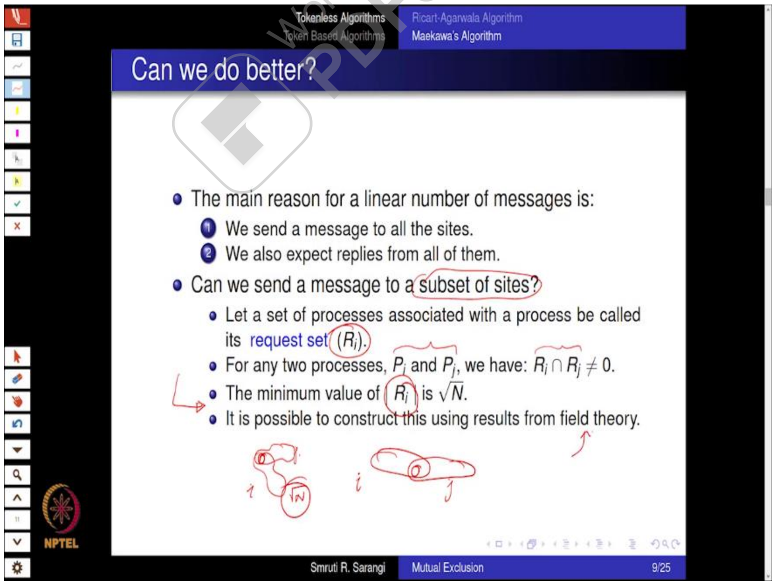


Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Now, we will try to reduce this order N business to order $\sqrt{N}$. So, we will discuss the Maekawa's algorithm.

(Refer Slide Time: 12:34)



Can we do better?

- The main reason for a linear number of messages is:
  1. We send a message to all the sites.
  2. We also expect replies from all of them.
- Can we send a message to a subset of sites?
  - Let a set of processes associated with a process be called its request set (R_i).
  - For any two processes, P_i and P_j, we have: $R_i \cap R_j \neq 0$.
  - The minimum value of R_i is $\sqrt{N}$.
  - It is possible to construct this using results from field theory.

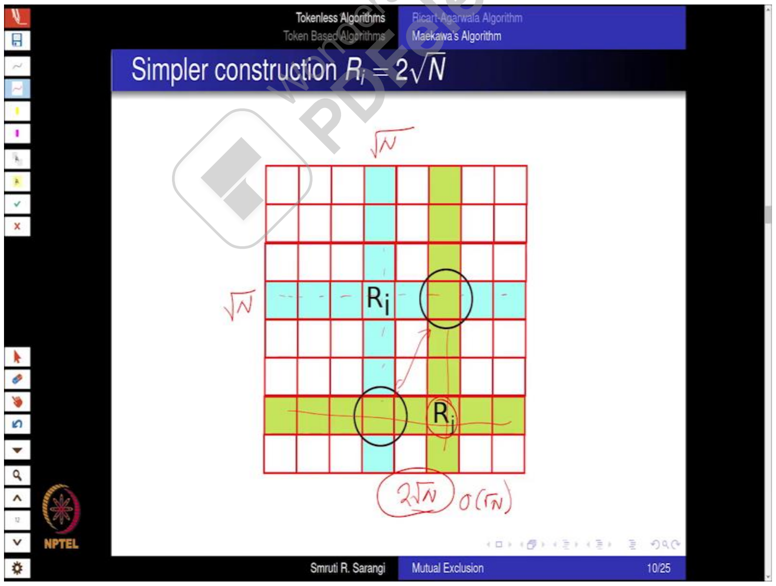
So, the main reason for a linear number of messages is basically we send a message to all the sites and we also expect replies from them. So the key insight is let us not send a message to all sites let us send a message to a subset of the sites. So let a set of processes associated with a process be called as request set R i, which is a subset.

For any two processes, let us guarantee that for two processes P i and P j, we have R i ∩ R j ≠ 0, it is non-null, which basically means that if Process i has an intersection set, if Process j has a request set, then the intersection regardless of I and j is always non-null, or which means there is one process which is, which lies in the intersections of both the request sets.

So we are not considering faults here. So let us take faults out of the picture. So it appears, so it happens to be the case that it is possible to construct such request sets where we can say that the size of each request set $\sqrt{N}$ values, or what I can say is that for every i I can have a request set so as I consider any Process j, its request set, there will at least be one process in common.

So the intersection will be non-null, and the size of these will be in the ballpark of $\sqrt{N}$. So of course, this will require results from field theory, so we will not go that far. We will consider a simpler argument.

(Refer Slide Time: 14:15)



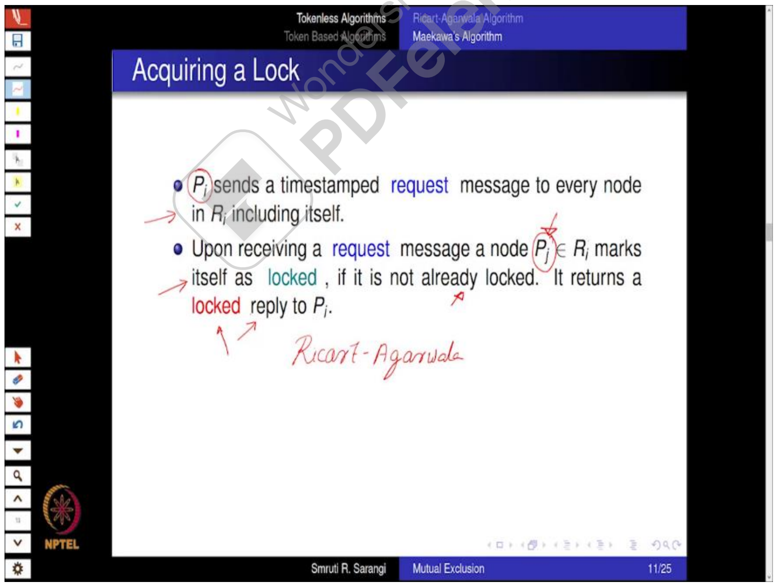
Simpler construction $R_i = 2\sqrt{N}$

So, consider N cross N processes, and consider Process i. So, Process i, I can give it a coordinate in the 2D space. So, or let us put it, so consider N processes and let us, consider N processes, let us arrange them in a $\sqrt{N}$ cross $\sqrt{N}$ matrix. And let be, this be the position of i. So, I can say that all the processes in its row and in its column comprise its request set.

So, what did I do? I took all the processes, N processes, arrange them in a $\sqrt{N}$ cross $\sqrt{N}$ matrix. So every process has a coordinate in this matrix. And I say that for Process i, I will consider all the processes in its row and in its column to be a part of its request set. And this automatically guarantees to me that regardless of the j, the value of j that you choose, for example it can be this one, as you can see, this will be the interest, request set of j and the intersection is non-null.

So of course, the size of the request set over here is 2 $\sqrt{N}$, but that is okay. We will live with this. It is still order $\sqrt{N}$, so we will live with this. And let us see, using this assumption, let us go forward. But of course the only assumption we will need is that the intersection is non-null. And bear in mind that this is a simple way of constructing such kind of non-null intersecting request sets. But if you want to go to $\sqrt{N}$, you will have to use field theory, and that is also not that hard.

(Refer Slide Time: 15:55)



Tokenless Algorithms
Token Based Algorithms
Ricart-Agarwala Algorithm
Maekawa's Algorithm

Acquiring a Lock

- P_i sends a timestamped request message to every node in R_i including itself.
- Upon receiving a request message a node $P_j \in R_i$ marks itself as locked, if it is not already locked. It returns a locked reply to P_i.

Ricart-Agarwala

Smruti R. Sarangi Mutual Exclusion 11/25

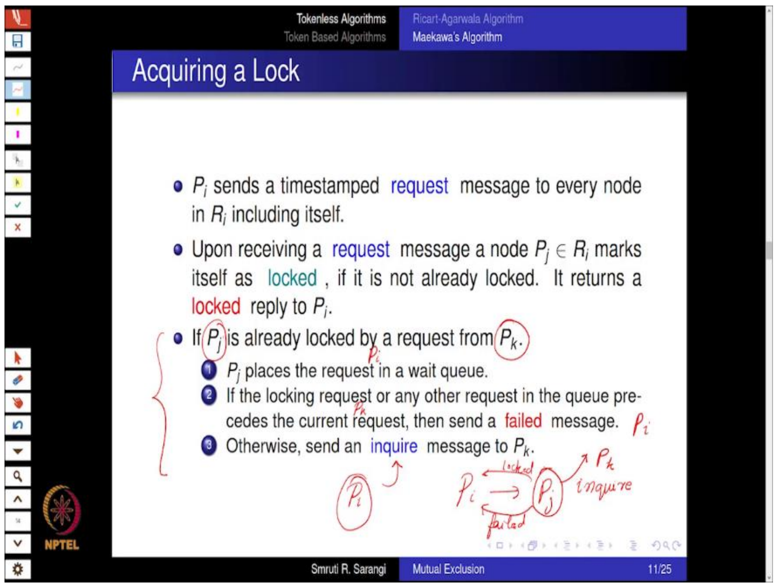
So here is the algorithm. So this algorithm is slightly involved but not all that involved. So, the first part is the same. So the first phase is the same where Process i sends a timestamped request message to every node in R i, including itself. So previously what was happening, that you are sending a request message to every other node in the system. This is not happening now. A message is being sent to every node only in the request set which includes itself.

Upon receiving a request message, a node P j which is a part of R i, marks itself as locked if it is not already locked. And it returns a locked reply to P i. So if you can see this part is pretty similar to the Ricart-Agarwala construction. So it is not all that different at all. So, up till now it appears that we are proceeding along the same line, where if a node P j, process P j, in the request set if it is not already locked, then it says look, I have no issues in being locked for Process i. So let me mark myself as locked.

So, it will return a locked reply, it will mark itself as locked, and it will return a locked reply to P i in the sense it is committing to P i that I stand behind you and if you want to acquire the lock, go ahead and acquire it. And essentially, I have reserved myself for you. So, this is similar to the Ricart-Agarwala construction, even though this notion of reservation was not there.

But nevertheless, the key idea is that if the node P j, if it is unlocked, it will mark itself as locked and sent a lock message back. So in the trivial case where there are no other competitors, the entire request set will send a lock message back, and then Process i will be sure that everybody in its request set, all of them were free and then there are no objections with me acquiring the lock. So, Process i can simply go forward.

(Refer Slide Time: 18:16)


Tokenless Algorithms
Token Based Algorithms
Ricart-Agarwala Algorithm
Maekawa's Algorithm

Acquiring a Lock

- P_i sends a timestamped request message to every node in R_i including itself.
- Upon receiving a request message a node $P_j \in R_i$ marks itself as locked, if it is not already locked. It returns a locked reply to P_i.
- If P_j is already locked by a request from P_k.
  1. P_j places the request in a wait queue.
  2. If the locking request or any other request in the queue precedes the current request, then send a failed message.
  3. Otherwise, send an inquire message to P_k.

Smruti R. Sarangi　Mutual Exclusion　11/25

Now, let us look at the next part. This part is slightly interesting. If P j is already locked by a request from another process P k, here is what we need to do. P j will then place the request from P i, which is me, in a wait queue. So it uses the same basic notion of deferral that we used in the Ricart-Agarwala construction, that if P j is already committed to some other process P k, it will place P i's request in a wait queue.

If the locking request or any other request in the queue precedes the current request, then send a failed message. If the locking request, in this case the locking request is from P k. So if this request or any other request in the queue is, before, it precedes the current request in the sense it has a lower timestamp, then we send a failed message to P i.

So, in this case what happens is that P j basically tells P i that look, I have other messages in my queue which have a lower timestamp than you, so hence I am sorry I cannot lock and I am sending a failed message. Otherwise if that is not the case, which basically means that if P i is coming with the least timestamp, so then the message from P i should get a higher priority. As a result, an enquire message is sent to P k to find out what is happening.

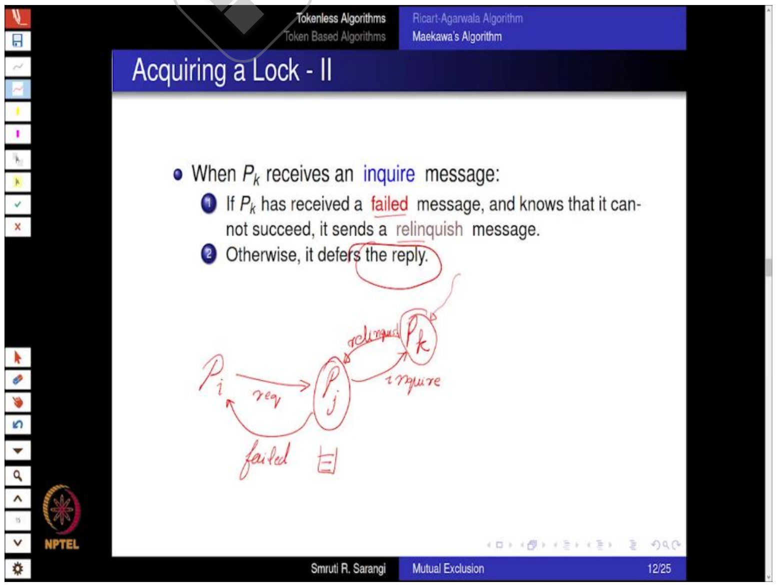
So what is the key idea? The key idea is that from the point of view of P j, which is getting a request from P i, if it is unlocked, no problem, it will send a locked message. If, let us say, it is locked, then it will see what is the priority of the request from P i, vis-à-vis the other messages it has. If it has a lower timestamp, then it clearly means that it is a high priority message. So it sends an enquired message to P k which has locked it to ask what has happened.

If that is not the case, then it sends a failed message to P i basically telling it that look I was not able to lock. So these messages are clear. So the idea basically is that the contention is at P j which is already committed, if I am a high priority requester in the sense my priority is low, my timestamp is low, high priority means timestamp is low, then what I do is I request P j to inquire with P k.

This is always the case. I am in an office, somebody important comes, then I just gauge the person's importance with respect to what I am currently doing. So let us say, I am talking to somebody else on the phone, and let us say somebody comes who is extremely important, then I place the phone down and I talk to the person.

But vice versa, if I am talking to somebody very important and let us say somebody less important comes to my room, then I just continue talking and send a failed message. So this is very similar to that. It mimics human behavior in this part.

(Refer Slide Time: 21:45)



Acquiring a Lock - II

- When P_k receives an inquire message:
  1. If P_k has received a failed message, and knows that it cannot succeed, it sends a relinquish message.
  2. Otherwise, it defers the reply.

So, when P k receives an inquire message it basically means that somebody is knocking on his door that look, somebody more important than you has arrived. What happened? If P k has received a failed message and knows that it cannot succeed, then it will send a relinquish message.

Fair enough. So if P k knows that it does not have a chance of acquiring the lock because it has already been spurned in that sense, it will send a relinquish message. Otherwise, it will simply defer the reply and it will not do anything. So who are the actors in this game? The actors in this game are P I, P j and P k. So think of this as let us say P j is me and I am talking to P k on the phone and then P i arrives.

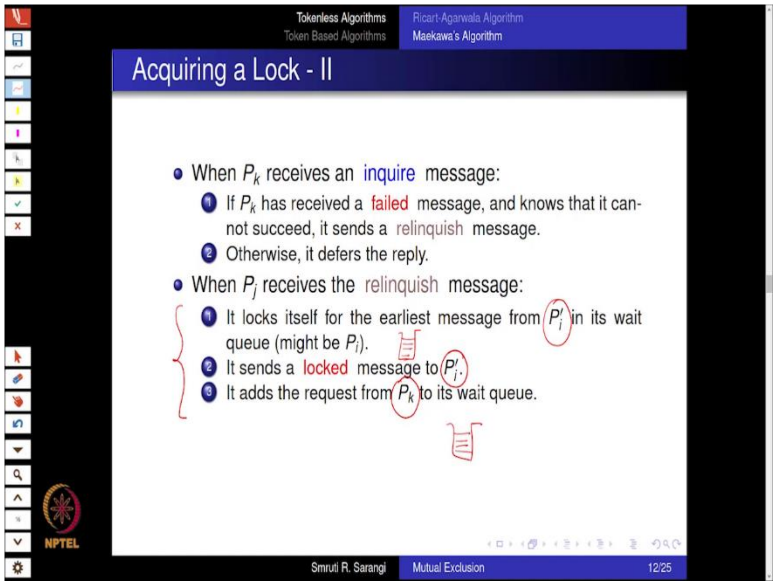
So, if P i sends a request message, so of course, the trivial cases that there is no P k in the picture. In this case, I just send a locked message to P i and I am done. But I am not looking at that. I am looking at the more interesting case where I am talking to P k in the sense I am already committed to P k, and P i arrives.

So, if it is the case that either P k has a lower timestamp than P i or have other requests in my request queue who have a lower timestamp than P i, I simply tell P i that look, you have no chance, you are too unimportant for me. So you, in a sense, go away. So I send a failed message.

If that is not the case and P i happens to be somebody very important like, for example if I am talking to a student on phone, and the Head of the Department arrives outside my door, well then, what I tell is, I send the student an inquire message, and I tell the student do you mind if I put the phone down. So then the student basically sees if, if there is any chance of continuing the conversation.

So, in this case P k basically sees if it has received a failed message or not from others. If it knows, anyway it does not have a chance. He does not have a chance in acquiring the resource. In this case, it means that does not have a chance in getting whatever he or she wanted. Then of course it makes sense to send a relinquish message to say that look, you go and lock yourself for somebody else, I do not care. Otherwise, what will happen is that P k will simply defer the reply and reply later.

(Refer Slide Time: 24:17)



Acquiring a Lock - II

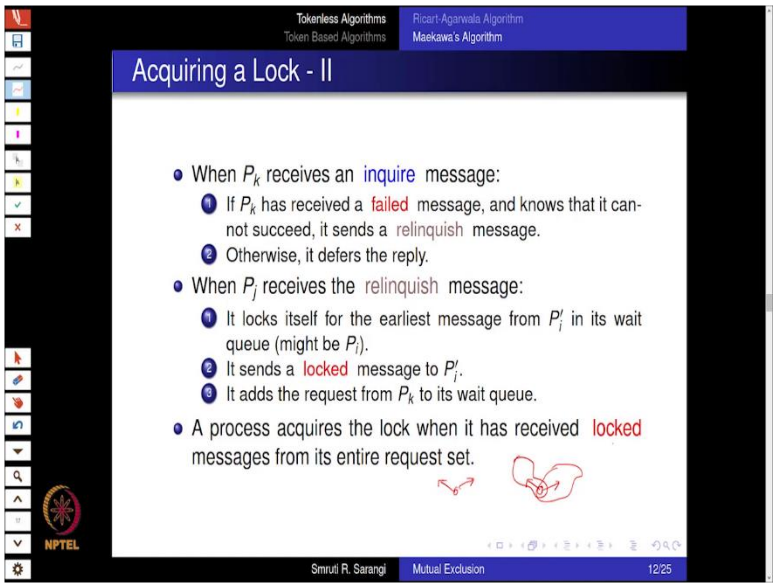
- When P_k receives an inquire message:
  1. If P_k has received a failed message, and knows that it cannot succeed, it sends a relinquish message.
  2. Otherwise, it defers the reply.
- When P_j receives the relinquish message:
  1. It locks itself for the earliest message from P_i' in its wait queue (might be P_i).
  2. It sends a locked message to P_i'.
  3. It adds the request from P_k to its wait queue.

This sounds to be a lot of fun with P i, P j and P k. Now, what else happens? So now when P j receives the relinquish message, it realizes that P k does not have a chance of acquiring the lock. So what needs to be done? So it locks itself for the earliest message from P'i in its wait queue.

So what it does is, it takes a look at its wait queue. So in the wait queue there are a bunch of requests including the one from P i. It locks itself, it locks itself for the earliest message from process P'i, which could be P i. So it locks itself. It changes its status. So it basically means if the student tells me on phone that the student is okay with me hanging up, I just hang up and talk to my Head of the Department.

Furthermore, what it does is that it sends a locked message to P'i telling it that look, I have locked myself for you. And it adds the request from P k, the one it just ditched into its wait queue. So essentially, the idea is that if there is somebody important at your door, you ask the person whom you are talking to on phone to kindly hang up, and of course, you are polite, in the sense, you wait for a reply from that side. So in this case this is what is meant by P k deferring the reply.

(Refer Slide Time: 25:49)



Tokenless Algorithms
Token Based Algorithms
Ricart-Agarwala Algorithm
Maekawa's Algorithm

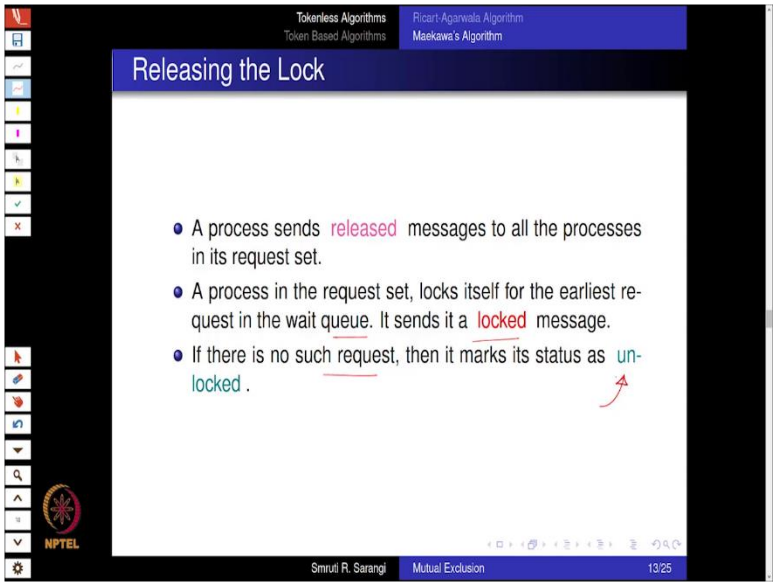
Acquiring a Lock - II

- When P_k receives an inquire message:
  1. If P_k has received a failed message, and knows that it cannot succeed, it sends a relinquish message.
  2. Otherwise, it defers the reply.
- When P_j receives the relinquish message:
  1. It locks itself for the earliest message from P_i' in its wait queue (might be P_i).
  2. It sends a locked message to P_i'.
  3. It adds the request from P_k to its wait queue.
- A process acquires the lock when it has received locked messages from its entire request set.

So, this sounds simple enough. So when does the process acquire the lock? It acquires a lock when it has received lock messages from its entire request set. So when from the entire request set it gets locked messages, it acquires it. And why is this algorithm correct? Well, the algorithm is, of course, correct. The reason being, that one process can never lock itself for two other processes.

So basically, given that two request sets will always have at least one intersecting process, and that intersecting process cannot commit to both, so it is not possible for two processes to acquire the lock and essentially break mutual exclusion. Mutual exclusion is essentially a long form for mutex. So it is not possible for this to happen.

(Refer Slide Time: 26:42)


Tokenless Algorithms
Token Based Algorithms
Ricart-Agarwala Algorithm
Maekawa's Algorithm

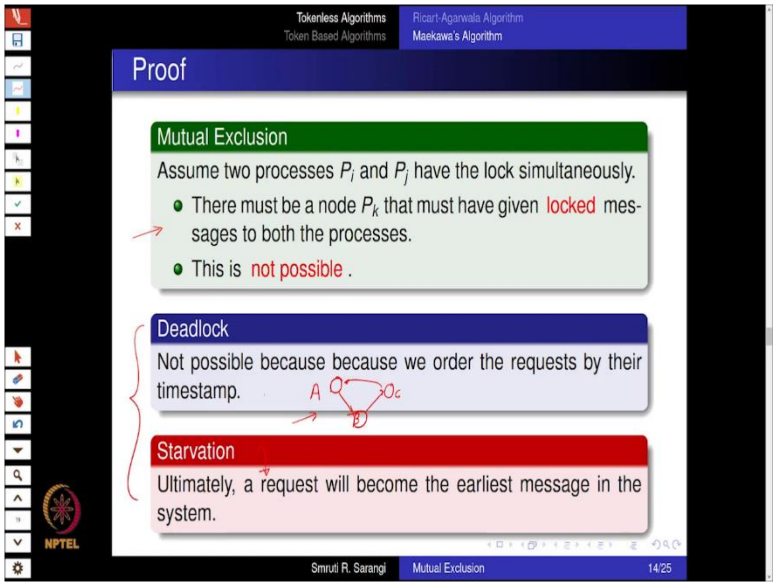
Releasing the Lock

- A process sends released messages to all the processes in its request set.
- A process in the request set, locks itself for the earliest request in the wait queue. It sends it a locked message.
- If there is no such request, then it marks its status as unlocked .

NPTEL
Smruti R. Sarangi
Mutual Exclusion
13/25

Releasing the lock. Well, once a process is done, it sends release messages to all the processors in its request set. A process in the request set locks itself for the earliest request in the wait queue, and it sends a locked message, it sends it a lock message. If there is no such request, it marks its status as unlocked. So this part is easy, and this has been a common feature in, so this was there in the Lamport's algorithm as well.

(Refer Slide Time: 27:15)



Proof

Mutual Exclusion

Assume two processes P_i and P_j have the lock simultaneously.

- There must be a node P_k that must have given locked messages to both the processes.
- This is not possible .

Deadlock

Not possible because because we order the requests by their timestamp.

Starvation

Ultimately, a request will become the earliest message in the system.

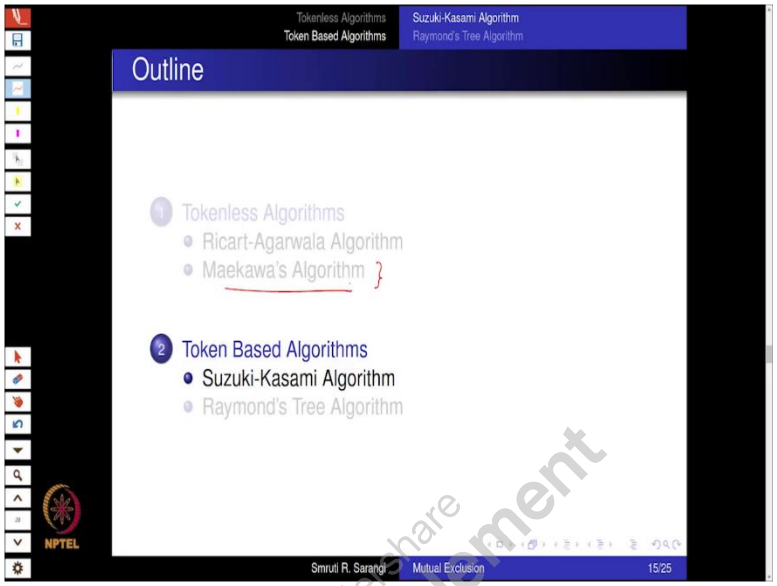
Proof. So the proof for mutual exclusion is easy. It is basically assume two processes P i and P j have the log simultaneously, which means there has to be a node P k that must have given locked messages to both at the same time. That is not possible. So as we have seen, if let us say I am giving a locked message to P i, I will have to withdraw the lock message from somebody else. This is not possible.

Given the fact that this is not possible, mutual exclusion is always guaranteed. But there are two more important things. What we need to look at is we need to look at deadlock and starvation, in the sense is it the case that Process A is waiting for Process B, Process B is waiting for Process C, Process C is again waiting for Process A. So do you have a circular wait?

If you have a circular wait, so why are deadlocks important? Because in any algorithm where we wait, the chances of a circular wait are always there. That was not there in a Lamport's algorithm because we never waited. But in algorithms where we actually wait, the chances of a circular wait are definitely there. And, so there is a need to look at deadlocks.

And the other is the issue of starvation, which means that is it possible that there is one message but it starves, in the sense it never gets the locked. So we will look at them. We are saying it is not possible, but we will look at them in some detail.

(Refer Slide Time: 28:49)



Tokenless Algorithms | Suzuki-Kasami Algorithm
Token Based Algorithms | Raymond's Tree Algorithm

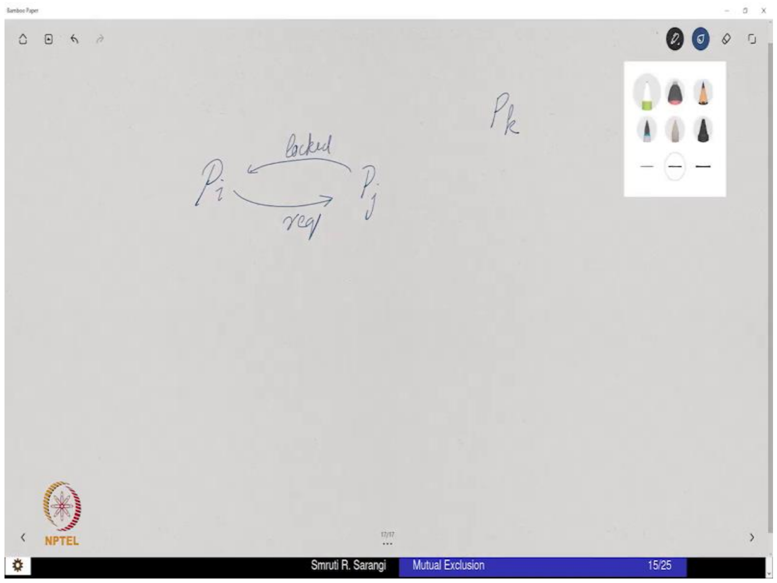
Outline

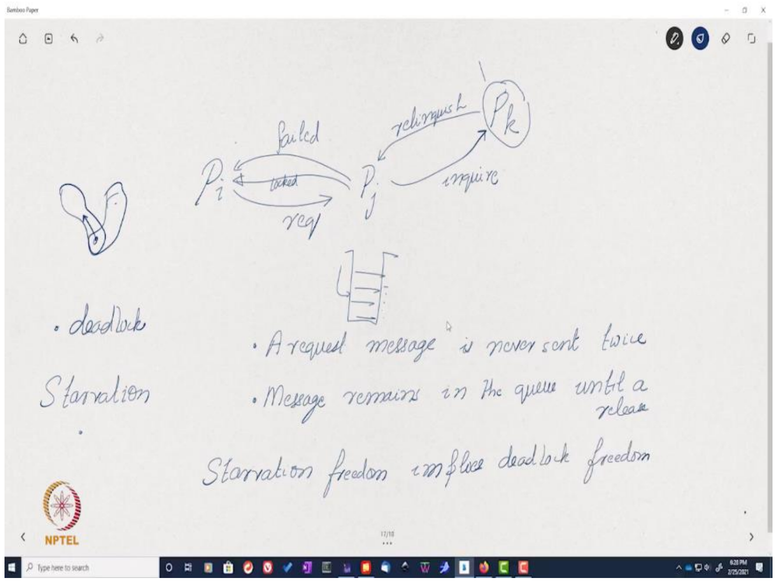
1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm }
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Smruti R. Sarangi | Mutual Exclusion | 15/25

Al so this finishes the algorithm. So we will now go for a deeper discussion of Maekawa. It finishes the slide part. So now we will go for a slightly deeper discussion using our e notebook.

(Refer Slide Time: 29:05)





• deadlock

Starvation

• A request message is never sent twice

• Message remains in the queue until a release

Starvation freedom implies deadlock freedom

So what, again, was the idea? We had three actors, P I, P j and P k. What was P i doing? It was sending a request. If it was, if P j was unlocked, it was sending an immediate commitment that okay, I am locked for you. But that is not clear, that is clearly not the simple case and then we are not looking at this because let us consider the more difficult case where our good friend P j has already committed to P k.

So then, in all cases within its request queue, it always puts in P i's request. So the requests are not sent once again. Requests are sent only once. They are clearly not sent once again. So even though the request is there, in here it sees the priority of this request with respect to all the requests that it has.

If it finds that this request is the lowest timestamp, it is a time to tell P k that look, I have received a request with a lower timestamp from P i. And, so basically you kindly tell me what you are up to. And let us say, if this is not the case in the sense there are other requests including the request from P k that have a lower timestamp or a higher priority, then of course a failed message is sent and P i is told that look, I cannot do it immediately.

So there is clearly no waiting over here in the sense that the failed message is sent immediately but the inquire message is also sent immediately, but as far as P i is concerned, it is not going to get a reply from P j immediately. It can get locked and failed immediately but not others. We will see why.

So when it asks P k to respond, if P k is holding the lock, of course, it will not respond. But let us say that P k has also requested a couple of people in its request set. Nobody has sent a failed message, it will just defer the reply, it will simply differ. And let us say if somebody says that look, I cannot satisfy your request, then of course, what comes back is a relinquish.

So then P k also records the fact that P j is not locked for it then P j is now ready to lock itself for any other message. So it takes a look at all the messages in its request queue which now include the request from P i and P k as well. So let us say the one with the lowest timestamp is P i at this point. So then it sends it a lock message and it locks itself for it. Or let us say it sends a locked message to somebody else and ultimately P i's request will become the one with the lowest timestamp at some point.

So what we will discuss now is, this is the basic idea, so some key points, a request message is never sent twice. A request message is never sent twice. So that is important. So once I send a request message to everybody in my request set, it remains in their queues. So even if they tell me failed, they tell me failed that means that I can, if I am P i, I can go and lock myself for somebody else, but, but essentially, that is it. But a request message is never sent multiple times. That is point number one.

After that, if you think about it, everything, so a message remains in the queue, so a message remains in the queue until I release. So until you do not release it, it remains in the queue. So it will continue to remain in the queue. There is no issues. What will happen is that mutual exclusion guaranteeing is not a big deal at all because the reason is that any node at one point of time, so node means the process here, is locking itself only for one other process.

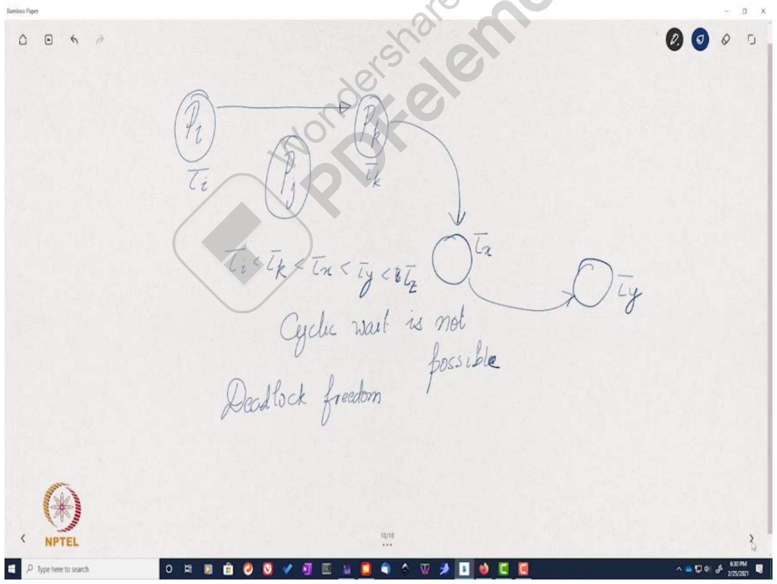
So, it will never be the case that actually two processors have acquired the lock at the same time because in their request sets, one intersection will be there, and that guy would have only committed to one, not the other. So that is the reason one of those processes would not have gotten a locked message from everybody in its request set. So mutual exclusion is guaranteed.

But what we should look at are two things. One is deadlocks. So deadlock, as I said, is a circular wait. And why? Because we are waiting over here. If P k has not gotten any failed message, we are essentially waiting. So since P k is waiting, P j is waiting, since P j is waiting, P i is waiting. So we have a wait over here, P i, P j and P k.

The other is starvation. Starvation means that I have a request, but the request is simply never getting the lock. I might not have a deadlock, there is no cyclic wait, but I am simply never getting the lock. So there is an interesting result that is there which is that starvation freedom, if I can show, that I never starve, in a sense if I am there in the system I will ultimately get the lock. Starvation freedom implies deadlock freedom, is very important.

Deadlock freedom does not imply starvation freedom, but starvation freedom implies deadlock freedom.

(Refer Slide Time: 35:38)



So let us look at proving this. So if you can prove it is starvation free, of course, it is much easier. But we will take a slightly different route. So let us understand what exactly is happening. So we have P i, we have P j in this case, and we have P k. So P j, as far as we are concerned, is incidental. So pretty much, P i is waiting on P k. So it basically wants P k to relinquish such that P j can give P i the locked message.

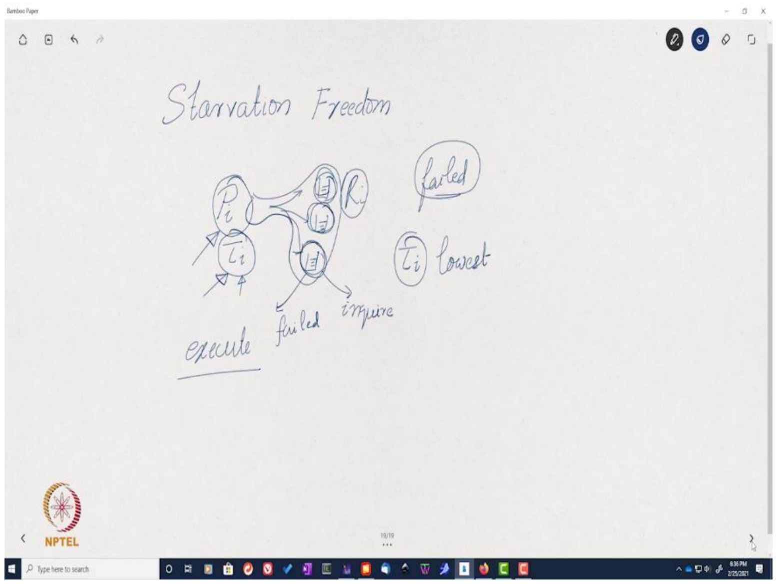
So what is the relationship between the priority, or let us say, the time stamp of i and k? Well clearly, P i is waiting on P k, the primary reason being that P i has higher priority or it has a lower timestamp. So pretty much, this holds. So, if P k is waiting on some other process, does not matter what it is, so again, this will also hold. And similarly, if that is waiting on some other process, again, this relationship will hold.

So it is clear that because of this less than ordering, a cyclic wait is not possible. So this is clearly an ordering which is not symmetric in the sense that it is not possible to have some other process z, and that waiting on i. That is not possible, because as you can see this relationship, the less than relationship needs to hold. So a cyclic wait is not possible. So given that a cyclic wait is not possible, a deadlock is not possible.

So this basically means that when we have proven deadlock freedom, it automatically means that in the system, you will not have a situation where nobody is making progress and nobody is acquiring the lock. That will not happen. Somebody will. So that is very important that somebody will definitely acquire the lock.

So now, let us prove starvation freedoms. As we have said, starvation freedom implies deadlock freedom, not the other way. So we did not find a good way of proving this protocol to be starvation free, but at least we have proven it to be deadlock free. So for starvation freedom, we do not have to do much, it is, this is actually easier. Given that we have proven deadlock freedom we can use some of it to prove starvation freedom as well.

(Refer Slide Time: 38:20)



So the key idea of starvation freedom is like this, that consider the process with, let us say, the least time stamp in the system. Let it be P i, and let this be its request set, R i. The moment it has sent a request to everybody, P i's request is present in the request set of everybody. Subsequently, any other message that they send has to be more than the timestamp of P i. But in any case, we are assuming that among all the requests, T i is the minimum. It has the least timestamp, or in other words it has the greatest priority. So this is what we are assuming.

Second, we are assuming that other requests are getting served because it is deadlock free. So at any point of time, some or the other requests will get served. Now, the question is, when will the time for this request come? So once the request message has been set, sent, it has been enqueued in all the queues. And we are sure, given the Lamport clock property, given the fact that we have this max operation, the timestamp always increases. It never reduces. So we will never have a situation in the future where a request will be sent with a lower timestamp. That is not going to happen.

Why? Well, because we have sent messages everywhere. Every message that is being sent in the entire system, this is the smallest time stamp. So clearly, in the future, no message will be sent with a timestamp less than this. Given that that has happened, let us just consider all the nodes in the request set, and that includes P i itself. So they will basically never send a lock, they will not send a locked message to any other request after getting the request from P i.

So that is clear. So that follows from our property, and that is mainly because this is the lowest in the system. So subsequently, if anybody else tries to send a locked message to this, all that it is going to get is it is going to get a failed message back. So it is not possible. So after a certain time, of course, let us say, all the messages which are there in the system may clear up, but ultimately, what will happen is, that all the nodes in the request set R i, will simply not participate in any other locking protocol.

And they will basically ask all the members of their request set to kindly send relinquish messages, and they will not participate in any new locking protocol. So when they get it, first if they have locked themselves for somebody else, they will immediately send an inquire message asking it to release the lock. And subsequently, they will not entertain any newer message, they will keep on sending failed messages, and because it is deadlock free, gradually, the messages are draining out from the system.

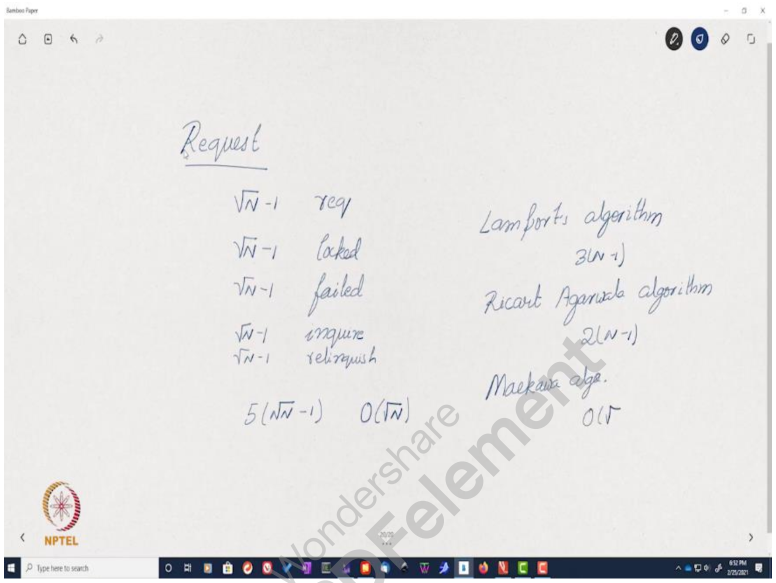
So ultimately, it will be the case that this request, the, the, the request from Process i is pretty much at the head of every queue and all of them have sent a locked message back because they are not sending a message to anybody else, and the system has to make progress, it is deadlock free, no other request is able to make progress, a time will come because all of their timestamps are greater than T i.

And furthermore, so consequently they have a lower priority, and they will not be able to make progress and they will also not be able to lock the nodes in the request set because the request set will not accept any requests, and then the request set, the nodes of the request set will continue to send locked messages to P i, and ultimately, it will get all the locked messages and will start to execute.

So this means that this protocol is starvation free because it is never possible that one request will wait for eternity to actually get the lock. Why? Because ultimately, it will become the liquid, request with the least time stamp, and all the notes in its request set will either inquire and ask for relinquished messages and stop any subsequent requests by sending failed messages. So the victory is for this process.

So this is why starvation freedom is avoided. And so this completes the proof that this protocol is both deadlock free as well as starvation free.

(Refer Slide Time: 43:45)



Request

$\sqrt{N}-1$ req

$\sqrt{N}-1$ locked

$\sqrt{N}-1$ failed

$\sqrt{N}-1$ inquire

$\sqrt{N}-1$ relinquish

$5(\sqrt{N}-1)$ $O(\sqrt{N})$

Lamports algorithm

$3(N-1)$

Ricart Agarwala algorithm

$2(N-1)$

Maekawa algo.

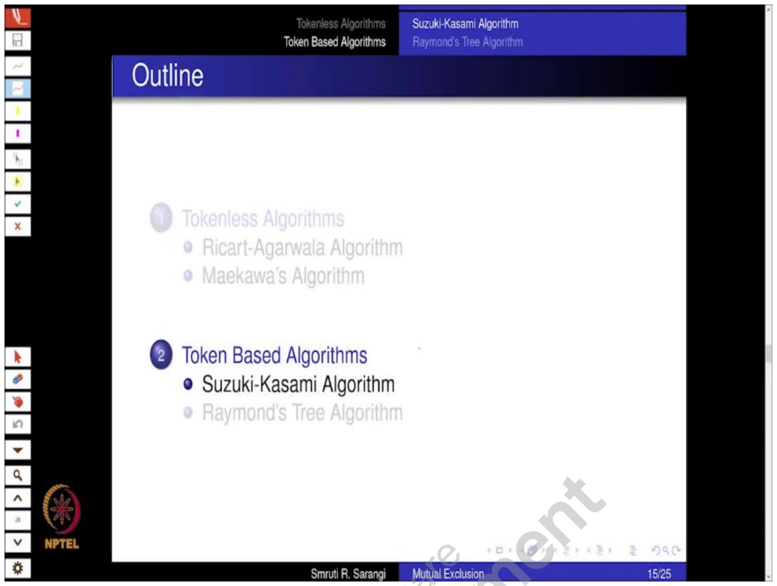
$O(\sqrt{\ }$

Let us now compute the number of messages that are sent. So for a single request, we need to send $\sqrt{N}$ - 1 request messages. That is to every node in the request set. We need to send $\sqrt{N}$ - 1. So we will receive square root of N minus 1 locked messages. If, let us say, our luck is really bad, we might also need, we might also receive these many failed messages.

Furthermore, if, let us say, the priority of the requester is very high, then a maximum of, again, these many inquire messages will be sent. The reason is that, let us say, at every P j in its request set, if it has the highest priority and that is locked by some other process then an inquire message will be sent, and it is possible that a maximum of these many relinquished messages will come back.

So, we can say that the maximum number of messages is 5($\sqrt{N}$ – 1), which is conveniently put as order of $\sqrt{N}$. So, what we have done is, we have done significantly better from 3(N – 1), we came down to 2(N – 1), to order of $\sqrt{N}$. So, the Lamport's algorithm was this much. Then, the Ricart-Agarwala algorithm was this, and then we have the Maekawa algorithm, which is order of $\sqrt{N}$.

(Refer Slide Time: 46:10)



Tokenless Algorithms | Suzuki-Kasami Algorithm
Token Based Algorithms | Raymond's Tree Algorithm

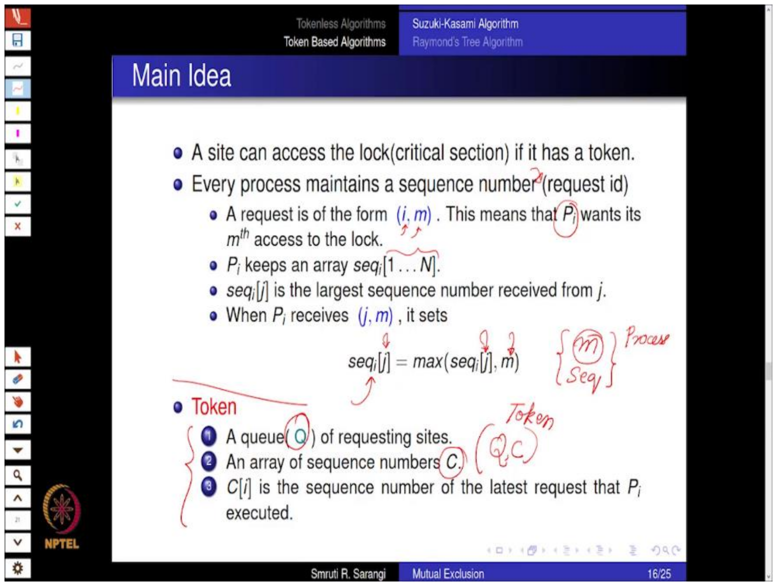
Outline

1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Smruti R. Sarangi | Mutual Exclusion | 15/25

We will discuss the two token based algorithms, the Suzuki-Kasami Algorithm and the Raymond's Tree Algorithm. So they differ in terms of their philosophy as compared to the previous algorithms.

(Refer Slide Time: 46:23)



Tokenless Algorithms | Suzuki-Kasami Algorithm
Token Based Algorithms | Raymond's Tree Algorithm

Main Idea

- A site can access the lock(critical section) if it has a token.
- Every process maintains a sequence number (request id)
  - A request is of the form (i, m). This means that P_i wants its m^{th} access to the lock.
  - P_i keeps an array $seq_i[1 \dots N]$.
  - $seq_i[j]$ is the largest sequence number received from j.
  - When P_i receives (j, m), it sets

$$seq_i[j] = max(seq_i[j], m)$$

$\left\{ \begin{matrix} m \\ Seq_i \end{matrix} \right\}$ Process

- Token
  1. A queue (Q) of requesting sites.
  2. An array of sequence numbers C.
  3. $C[i]$ is the sequence number of the latest request that P_i executed.

Token (Q, C)

Smruti R. Sarangi | Mutual Exclusion | 16/25

So, the idea over here is that a site can access the critical section, a process can access the critical section if it has a token. So, every process maintains a sequence number or a request ID. So, a request is of the form, it is a tuple of (i , m). So i means that, i refers to process P i. It is the process ID i. And m is the sequence. It says that it needs the mth access to the lock. So this does take some amount of past history into account but not much.

Essentially, every request in this case is a tuple of the process ID, and the access number of the lock, in the sense, is it your third access or fourth axis or fifth access to the lock. Furthermore, every process P i keeps an array of, a sequence array 1 to N where the jth entry is the largest sequence number it has received from j. So, think of this as some sort of a vector clock. So the idea is similar to a vector clock. We were using scalar clocks earlier.

So, this has been inspired from the vector clock. So, when P i receives a message (j, m) from Process j, what it basically does is akin to what we do in a vector clock, it sets the jth entry as the maximum of what it had, and the value that Process j is sending. So, what is the idea? The idea is that you have a scalar clock which is the sequence, which is m in this case, and you have a vector clock, a sequence array.

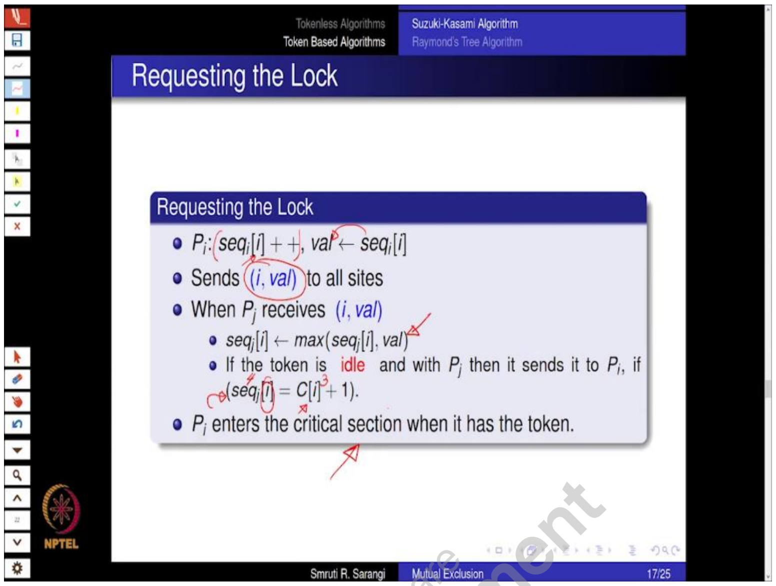
So, this is the state that a process keeps. What is the token? It is a queue of requesting sites, sites are the same as process over here. So, it is a queue and it is an array of sequence numbers, so, we will see what they do, Array C. So C i is the sequence number of the latest request that P i executed. So basically, it is basically saying that, let us say, if C i = 3, this means that P i, Process i just finished the third access to its lock.

Just finished means finished in the past, the third access to its lock. So, this is the arrays C i. So we have many data structures here. So, I would request you to kindly go through this piece of the video many times to memorize what these are. So, for a process, we have its internal variable m, which is its, essentially its lock count, it is a scalar clock.

Seq is a vector clock, and then for the token, we have two fields, one is the queue of requesting processes and other is an array of sequence numbers C, capital C, where C i is the sequence number of the latest request that the ith process executed, ith process finished.

So, there are these four things for a token Q and C, and for a process m and seq. So let us not forget these, m and seq for a process, Q and C for a token, not to be forgotten.

(Refer Slide Time: 49:45)



Requesting the Lock

Requesting the Lock

- P_i: $seq_i[i]++$, $val \leftarrow seq_i[i]$
- Sends (i, val) to all sites
- When P_j receives (i, val)
  - $seq_j[i] \leftarrow max(seq_j[i], val)$
  - If the token is idle and with P_j then it sends it to P_i, if $(seq_j[i] = C[i] + 1)$.
- P_i enters the critical section when it has the token.

So to request the lock, what you do is that, so this is very similar to a vector clock. It increments its ith entry. So, it pretty much implement, increments m. So, in the vector clock, it says that look, I am going to request, so it increments that entry. And the current value, it puts a temporary variable val, and this I, val is sent to all the sites. So, it is basically saying that look, I am Process i, I just incremented my internal scalar count. Now it is val. And I need access to the lock.

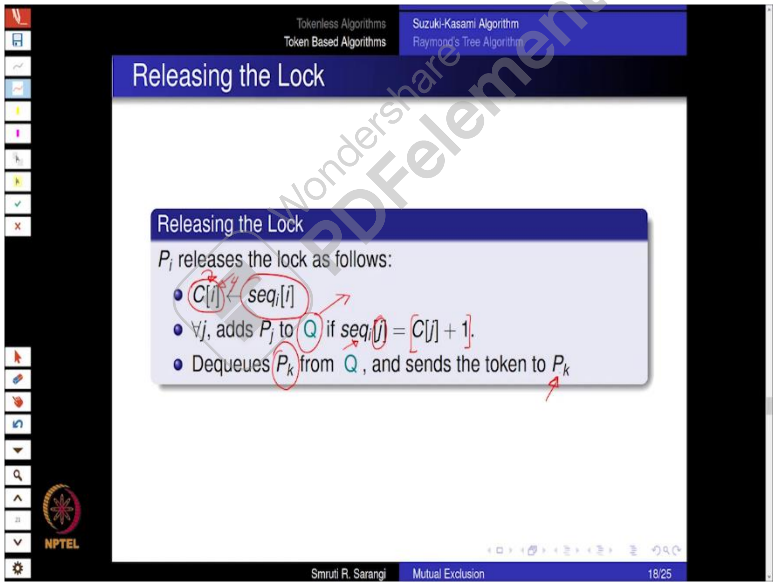
When P j receives i val, what does it do? Well, it does the standard thing that it increments its vector clock. So it sets the ith entry as the max of seq, j, i and val. If the token is idle, in the sense if the token is idle, meaning that, we will see what it means, but let us say it is not being used and it is there with P j, then it sends the token to P i, if, if, so this if is important, if seq j I, which means as far as Process j is concerned, the value of i that it is getting, which was essentially set by this operation, = C [i] + 1.

Which basically means that look, the token is recording that Process i has finished three requests and now the fourth request is coming. 4 = 3 + 1, which means it is the next request, which means the token should be sent. So, it means it is not a stale message, it is not one of the old messages floating around, and this is genuinely a fresh request because this process has already accessed the lock three times. It wants to access for the fourth time, so access may be given.

So, the token goes to a process, then it enters the critical section, which means it grabs the lock. So, in this case establishing mutual exclusion is rather trivial because if you have the token then you enter the lock and grab the resource, otherwise you do not. And given the fact that only one process has a token at any point of time, mutual exclusion is trivial.

So, we basically have to look for starvation freedom and deadlock freedom. And I want to repeat that we were able to prove in the previous lecture that starvation freedom implies deadlock freedom.

(Refer Slide Time: 52:23)



Tokenless Algorithms | Suzuki-Kasami Algorithm
Token Based Algorithms | Raymond's Tree Algorithm

Releasing the Lock

Releasing the Lock

P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q, and sends the token to P_k

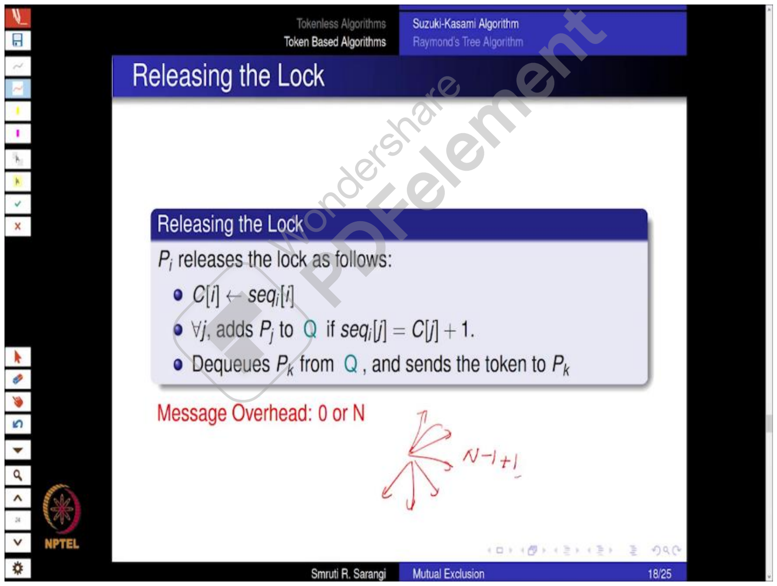
Smruti R. Sarangi | Mutual Exclusion | 18/25

Releasing the lock. Again, very easy. P i releases the lock as follows. So it sets the ID C [i]← $seq_i[i]$ as the ith entry, which means that if, let us say, it finished the fourth access to the lock, within the C array of the token, it is going to add 4. As simple as that. Furthermore, furthermore, for ∀ j, it adds Pj to Q, which is the array within the token, which is the queue within the token, as long as if $seq_i[i]$, which means that in i's vector clock we will look at the jth entry, if this is equal to C [j] +1.

Which means that it is aware of the fact that in some other Process j, has already acquired the log C j times, but it is desirous of accessing it once more. So what it does, and let us say it gets to know that, because messages are being sent around, so j may be, send some message to k and then k send it to i. So that is all right. So, it essentially scans its sequence array whenever it sees this relationship to hold, which means that Process j is desirous of accessing, acquiring the lock for one more time. It adds it to the array Q, the Q queue.

Then what it does is, it dequeues P k from Q and so it dequeues one process from the queue and it sends the token to P k. So to, for starvation freedom, pretty much, if this is a first in, first out queue, so it is guaranteed that as long as the process enters the queue, it will be dequeued and the token will go to it, and it will access the resource.

(Refer Slide Time: 54:28)



Releasing the Lock

Releasing the Lock

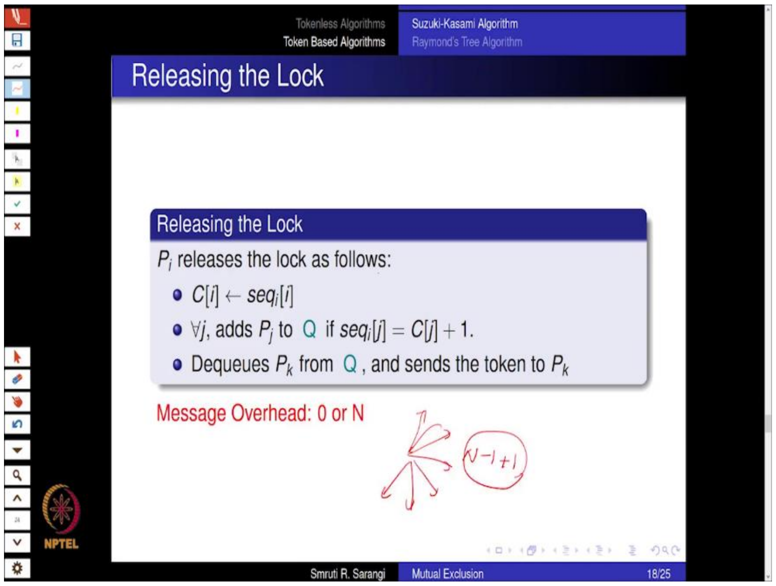
P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q, and sends the token to P_k

Message Overhead: 0 or N

So what is the message overhead? Well, the message overhead is that look if I am the process, I have the token, I immediately grab it, otherwise it can be N. So N basically means that pretty much I send a message to (N – 1) other processes, and then ultimately, one of the processes sends a token back to me. So it is N - 1 + 1. As simple as that.

(Refer Slide Time: 55:01)



Releasing the Lock

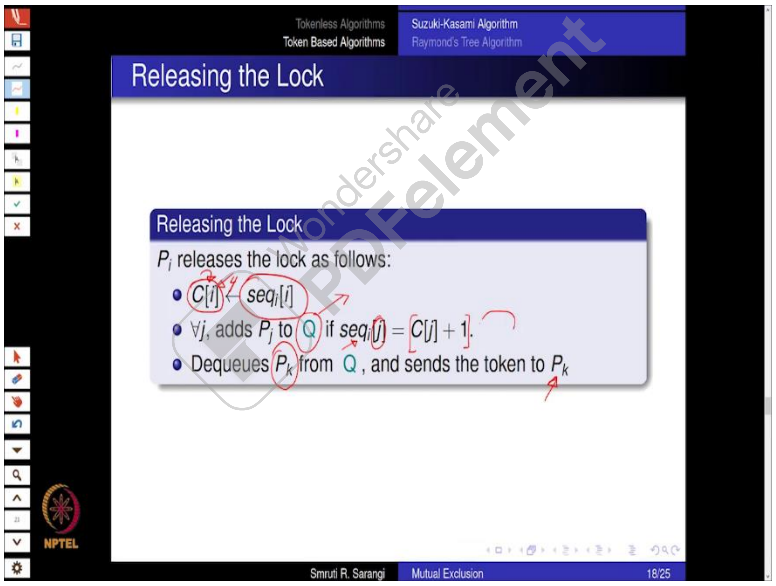
Releasing the Lock

P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q , and sends the token to P_k

Message Overhead: 0 or N

$N-1+1$

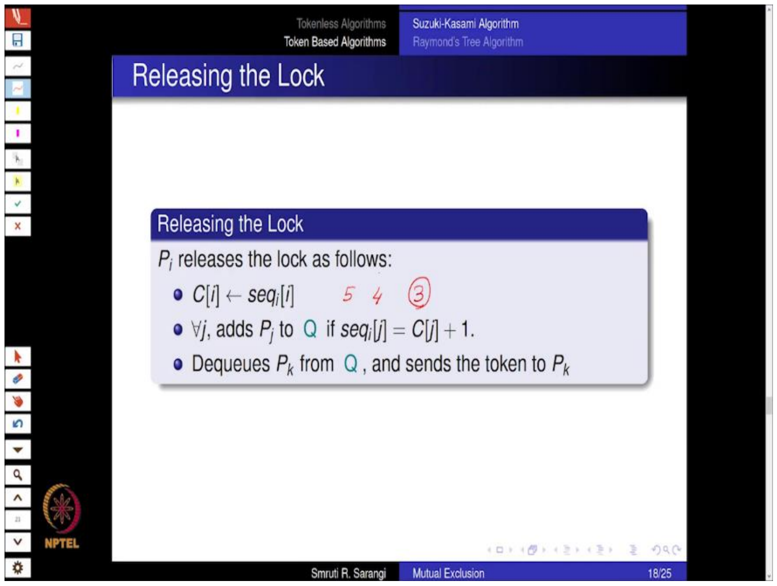


Releasing the Lock

Releasing the Lock

P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q , and sends the token to P_k



Releasing the Lock

Releasing the Lock

P_i releases the lock as follows:

- $C[i] \leftarrow seq_i[i]$ 5 4 3
- $\forall j$, adds P_j to Q if $seq_i[j] = C[j] + 1$.
- Dequeues P_k from Q, and sends the token to P_k

So, why do I say that the requesting process gets the lock in finite time? Because the request will reach all the processes in finite time. So by induction, one of these processes will have the token and finite time because if I do not get it, if I assume that in an N - 1 node system, one of them gets in finite time, if that is assumption that I make, one of them will get it, that process would have seen my request.

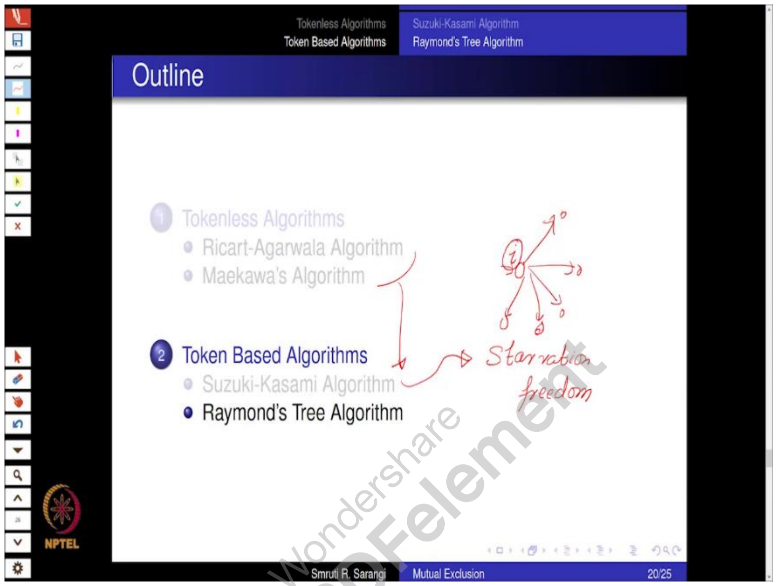
So my request will get added to the q queue by precisely which line, by precisely this line, which means that if I have sent a request, if I have sent a request and if the request either reaches directly or indirectly, it does not matter, but as long as it reaches any node which currently has the token, it is bound to add me to the queue.

And sooner or later my time will come, and at max, N processes will be there before me, N - 1, rather, and each one of them would have sent a message, N - 1 messages slash processes, that is the way that you interpret this line, and there will be N - 1 messages slash processes before it. So they will finish and my time will come. So clearly, I cannot starve. So there is a built-in mechanism where I cannot starve because the moment that a process sends a request, it gets added to the queue.

And furthermore, a process can also not swap the network by sending multiple messages because of precisely this little check that is done over here, which, so this little check basically says that, let us say, that you have acquired the lock three times and then in the next message, you send four. Then, only one will be added. If you send one more and then, let us say, it becomes five, then you lose your chance. In a sense, you get disqualified.

So you can only have one outstanding request at any point of time. This also stops starvation to a large extent, in this algorithm.

(Refer Slide Time: 57:17)


Tokenless Algorithms
Token Based Algorithms
Suzuki-Kasami Algorithm
Raymond's Tree Algorithm

Outline

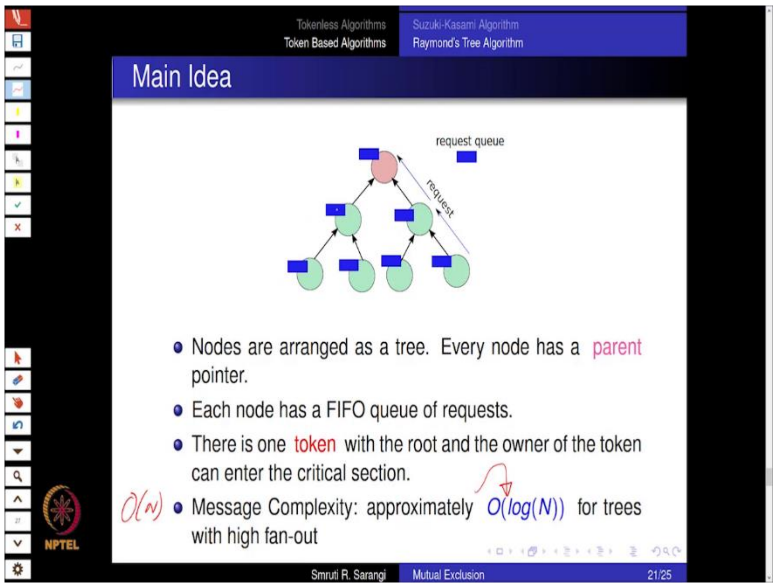
1. Tokenless Algorithms
   - Ricart-Agarwala Algorithm
   - Maekawa's Algorithm
2. Token Based Algorithms
   - Suzuki-Kasami Algorithm
   - Raymond's Tree Algorithm

Smruti R. Sarangi
Mutual Exclusion
20/25

So in this algorithm, proving starvation freedom was very trivial. So, that is the reason I am not discussing this any further because this kind of extends what we did in the previous two algorithms, Ricart-Agarwala and Maekawa. And starvation freedom is not there, primarily because any process will basically send messages to the rest of the processes. The token has to be there with somebody.

So the token, it is not possible that the token cannot be there with anybody. Otherwise, the token is with this process, then there is no problem at all. If the token is there with somebody, it will see this request, it will add this process to the queue and be rest assured, after a maximum of N - 1 processes finish the request, the originating process will get the token.

(Refer Slide Time: 58:20)



Tokenless Algorithms
Token Based Algorithms
Suzuki-Kasami Algorithm
Raymond's Tree Algorithm

Main Idea

- Nodes are arranged as a tree. Every node has a parent pointer.
- Each node has a FIFO queue of requests.
- There is one token with the root and the owner of the token can enter the critical section.
- Message Complexity: approximately $O(log(N))$ for trees with high fan-out

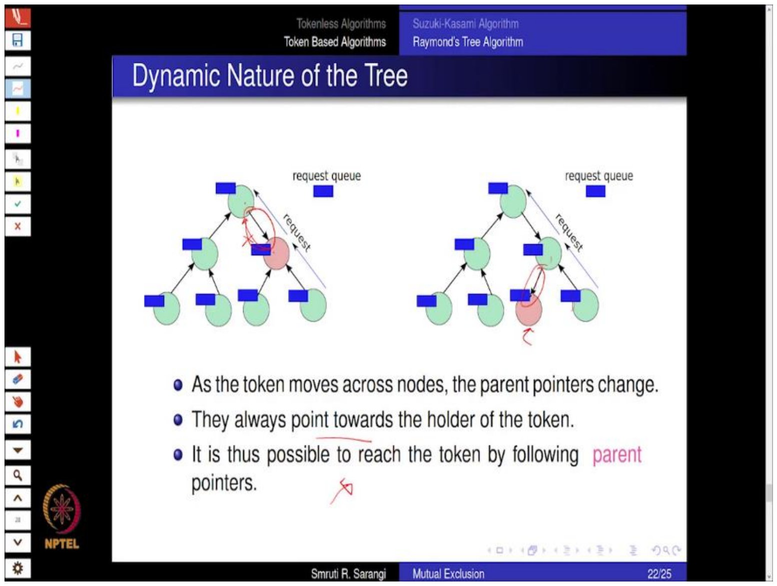
$O(N)$

Smruti R. Sarangi Mutual Exclusion 21/25

Now, let us discuss one more approach, Raymond's Tree Algorithm. So, here the idea is that nodes are arranged as a tree. Every node has a parent pointer. Furthermore, each node has a FIFO queue of requests. There is one token with the root and the owner of the token can enter the critical section, which means grab the lock.

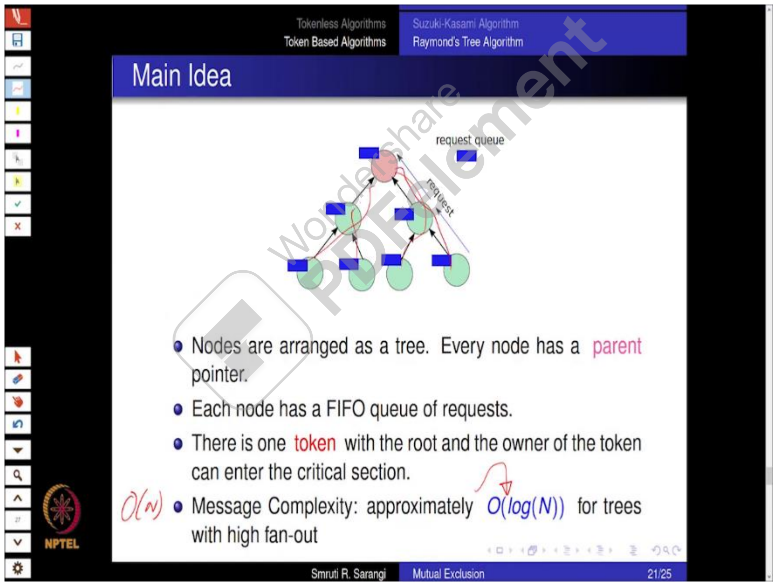
So in the previous case, we said that the message complexity is either 0 or N. So let us say worst case is order N. So this is reducing O(N) to O(log(N)) for trees with a reasonably high fan-out. That is fine. But if we take a look at this tree, what we can see is a very simple arrangement. So, let us assume it is a nice and balanced tree. Furthermore, every node has a parent pointer, a FIFO queue at each node.

(Refer Slide Time: 59:13)



Dynamic Nature of the Tree

- As the token moves across nodes, the parent pointers change.
- They always point towards the holder of the token.
- It is thus possible to reach the token by following parent pointers.



Main Idea

- Nodes are arranged as a tree. Every node has a parent pointer.
- Each node has a FIFO queue of requests.
- There is one token with the root and the owner of the token can enter the critical section.
- $O(N)$ Message Complexity: approximately $O(log(N))$ for trees with high fan-out

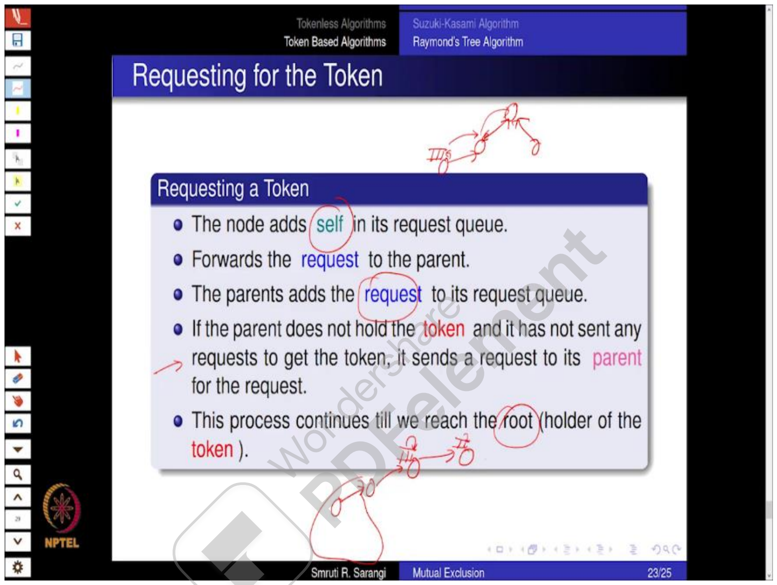
So, let us see how this works. So the way this works is that as the token moves across the nodes, the parent pointer changes. So let us say the token moves from here to here, what you actually see is that, so just look at the previous thing. All the notes were essentially pointing towards the root, and the root contained the token. In this case, the root has moved because the token will always there with, will always be there with the root.

So pretty much, this arrow over here got reversed. Previously it was pointing in this way, but it got reversed. Now, as you can see, still all the nodes point towards the token holder.

And wherever the token goes, let us say, next time the token goes over here, so as you can see this arrow also got reversed. So pretty much all the nodes can approach the node that contains the token. Of course, the arrows do get reversed as the token moves.

So, they always point towards the token holder. So, it is always possible to reach the token by following your parent pointers. Always, your parent pointers will take you towards the token. That is the most important learning over here.

(Refer Slide Time: 60:23)



Requesting for the Token

Requesting a Token

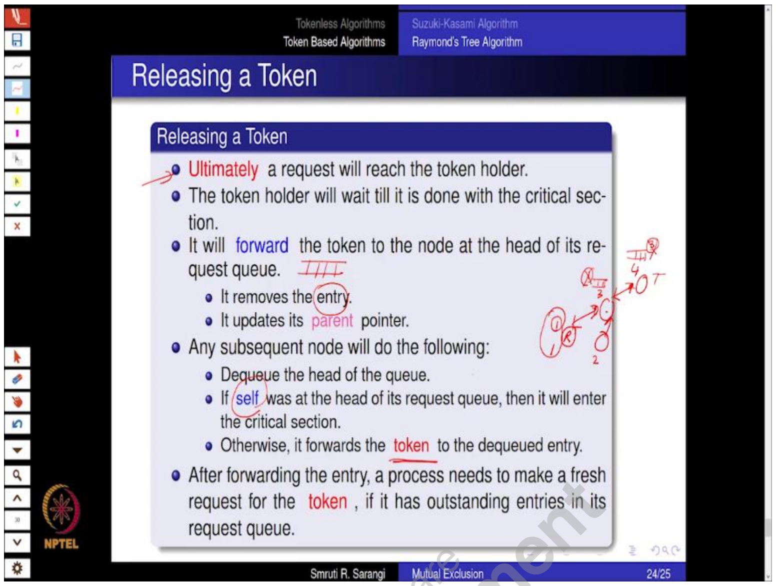
- The node adds self in its request queue.
- Forwards the request to the parent.
- The parents adds the request to its request queue.
- If the parent does not hold the token and it has not sent any requests to get the token, it sends a request to its parent for the request.
- This process continues till we reach the root (holder of the token).

Requesting for a token. So what it does is that within its own request queue, the node add itself, adds itself. So what a node would do is, within its own FIFO queue, it will add itself if it is requesting for the token. It will also forward the request to its parent. The parent will add the request of the child to its request queue.

Furthermore, if the parent does not hold the token, and it has not sent any request to get the token on behalf of any other child or itself, it sends a request to its parent and so on. So this process continues till we reach the token holder, which is the root. So the idea is very simple. I want to get the token; I insert a request into my FIFO queue. So far, so good.

Subsequently, what do I do? Subsequently, what I do is I send a request to my parent. And then, my parent basically sees whether it has sent any request on behalf of any other child of its or not. If not, then it sends a request to its parent. Its parent again inserts the request in its own FIFO queue. Again, sends a request to its parent, it does the same. Ultimately, we reach the token holder, which is the root.

(Refer Slide Time: 62:03)



Tokenless Algorithms | Suzuki-Kasami Algorithm
Token Based Algorithms | Raymond's Tree Algorithm

Releasing a Token

Releasing a Token

- Ultimately a request will reach the token holder.
- The token holder will wait till it is done with the critical section.
- It will forward the token to the node at the head of its request queue.
  - It removes the entry.
  - It updates its parent pointer.
- Any subsequent node will do the following:
  - Dequeue the head of the queue.
  - If self was at the head of its request queue, then it will enter the critical section.
  - Otherwise, it forwards the token to the dequeued entry.
- After forwarding the entry, a process needs to make a fresh request for the token, if it has outstanding entries in its request queue.

Smruti R. Sarangi | Mutual Exclusion | 24/25

The token holder will wait till it is done with the critical section. I mean, till it is done with the lock, it will wait. Subsequently, the token holder will take a look at the queue of the root node, which is the road, node that holds the token. Then, the head of the queue, whatever node it is, it will remove the entry, and it will update its parent pointer.

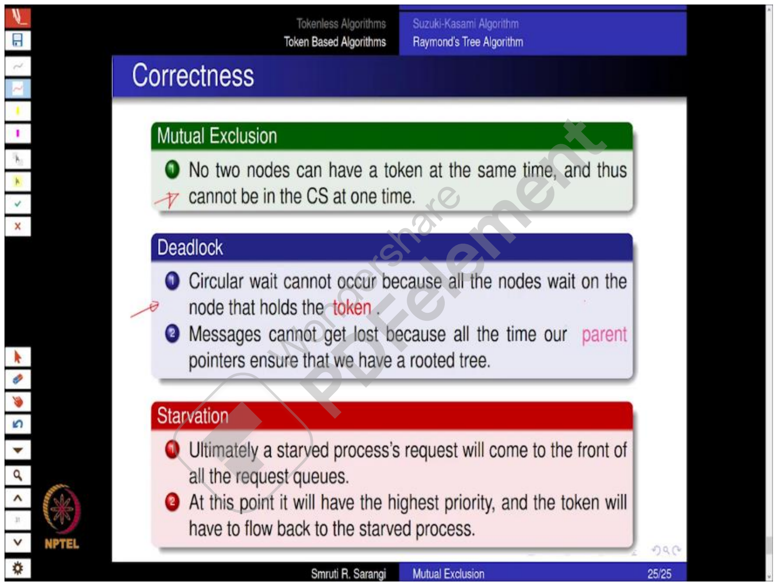
So basically what it will do, what it will do is that it will remove the entry, send the token to that entry and update its parent pointer. So any subsequent node will do the following. It will dequeue the head of the queue. If self was at the head of his request queue, then it will enter the critical section, otherwise it will forward it to the dequeued entry. So let us give an example. So let us say that this is the way that we had. And let us say this was the original requested node. And let us number these nodes, 1, 2, 3 and 4.

So then, what happened is that this node added itself, 1, to it, and it sent a request to its parent. So the parent, let us say, was not interested. So it also added node number 1 to it, and it sent a request to its parent, and let us say this was a token holder. So it added node number 3 to it because as far as it is concerned, node number 3 is requesting.

Then, once the token was released, what it did is that it said okay, 3 is at the head of my queue, so then it reversed the direction of this arrow, and it made 3 the token holder. 3 did the same thing. It said okay, all so 1 is at the head of my queue, so let me make 1 the token holder. So, it reverses the direction of this pointer, 1 becomes a token holder. And 1 sees that it is a token holder as well as 1 is at the head of its queue, so it uses the value.

If self was at the head of the request queue, then it will enter the critical section, otherwise it will forward it to the dequeued entry. So, after forwarding the entry, a process needs to make a fresh request for the token if it has outstanding entries in its request queue.

(Refer Slide Time: 64:34)



Tokenless Algorithms | Suzuki-Kasami Algorithm
Token Based Algorithms | Raymond's Tree Algorithm

Correctness

Mutual Exclusion

1. No two nodes can have a token at the same time, and thus cannot be in the CS at one time.

Deadlock

1. Circular wait cannot occur because all the nodes wait on the node that holds the token.
2. Messages cannot get lost because all the time our parent pointers ensure that we have a rooted tree.

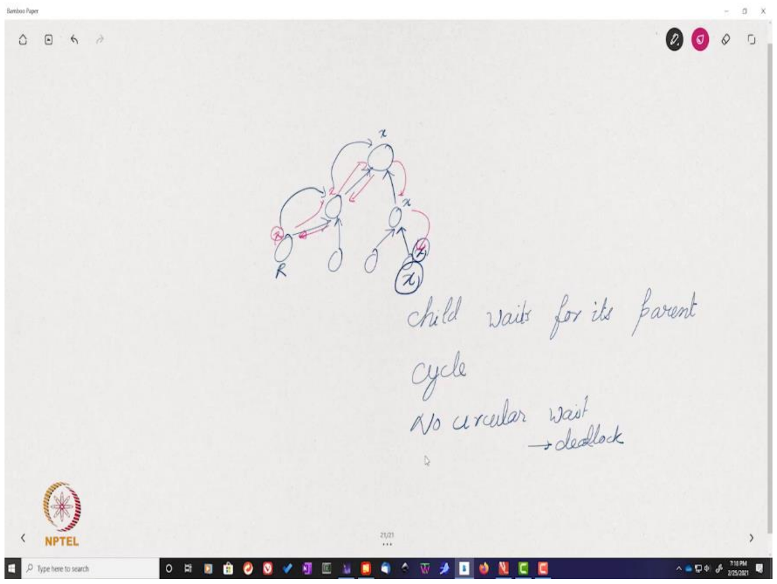
Starvation

1. Ultimately a starved process's request will come to the front of all the request queues.
2. At this point it will have the highest priority, and the token will have to flow back to the starved process.

Smruti R. Sarangi | Mutual Exclusion | 25/25

So, mutual exclusion is very simple in any token based algorithm because the idea is that no two nodes can have a token at the same time. Deadlock. Circular wait cannot occur because all the nodes wait on the node that holds the token. And furthermore, we are not losing messages and we will always have a rooted tree. And given the fact that we have a rooted tree, what will happen is as follows.

(Refer Slide Time: 65:06)



Let me go to our paper over here. So pretty much what is happening is, if, let us say, this node is a requester, it is waiting on this node, and then this node is waiting on this node, but it is clearly not possible. So always as you can see these waits are going along this parent edges. So a child always waits for his parent, but a parent never waits for his child.

So let us say that the child always waits for its parent. So, this is always happening that, let us say, if I do not have the node, I am asking my parent for it. If I do not have the token, I ask my parent. If my parent does not have the token, it asks its parent. So on and so forth. But the point is you will never ask a child.

As a result, given the fact that the tree itself is a cyclic, you will never have a cycle of dependencies because you will have never have a cycle of parent pointers because you will not have a circular wait. No circular wait implies deadlock freedom. So this approach will not have a deadlock.

So now, what was left? Starvation. So ultimately, a starved process, this request, will come to the front of all the request queues. At this point, it will have the highest priority, and the token will have to flow back to the starved process. So let us take a look at this once again.

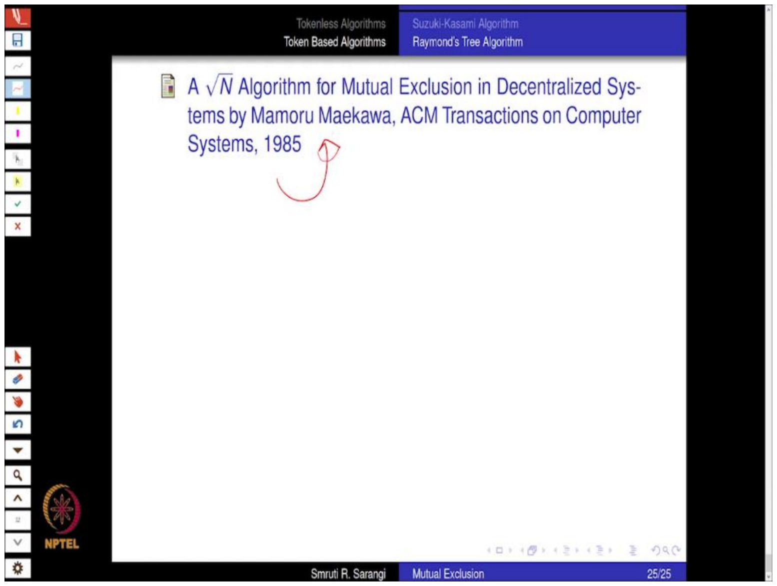
So let us say that there is one process, let us say this one. So what will happen is that this process, so let us call it Process x. It will insert x in its own queue, send a request to its parent, x will be over here, x will be over here. And let us say later on even if the parent pointers change, and let us say that the parent pointers go this way, that is all right. You will still have x over here, you will still have x over here.

So then, what will happen is the moment that the request is satisfied, so basically mind you, we are always adding x as long as there is no other outstanding request that has gone from, that has gone from this node to its parent. So then ultimately, when it will become the oldest, everybody will find x to be at the head of its queue, so it will dequeue x. Again, the token will flow this way, the token will flow this way, this way and this way.

So the advantage of any algorithm that has a queue is basically starvation freedom where we say that look, you will have a queue and ultimately you will come to the beginning of each queue, and when you come to the beginning of a queue, it is guaranteed that you will get it.

So basically, nobody can displace you from your position. And given the fact that the only direction in a queue that you can move is up, which means towards the head, ultimately you will come at the beginning of all queues, and you are bound to get the token, which in this case, will happen.

(Refer Slide Time: 68:40)



A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems by Mamoru Maekawa, ACM Transactions on Computer Systems, 1985

So given that we have seen this, token based algorithms are very simple. The key algorithm that you should focus on is the Maekawa's Algorithm. And this algorithm is kind of involved but I think we have discussed this in great detail. The token based algorithms are quite simple.